## Aims of the Project 2

1. **Allows servers to join the network at any time**

As the similar principle in Project 1, Server joins the distributed system by sending *AUTHENTICATE* protocol to its target server. In addition, for this project, other fields of attributes are added in this type of protocol—requesting server’s hostname and port number, which are attributes that helpful to eliminate circulate connections afterwards.

1. **Allows clients to join and leave the network at any time**

In Project 2, similarly the protocols including *REGISTER*, *LOGIN* and *LOGOUT* are taken into usage, to handle the standard user logging in and out like they did in Project 1.

In addition, mechanism as testing the connections to clients are implemented as well, which ensures the user’s abrupt quitting from the server (without sending *LOGOUT*, due to error or network failure) would not invoke any failure or exception to the whole system, followed by safe removal of connections to clients.

1. **Maintains that a given username can only be registered once over the server network**

The protocol *SERVER*\_*ANNOUNCE* is modified—it takes the whole clients list of the sender server, which guarantees that the servers in same system could maintain a same clients list that includes all registered clients. Hence, once a new client registering at a server in the system, the server would know whether this client name exists or not. Rejection would be sent immediately once the registering client’s name has already existed in the local clients list.

Also, the protocols as *LOCK*\_*REQUEST*, *LOCK*\_*ALLOWED* and *LOCK*\_*DENIED* are reserved as well but they function in a different way—only used when synchronizing latency of *SERVER\_ANNOUNCE* occurred, in which case the system would run as the way in Project 1, to guarantee the overall availability.

1. **Guarantees 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**

Under the normal system status, all activity message would be delivered to all the clients existed at a time in the system. Hence, in this project, this availability issue could only happens when something goes wrong, including network isolation, server failure etc. The failover mechanism would be discussed in Failover Section afterwards.

On solving the issues above, the availability of activity sending could be achievable. One possible access to this is to save all clients messages during the fail time as sound as possible. Then after the restoration of the system, the server would send previous saved messages to the remaining clients that still existed in the system.

To achieve these, following mechanisms in each server are implemented:

**Activity Messages Pool:**

Which contains all the activity messages the server received from the network.

By maintaining this kind of Messages Pool, each server is capable to cache the Activity message sent by clients over the network, which would save the Messages even if the unexpectable failures occur in the system. Then after the restoration of the system, another resending mechanism, which would be discussed later, resents the delayed messages back to the target clients.

Along with the infrastructures above, another protocol *SERVER\_ACTIVITY\_BROADCAST* is created, which functions as most of the previous *ACTIVITY\_BROADCAST* protocol but only transferred between servers, which additionally records the SENT and RECV number for each activity from clients. Once a server receiving this type of protocol, it would extract necessary information from it and push it into the Activity Messages Pool.

Then another thread in resending mechanism, which would take charge of taking activity messages from Activity Message Pool, is created, and sending *ACTIVITY\_BROADCAST* message to each clients that connected to current server, on referring clients information in the Message Sending and Receiving List.

1. **Guarantees that all activity messages sent by a client are delivered in the same order at each receiving client**

The solution to this issue could be regarded as continuous parts of previous issue, which, in the same time, shares the most infrastructures above. Besides, additional issues should consider as following:

Record the behaviours of clients that in local, including:

1. Index of message for each client that sends out to other clients.
2. Index of message for each client that receives message from the server.

By consulting SENT and RECV index for each client before sending, the server could acknowledge which activity in the Activity Message Pool should be sent to which client that connect to it. This process could be illustrated more clearly in following graph:

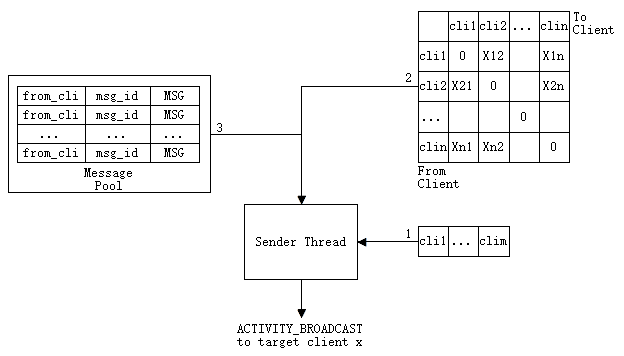


Figure 1: Resending Mechanism

To broadcast Activity Messages to clients:

1. The Sender Thread iterates each connected client connections on server (cli 1 … cli m).
2. For each client connection (clia), check the 2-D table at the column of (clia).
3. For each pair (clib, Xba) in column of (clia), check in the Message pool to make sure the existence of item: [from\_clib | msg\_id = = Xba | MSG], 2 branches as following:
   * 1. True:

Form ACTIVITY\_BROADCAST based on MSG, then send to client clia

In 2-D table: Xba += 1, return to step 3.

* + 1. False:

Move to next client in local client connections list (clia+1), return to step 2.

\* In this Project, every single user is allowed to login only once at a time. Hence, the diagonal of the 2-D table would be always 0—one client could not receive Message from itself.

1. **Ensures that clients are evenly distributed over the servers as much as possible**

In project 1, the server could automatically distribute the new connected clients to the server that has relatively lower load, which could be regarded as half-way done of this project. Moreover, in this Project, even the previously connected clients would be redirected to other servers, according to the current changes to the servers’ loads, including new connections from incoming clients, network isolations and server failures etc.

To achieve this, the place that invokes REDIRECTION is changed to the regular doActivity()—the function would be invoked between every certain time interval. By modified like this, the server could check the latest status over the whole network—especially the latest servers’ client loads, then make necessary REDIRECTION of local connected clients immediately.

1. **\*Single user login control**

The mechanism for single client login control is implemented as: once the same client login elsewhere, the same client that originally logged in, would be forced to offline—by receiving LOGIN\_FAILED message.

Here a new protocol is added—CLIENT\_LOGIN\_SUCCESS, which broadcasts a client’s success login information between servers, along with the client’s name and secret. Once a server received this type of message, it would check its client connection list that it whether contains the CLIENT\_LOGIN\_SUCCESS assigned client or not. If existed, current server would send LOGIN\_FAILED message to the original client and abort the connection to it, which achieves the goal of single client control over the whole network.

## Messages & Protocols

All new-added protocols are mentions during previous sections.

## Failover Handling

In terms of abnormal situations, the problem would happen under the following two scenarios:

1. **Network Failure**

Once this type of failure happened, **it is for sure** that the system would be divided into several network isolations, depicted as following graph:

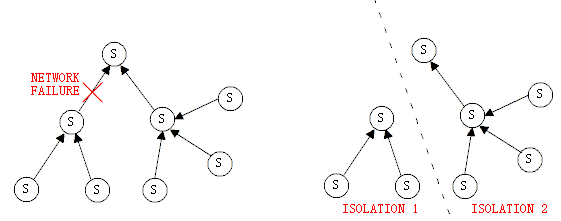


Figure 2: Network Failure towards Isolation

The solution to this problem is based on following rules:

* The servers should detect the failures in the system. To be specific, some originally accessible servers turn out to be inaccessible at a time.
* The servers should know which servers still accessible—within the same network isolation, which are not—failed or within other network isolations.
* Only the ROOT of each network isolation can invoke reconnection to other servers in other isolations.
* Once the non-root server of network isolation received reconnection request, it should redirect the requesting server to its father server.
* The architecture after reconnections should remain as a tree—circle connections are not allowed (e.g. S1🡪S2🡪S3🡪S1)

To implement which, other protocols are introduced:

*SERVER\_RECONNECT: {*

*REQUEST\_SERVER\_HOSTNAME*

*REQUEST\_SERVER\_PORTNUM*

*}*

*SERVER\_REDIRECT: {*

*REDIRECT\_SERVER\_HOSTNAME*

*REDIRECT\_SERVER\_PORTNUM*

*}*

Protocol *SERVER\_RECONNECT* is used to request reconnection to other servers located in other isolations, with provided requesting server hostname and port number. After successfully sending this message, the requesting server adds the connection into local server connections list.

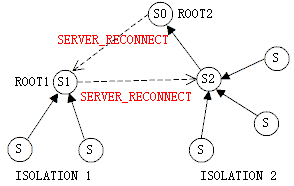


Figure 3: ROOT Servers send SERVER\_RECONNECT to opposite isolations

On the opposite, the target server once received this *SERVER\_RECONNECT* message, it would check its status in network isolation for decisions:

**ACCEPT** this request by adding this connection to local server connections list. To accept the SERVER\_RECONNECT request, the server replies nothing back. Only the ROOT server (S1) of a network isolation could do this.

**SERVER\_REDIRECT** the requesting server to other server in this isolation (S2 redirects S1 to S0), by using the protocol SERVER\_REDIRECT that contains the redirect-target’s hostname and port number. Only the non-ROOT server (S2) in a network isolation could do this.

In this example, S1 and S0 are the two ROOT servers in their network isolations. Then supposing: S1 sends SERVER\_RECONNECT to S2, and S0 sends SERVER\_RECONNECT to S1. As a root server, S1 would ACCEPT S0’s request, whereas server S2 would redirect S1 to its father S0.

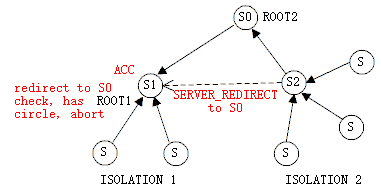


Figure 4: SERVER\_REDIRECT and Circle Elimination

By this implementation, there would be no existence of long circle connections across more than 2 servers—circle is only able to present between 2 nodes (S1🡨🡪S0 in this example). To eliminate this circle, S1 should check its local connections to other servers, and would find it has already ACCEPT S0 reconnection, just abort its attempt for connecting to S0 again.

Finally, the reconnected system would be presented as following. The ROOT server is not remained as S0 and changed to S1, which is based on the sending sequence of SERVER\_RECONNECTs between network isolations.

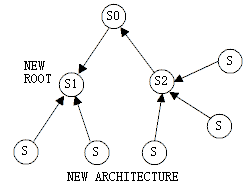
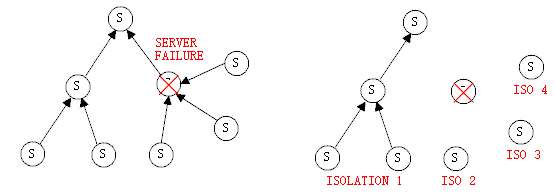


Figure 5: Reconnected System after Failover

1. **Server Failure**

Server failure turns out to be more problematic—it would generate isolations of servers, which could process the similar mechanism as network isolation. As a result, this type of server failure would result in reconstruction of the original architecture.



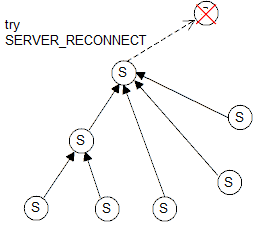


Figure 6: Server Failure with Network Isolation

The difference between this case and the previous Network Failure lies in the number of Server changes and the new ROOT server would try reconnect to the failed server, which enables the situation that: once the failed server back to online, with AUTHENTICATE or not, to its original connected server, the system would connect together as a single network again.

**Elsewise,** Server Failure would not create any isolation, considering following situation:

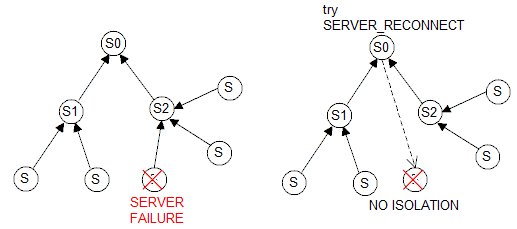
****

Figure 7: Server Failure without Network Isolation

One of the leaf server fails, all other servers notice this failure after a certain time span. Then, only the ROOT server S0 (without father server) invokes SERVER\_RECONNECT to failed server. This type of server failure does no big structure change to the system.

## High Availability

Since high availability is of the first priority in this system, user operations would never be banned regardless what issues are facing among servers. Servers will always try to maintain in or restore to a normal status for providing stable and continuous service. Proper communications and routine checks will be hold by every server. Thus any potential or actual problems could be found in time in order to minimize its impact on the whole system.

Given any situations, multiple clients can register, login or send messages at the same time on an arbitrary server. If any servers crash happens or system broken into partitions, new user and former user could still register and login to any of the partitions correspondingly and for those clients already logged onto the system will not be influenced. While regarding to activity broadcast, clients are allowed to send activities continuously no matter what, and those clients on the same server can always receive activities from each other in order without any delay, which is the same to clients on different servers normally.

## Eventual Consistency

All servers are diligently sending its state on the system and seeking for up to date with others. In fact, even if only parts of the system problems are solved, those involved servers will start synchronizing their state immediately. So once the whole system restores to a normal status, eventual consistency could be fulfilled in short.

Many external problems may cause servers could not communicate with each other or can only receive commands with delay, servers could still make sure all things be done as usual except been put off. Because once when problems been solved, differences between registered clients list on each server would be realized, synchronization will be done after several rounds of SERVER\_ANNOUNCE. In addition, because all client activities are numbered and stored when it first arrives a server and also the received sequence of activities of each client are recorded, servers will resume delivering latest broadcast messages, start from the sequence of last received message, to each client.