

1. Introduction

In this project, we have implemented a server protocol – from first principles – that seeks to achieve high availability and eventual consistency in light of a revised failure model. Specifically, the system now detects and robustly handles transparent server and client failure, as well as intermittent network (and hence TCP) connection breaks.

2. High-level system/failure model overview

i. Revised failure model

The revised failure model allows for the possibility of both:

a) Transparent server/client failures, where a node abruptly terminates before sending a quit message to other nodes. In our previous system, such failure meant affected clients could no longer register, log-in, or send/receive messages. And crucially, this loss of functionality was *unknown to the rest of the system*. Additionally, the crash of a server meant the loss of its client-data, hence rebalancing across servers could not occur, and the user details were permanently lost.

b) Intermittent network connection breaks, disrupting both server-to-server and client-to-server communication. This causes TCP breaks and thus thrown errors/message loss which, if not addressed, will lead to system inconsistency.

ii. Revised system architecture

To ensure high availability for clients, we have *flattened* our previously hierarchical system structure. This has removed the need for a designated master/coordinator node (that tracks the evolving global state), thus decoupling servers and eliminating central points of system failure. The resulting topology is a ‘full mesh’ where each node is connected dynamically and non-hierarchically to all others (seen in fig. 1).

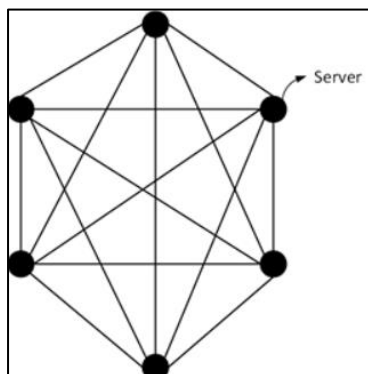


Figure 1.

To achieve this, we added an ‘agreement’ layer to the server-to-server communication protocol. Our technical implementation will be the primary focus of this report. But from a high-level view, this layer has sought to ensure: a consensus among active nodes regarding the global server- and user-lists (and thus allowing servers to join at any time); the detection of transparent server and client failure; and an eventually consistent message/event ordering between servers and messages sent from a single client.

In line with project specifications, we have optimized for client availability (such that logged-in clients can always send and receive messages with minimal delay) whilst ensuring an eventual consistency across clients and servers.

3. Establishing a flat system structure

i. Global-checkpointing

In a flat system, global consensus regarding the system state is crucial. Previously, we had used a tree structured server network to authoritatively track the evolving global state. But, now we need individual servers to receive a message about the network status at the time of joining the network. When a new server authenticates, it receives the following command:

```
{
    "command": "NETWORK_STATUS",
    "server_info": {.....},
    "user_info": {.....},
    "backlog_info": {.....}
}
```

Server_info contains a list of the IP and port numbers for all active servers in the system; User_info returns a hashmap of the user name (key) and secret (value) for each user; and backlog_info is a hashmap that tracks messages unseen by the client, such that a username (key) contains all messages (value) that were not received due to inactivity. Backlog_info is updated as client messages are received and then processed FIFO, ensuring that when a client becomes active, these messages are received in the correct ordering.

Crucially, *every server will store the same global network status*. For instance, when a new server successfully authenticates with an active server, the active server adds the new servers details to server_info and sends connection requests to all other active servers in the network and form a full mesh network.

Previously, horizontal expansion of the system would create a problem where new servers did not

have access to the pre-existing user list. NETWORK_STATUS solves this problem, allowing servers to join the network at any time and have access to an up-to-date global checkpoint.

ii. Locking

Previously, LOCK_REQUESTS were used strictly to ensure each registered username was unique. Namely, if a client requested to register a username, the server would enact a LOCK and query each individual node to check if that username was already registered on their server, as shown in fig 1. If the test failed, REGISTER_FAILED would be sent to the server and forwarded to the client. If the username was in fact unique, REGISTER_SUCCESS was received instead, causing the server to update its user list and forward that message to the client.

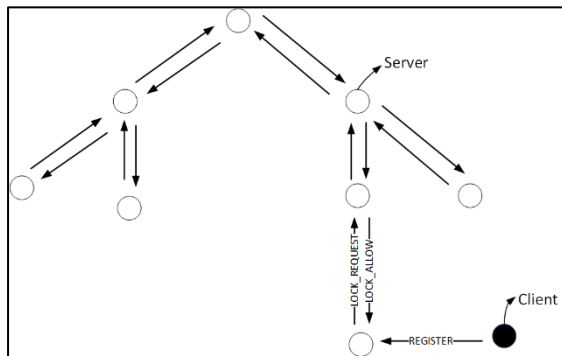


Figure 2.

Our newly flat system structure adjusts this process. Instead of lock requests and search query's progressively making its way up the hierarchy, NEW_USER commands are simultaneously sent to every server in the system (Figure:3).

This reduces the message complexity for user registration, since there is no need to query the user list of other servers. And if the outcome is successful, REGISTER_SUCCESS is forwarded to the client, followed by a multicast containing the following message:

```
{
  "command": "NEW_USER",
  "username": "fahmin",
  "secret": "mnmpp3ai91qb3gs"
}
```

This incurs a global update to user_info among all servers.

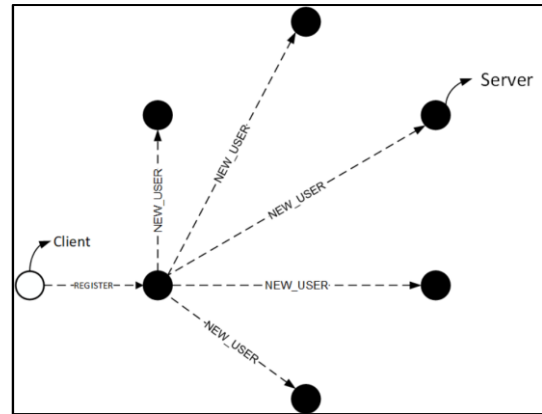


Figure 3.

iii. Concurrency concerns

Initially, we had thought to extend the global lock to ACTIVITY_MESSAGE operations between servers to ensure a fixed message order for all clients. For now, this is done by including "timestamp" command into the activity broadcast message among the servers. The client won't receive the timestamp though. This timestamp also helps to check the time frame between lost messages and abrupt connection closing. However, the project only specified a consistent ordering within a *single* user's sent messages, for all recipient clients. And since each client connection occupies its own thread, and activity messages are processed in FIFO order, no additional work was required to ensure a consistent ordering among messages from a single client.

4. Detection and handling of crashes

i. Failure Detection

Given that our servers are always listening for messages received from clients and servers, failure detection is quite straightforward. In the event of a crash, a TCP exception is produced at the corresponding port, informing the listening server *where* the crash took place. And since all servers are listening to all others at all times (by virtue of the flat 'mesh' structure), an exception will be universally produced and detected, ensuring a consistent global state throughout the system.

ii. Client Connection Closure

Whenever a client becomes inactive (whether it be from logging out or a detected crash/connection loss), the server will multicast the following message:

```
{
  "command": "LOGOUT_USERNAME",
  "username": "FAHMIN",
  "timestamp" : ...
}
```

In turn, every server will create a 'FAHMIN' key in their backlog hashmap, proceeding to add all activity messages received after the LOGOUT_USERNAME command.

If/when the client reconnects to the system, the connected server will search the backlog for messages. If found, messages are sent in FIFO order to the client, followed by the universal removal of the user's backlog entry across all servers in the system.

This ensures that a client who abruptly disconnects following a message being sent, but prior to receiving it, will receive the message upon logging back in.

iii. Server Connection Closure

In the event of a detected server failure, its connection will be closed by every server on the system. Furthermore, since the intermittent SERVER_ANNOUNCE messages have been informing the system which clients were logged in at the crashed server, every server will add those clients to their backlog, and be ready to deliver messages if/when they log back in.

iv. Rebalancing clients across servers

Our system continues to balance clients across the system such that no server will have two more connected clients than any other in the network. This is done at the time of receiving SERVER_ANNOUNCE messages. Server checks its load with other server's load, if the difference is more than 2, then the server randomly pick any of its client and send REDIRECT message to the client.

This holds for server authentication. Specifically, when a server is added to the system, a system wide rebalancing will occur to ensure an even (within 2-clients) distribution.

5. Possible improvements/concerns

i. Differentiate between a crash and intermittent connection loss

Unfortunately, our implementation doesn't differentiate between an intermittent loss of connection and a fundamental crash of a client/server; any sign of inactivity is treated the same way. This would be frustrating for an end-user, who would be required to log back in any time a minor disruption to their connection occurs.

A greater problem would be if, for instance, a server had an unreliable connection to other servers – persistently becoming inactive and dropping its clients. In this case, the cost of repeatedly re-authorizing, multicasting a NETWORK_STATUS update, and rebalancing the client list, only to have it quickly drop-out again, could become a significant detriment to the performance and availability of the service for clients.

This problem would be amplified as the system grows, since the cost of performing a NETWORK_STATUS, as well as the likelihood of (possibly) multiples servers having an unreliable connection, will grow.

ii. Possible global state inconsistency

We have also identified the possibility of a concurrency error when multiple servers are authenticated at once. Specifically, if two servers multicast a NETWORK_STATUS before having received the other, then two distinct server lists may coexist in the system (since each server overwrites their server list based on the most recent NETWORK_STATUS they have received). This violates system consensus on the global state. And crucially, this means a portion of the servers in the system would be unaware of some server's existence, leading to further inconsistency as the system continues.

This could be solved by implementing a global lock state on NETWORK_STATUS operations. However it is important to note this would impact the scalability of the protocol.

Group Meeting Minutes

I. Meeting on 18/05/18

Attendance: He Huang, Ahmed Fahmin, Mohammed Nafis Ul Islam, Alexander Thomson

Time: 3:00 pm to 4:00 pm

Concerning: Discussion of the first project, focusing on what we thought we did well and poorly.

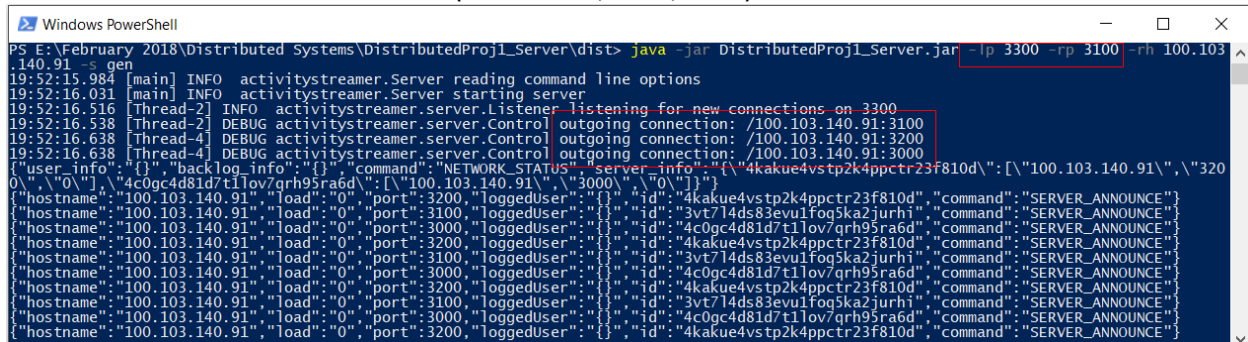
Moved onto discussing project 2 and group member roles. Given our success in project 1, we opted for a similar delineation of work in this project.

Appendix

Here, we are showing steps and illustration about how different cases work, according to the aims of the project.

Case 1: Allowing server to join at any time, even after some clients have registered.

Our server network is a full mesh. So, any server can join at any time and get the information shared by the previous servers. The creation of complete connection is shown of Figure: 1. A new server is being opened on port 3300, and it is creating outgoing connections with all other servers (Port: 3100,3200,3000)

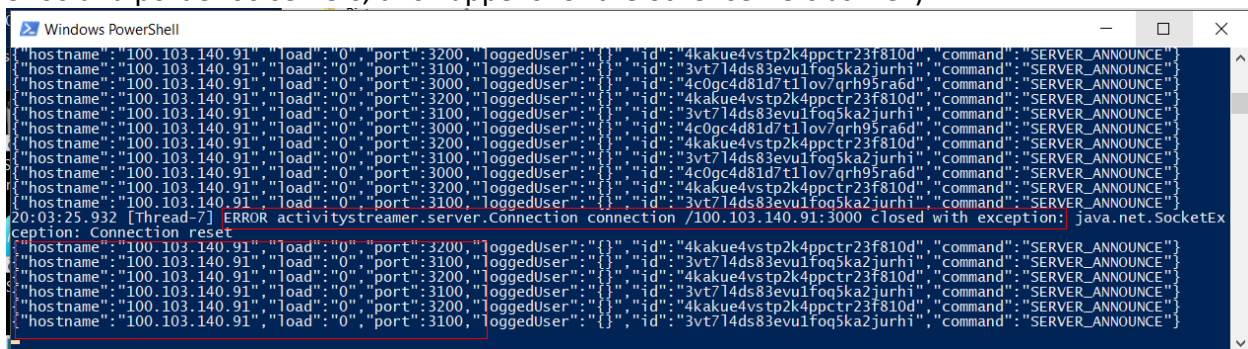


```
PS E:\February 2018\Distributed Systems\DistributedProj1_Server\dist> java -jar DistributedProj1_Server.jar -lp 3300 -rp 3100 -rh 100.103.140.91 -s gen
19:52:15.984 [main] INFO activitystreamer.Server reading command line options
19:52:16.031 [main] INFO activitystreamer.Server starting server
19:52:16.516 [Thread-2] INFO activitystreamer.server.Listener listening for new connections on 3300
19:52:16.538 [Thread-2] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3100
19:52:16.638 [Thread-4] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3200
19:52:16.638 [Thread-4] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3000
{
  "user_info": {},
  "backlog_info": {},
  "command": "NETWORK_STATUS",
  "server_info": {
    "id": "4kakue4vst2k4ppctr23f810d",
    "hostname": "100.103.140.91",
    "load": "0",
    "port": "3300",
    "loggedUser": {}
  }
}
[{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"}]
```

Figure 1 : Case 1

Case 2: Handling Server Crash/Failure.

Since, it is a full mesh topology, the server network does not fail, and it also does not need any network partitioning. Figure: 2 shows that even when the server on port 3000 crashes, the other servers relate to each other. (Server on port 3300 has connection with both port 3100 and port 3200 servers, this happens for the other servers as well)



```
20:03:25.932 [Thread-7] ERROR activitystreamer.server.Connection connection /100.103.140.91:3000 closed with exception: java.net.SocketException: Connection Reset
[{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3200", "loggedUser": {}, "id": "4kakue4vst2k4ppctr23f810d", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3100", "loggedUser": {}, "id": "3vt714ds83evu1foq5ka2jurhi", "command": "SERVER_ANNOUNCE"},
{"hostname": "100.103.140.91", "load": "0", "port": "3000", "loggedUser": {}, "id": "4c0gc4d81d7t1lov7qrh95ra6d", "command": "SERVER_ANNOUNCE"}]
```

Figure 2: Case 2

Case 3: Allowing clients to join and leave at any time, and maintaining that, a given username can only be registered once over the server network.

Clients can freely join and leave at any time. All exceptions are handled with consistency. To handle the registration consistency, instead of lock request, lock allowed and lock denied commands, we are using the server announce to share the information of the newly registered user to all other servers(including any new server that comes after that client registration.) Figure: 3a shows a new client registration into server with port 3000.

```
Windows PowerShell
PS E:\February 2018\Distributed Systems\DistributedProj1_Client\dist> java -jar DistributedProj1_Client.jar -rp 3000 -rh 100.103.140.91 -u Nafis
20:19:11.568 [main] INFO activitystreamer.Client reading command line options
20:19:11.599 [main] INFO activitystreamer.Client starting client
20:19:12.154 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"REGISTER_SUCCESS","info":"register success for Nafis"}
20:19:12.154 [Thread-1] INFO activitystreamer.client.ClientSkeleton Secret for the user Nafis is : 8sfcesvcjokl1vsr3agn36tmk
20:19:12.154 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"LOGIN_SUCCESS","info":"logged in as user Nafis"}
```

Figure 3a: Case 3a

Figure: 3b shows that this new user info is passed to other servers using the NEW_USER command. Upon receiving this command, the server adds this user to its registered user list. This happens for all the servers.

```
Windows PowerShell
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3000,"loggedUser":{"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3000,"loggedUser":{"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"secret":"8sfcesvcjokl1vsr3agn36tmk","command":"NEW_USER","username":"Nafis"}
{"hostname":"100.103.140.91","load":"1","port":3000,"newLoggeduser":"Nafis","id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"Nafis":"8sfcesvcjokl1vsr3agn36tmk"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"Nafis":"8sfcesvcjokl1vsr3agn36tmk"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
```

Figure 3b : Case 3b

When a new server (port 3300) comes in, it saves the new registered user info using the NETWORK_STATUS command. It is shown on Figure: 3c.

```
Windows PowerShell
PS E:\February 2018\Distributed Systems\DistributedProj1_Server\dist> java -jar DistributedProj1_Server.jar -lp 3300 -rp 3100 -rh 100.103.140.91 -s gen
20:28:33.236 [main] INFO activitystreamer.Server reading command line options
20:28:33.273 [main] INFO activitystreamer.Server starting server
20:28:33.791 [Thread-2] INFO activitystreamer.server.Listener listening for new connections on 3300
20:28:33.800 [Thread-2] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3100
20:28:33.911 [Thread-4] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3000
20:28:33.913 [Thread-4] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3200
{"user_info":{"Nafis":{"8sfcesvcjokl1vsr3agn36tmk"},"backlog_info":{"},"command":"NETWORK_STATUS","server_info":{"8ieg1c0ap0891oou3trcq91n55":{"100.103.140.91":{"3000"},"1"},"5ettuqhnnavusfe15f00umor3dh":{"100.103.140.91":{"3200"},"0"}}}}"}
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
```

Figure 3c: Case 3c

So, even if the server on port 3300 has come after that user registration, it will know about it, and won't allow to register any user with that same user name. It is shown of Figure: 3d.

```
Windows PowerShell
PS E:\February 2018\Distributed Systems\DistributedProj1_Client\dist> java -jar DistributedProj1_Client.jar -rp 3300 -rh 100.103.140.91 -u Nafis
20:30:56.460 [main] INFO activitystreamer.Client reading command line options
20:30:56.491 [main] INFO activitystreamer.Client starting client
20:30:57.024 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"REGISTER_FAILED","info":"Nafis is already registered with the system"}
PS E:\February 2018\Distributed Systems\DistributedProj1_Client\dist>
```

Figure 3d: Case 3d

Case 4: Handling client crashes and ensuring that an activity message sent by a client reaches all clients that are connected to the network at the time that the message was sent.

This is handled by maintaining a backlog to contain the lost message incase of abrupt failure. And, the disconnected user info is passed to the servers through the LOGOUT_USERNAME command. Figure: 4a shows a client disconnection and Figure: 4b shows the information passing regarding this disconnection to other servers.

```

Windows PowerShell
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
20:33:32.063 [Thread-7] ERROR activitystreamer.server.Connection connection /100.103.140.91:62974 closed with exception: java.net.SocketException: Connection reset
20:33:32.064 [Thread-7] INFO activitystreamer.server.Connection Client : Nafis connection closed.
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3300,"loggedUser":{"},"id":"kobdm89mf7031efim4puju2ssf","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3300,"loggedUser":{"},"id":"kobdm89mf7031efim4puju2ssf","command":"SERVER_ANNOUNCE"}

```

Figure 4a: Case 4a

```

Select Windows PowerShell
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\Nafis\\":"8sfcesvcjokl1vsr3agn36tmk\\"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
{"command":"LOGOUT_USERNAME","username":"Nafis","timestamp":"2018-05-26 20:33:32.062"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\Nafis\\":"8sfcesvcjokl1vsr3agn36tmk\\"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3000,"loggedUser":{"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3100,"loggedUser":{"},"id":"cr5v1h31a161tvgr41kdhsua","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3200,"loggedUser":{"},"id":"5ettuqhnavusfe15f00umor3dh","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"0","port":3000,"loggedUser":{"},"id":"8ieg1c0ap0891oou3trcq91n55","command":"SERVER_ANNOUNCE"}

```

Figure 4b: Case 4b

Now, we have used a delay of 10 second in our code, to test the successful receiving of the lost message in the crashed client/user, after it logs back in. And to check that, we have also passed time stamp with the activity broadcast message, so that, we can check the time frame between connection lost and message sent. The different times are shown on Figure: 4c. It can be seen that the message was sent before the connection was closed.

```

{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}
20:47:48.567 [Thread-6] ERROR activitystreamer.server.Connection Client : nafis connection closed.
{"command":"ACTIVITY_BROADCAST","Testing":"Message sent before crashing","timestamp":"2018-05-26 20:47:48.54"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"1","port":3000,"loggedUser":{"\\ZanMalik\\":"55triilhj92ghf3e4baolicujo\\"},"id":"bmrdrv3494c74ocvklpqkdj0vhj","command":"SERVER_ANNOUNCE"}

```

Figure 4c: Case 4c

Now, Figure: 4d shows that when that user logs back again, it readily receives that lost message after logging in.

```

20:50:35.424 [main] INFO activitystreamer.Client reading command line options
20:50:35.456 [main] INFO activitystreamer.Client starting client
20:50:35.487 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"LOGIN_SUCCESS","info":"logged in as user nafis"}
20:50:35.502 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"ACTIVITY_BROADCAST","Testing":"Message sent before crashing"}

```

Figure 4d: Case 4d

Here, the previous (project 1) load balancing mechanism works fine when a new user tries to log in and is redirected to the server, so that the loads are balanced. But the previous assumption was that, the servers start first, and never crash and then comes the clients. But, in this project, servers can enter the network at any time. And when it does, the load balancing is handled with the new server too.

[illegible][illegible]

Now, we connect a new server to the network using port 3300. This server will have 0 load, and will create the opportunity to balance the loads of the previous servers(port 3000 and port 3100). This is shown on Figure: 5c. The initial loads for them were 5 and then balanced to 3 loads later, as shown. So, the new server will have 4 clients redirected to it for balancing. It is shown on Figure: 5e.

```

21:12:16.204 [Thread-2] INFO activitystreamer.server.Listener listening for new connections on 3300
21:12:16.204 [Thread-2] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3100
21:12:16.319 [Thread-4] DEBUG activitystreamer.server.Control outgoing connection: /100.103.140.91:3000
{"user_info":{"fis":"juq4grsrhlhlnio94dp5pmmjit","na":"te2lvto17see8jggihniq8kqn","ZanMalik":"f09b978g0lh9hur0li74jsp3lk","zan":"lqdav0
","hostname":"100.103.140.91","load":"5","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","ZanMalik":"f09b978g0lh9hur0li74jsp3lk"},"
{"hostname":"100.103.140.91","load":"5","port":3100,"loggedUser":{"na":"te2lvto17see8jggihniq8kqn","zan":"lqdav0i56716q2m4ls0i2ur0pk","nafis\
21:12:21.321 [Thread-2] DEBUG activitystreamer.server.Control incoming connection: /100.103.140.91:63245 User: anonymous Secret: null
21:12:21.321 [Thread-2] DEBUG activitystreamer.server.Control incoming connection: /100.103.140.91:63246 User: anonymous Secret: null
{"secret":"lqdav0i56716q2m4ls0i2ur0pk","command":"LOGIN","username":"zan"}
{"secret":"f09b978g0lh9hur0li74jsp3lk","command":"LOGIN","username":"ZanMalik"}
21:12:21.336 [Thread-2] DEBUG activitystreamer.server.Control incoming connection: /100.103.140.91:63247 User: anonymous Secret: null
21:12:21.336 [Thread-2] DEBUG activitystreamer.server.Control incoming connection: /100.103.140.91:63248 User: anonymous Secret: null
{"secret":"bs4p686ib8jv75eueikcpvi4","command":"LOGIN","username":"asifl"}
{"secret":"td95dhghqvtj0nb7lhmvms0bvcm","command":"LOGIN","username":"hulu"}
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","naf
{"hostname":"100.103.140.91","load":"3","port":3100,"loggedUser":{"na":"te2lvto17see8jggihniq8kqn","nafis":"f6bbdkham63imal7v4q74i0o3d","nusl
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","naf
{"hostname":"100.103.140.91","load":"3","port":3100,"loggedUser":{"na":"te2lvto17see8jggihniq8kqn","nafis":"f6bbdkham63imal7v4q74i0o3d","nusl

```

Figure 5c: Case 5c

A snapshot from a redirected client, when the new server(port 3300) comes in is given on Figure: 5d.

```

21:08:52.372 [main] INFO activitystreamer.Client reading command line options
21:08:52.394 [main] INFO activitystreamer.Client starting client
21:08:52.927 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"REGISTER_SUCCESS","info":"register success for ZanMalik"}
21:08:52.927 [Thread-1] INFO activitystreamer.client.ClientSkeleton Secret for the user ZanMalik is : f09b978g0lh9hur0li74jsp3lk
21:08:52.927 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"LOGIN_SUCCESS","info":"logged in as user ZanMalik"}
21:12:21.321 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"hostname":"100.103.140.91","port":"3300","command":"REDIRECT"}
21:12:21.336 [Thread-1] INFO activitystreamer.client.ClientSkeleton {"command":"LOGIN_SUCCESS","info":"logged in as user ZanMalik"}

```

Figure 5d: Case 5d

Finally, Figure: 5e shows the new loads of the servers (port 3300(new server) and port 3000) after balancing. It gradually rises from 0 to 4, and then becomes stable, by balancing the loads as much as possible. The loads of port 3100 server is already shown on Figure: 5c.

```

{"hostname":"100.103.140.91","load":"5","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","ZanMalik":"f09b978g0lh9hur0li74jsp3lk","naasl\
21:12:21.321 [Thread-11] INFO activitystreamer.server.Connection closing connection /100.103.140.91:63228
21:12:21.321 [Thread-6] DEBUG activitystreamer.server.Connection connection closed to /100.103.140.91:63228
{"hostname":"100.103.140.91","load":"0","port":3300,"loggedUser":{"id":"lcl6c0vs5mdnk7ejiek0embrfo","command":"SERVER_ANNOUNCE"}
21:12:21.321 [Thread-11] INFO activitystreamer.server.Connection closing connection /100.103.140.91:63231
21:12:21.336 [Thread-7] DEBUG activitystreamer.server.Connection connection closed to /100.103.140.91:63231
{"hostname":"100.103.140.91","load":"1","port":3300,"newLoggedUser":"zan","id":"lcl6c0vs5mdnk7ejiek0embrfo","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"2","port":3300,"newLoggedUser":"ZanMalik","id":"lcl6c0vs5mdnk7ejiek0embrfo","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"3","port":3300,"newLoggedUser":"asifl","id":"lcl6c0vs5mdnk7ejiek0embrfo","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"4","port":3300,"newLoggedUser":"hulu","id":"lcl6c0vs5mdnk7ejiek0embrfo","command":"SERVER_ANNOUNCE"}
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","nafisl\
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","nafisl\
{"hostname":"100.103.140.91","load":"4","port":3300,"loggedUser":{"ZanMalik":"f09b978g0lh9hur0li74jsp3lk","zan":"lqdav0i56716q2m4ls0i2ur0pk","hulu\
{"hostname":"100.103.140.91","load":"4","port":3300,"loggedUser":{"ZanMalik":"f09b978g0lh9hur0li74jsp3lk","zan":"lqdav0i56716q2m4ls0i2ur0pk","hulu\
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","nafisl\
{"hostname":"100.103.140.91","load":"4","port":3300,"loggedUser":{"ZanMalik":"f09b978g0lh9hur0li74jsp3lk","zan":"lqdav0i56716q2m4ls0i2ur0pk","hulu\
{"hostname":"100.103.140.91","load":"3","port":3000,"loggedUser":{"fis":"juq4grsrhlhlnio94dp5pmmjit","naasl":"amlt4sdi9kjvuir98450h3uil","nafisl\
{"hostname":"100.103.140.91","load":"4","port":3300,"loggedUser":{"ZanMalik":"f09b978g0lh9hur0li74jsp3lk","zan":"lqdav0i56716q2m4ls0i2ur0pk","hulu\

```

Figure 5e: Case 5e