

```

*****
*                COP5615 Project 3.1                *
*                Reza Mahjourian                    *
*                UFID: 9254-5986                    *
*                Course account: rml                 *
*****

```

## Protocol Design

### 1. Introduction

In this document we explain our suggested extensions both to the protocol and to the software infrastructure of the components in previous projects to fulfill the requirements of project 3. We present our server discovery algorithm and protocol. We also explain the necessary changes that need to be made in the agent and the client to achieve the required dynamism and fault tolerance. At the end of the document we will list new message formats and new commands that should be added to the previous implementations. *We do not repeat the commands and message formats that are already specified in Project 1 and Project 2.*

### 2. Server Graph

The network of servers can be represented as a connected graph. The nodes in this graph represent the servers and an edge between two nodes denotes existence of a connection between those two servers. Theoretically the degree of nodes in this graph can be anywhere from 1 to  $n-1$ . However, for efficiency purposes we prefer not to have a completely connected graph where the degree of all nodes is  $n-1$ . This is because requiring that all servers be communicating with all other servers can reduce the system's scalability. On the other hand, we need to guarantee a reliable and dependable system. In particular we don't expect the failure of any node to break the chain of connections between the nodes, which can create separate islands of servers. So we need to employ some fault tolerance mechanism to overcome this problem.

Our solution is that we let the servers exchange information about their neighbors. Every server will update its neighbors about all its neighbors as soon as it detects any change in its active connections. In the advent of a server failure, its adjacent nodes already know about each other and can connect to each other to maintain the connectivity of the graph.

In addition, to increase the reliability of our network, we can set a minimum required degree of connectivity for every node in the network. For example if we set a minimum degree of two, then every single node must have at least two neighbors. If a server dies, then its

neighbors will try to establish new connections to bring the degree of their corresponding nodes back to two. The rationale behind this decision is that if some servers are connected to only one neighbor, then it is possible that a sudden failure in two adjacent servers can break the connectivity chain. Determining an optimum value for this lower limit depends on the expected probability of failures.

### 3. Detection of Failure

All servers periodically send “ping” messages to all their neighbors. Also all agents periodically send these ping messages to the server to which they are connected. The format of this ping packet is quite simple and does not contain any parameters. The receiver of a ping message should respond with a “pong” message. There is a failure detection thread running on all servers and agents which is in charge of both sending and receiving the “ping” and “pong” messages. As soon as a “ping” or “pong” message is received, this thread updates a `last_seen_timestamp` for the corresponding node in its internal database. The failure detection thread constantly monitors this table and when it detects that it has not received any message from a node for a specified period of time, it assumes that the node is dead and tags it as dead in its internal database.

We will later see that there is a handshake that is supposed to happen between servers when they want to connect to each other. However if a server receives a ping or pong message from a dead neighbor, it should reactivate its record without requesting a new handshake. The rationale behind this decision is increasing the robustness of the system in situations where long network delays or short communication interruptions can happen.

### 4. Inter-server communications thread

We need to run a special thread on all servers which is in charge of handling the inter-server communications required for doing connection handshakes, performing distributed searches, communicating list of neighbors and responding to ping messages. We will explain the details of each of these operations in the following sections. It is also possible to design separate threads for each of these tasks.

### 5. Server Discovery

For server discovery we suggest a protocol which resembles the DHCP protocol. When a server comes online, it sends a multicast UDP packet to a pre-specified port to announce that it is up and is interested in connection offers from other servers in the network. Then one or more servers are going to respond back with connection offers, which are sent in unicast UDP packets. Depending on the minimum connectivity degree set on servers, the new server may

accept one or more of these offers by sending a connect-request message in a unicast UDP packet. The receiver of the connect-request message is going to respond with a connect-acknowledged message. It's only after exchanging the connect-request and connect-acknowledged messages that the connection becomes official and the two servers mark each other as neighbors. The multicast packets is sent to the pre-specified port for server discovery messages. But all the unicast messages are sent to the inter-server communication port of each server.

The diagram below outlines the sequence of messages:

Communication	Message Content	Type
New Server -----> All Servers	Announce	Multicast UDP
New Server <----- Existing Server	Connection Offer	Unicast UDP
New Server -----> Existing Server	Connect Request	Unicast UDP
New Server <----- Existing Server	Connect Acknowledged	Unicast UDP

As we mentioned earlier, we prefer not to have a fully connected graph. One possible solution to achieve this quality is setting a tentative maximum connectivity degree in servers. In that case, the two bounds of minimum connectivity degree and maximum connectivity degree determine the probability that a server is going to send a connection offer message in response to receiving an announce message. We suggest the following routine to decide the probability of sending a connection offer message:

```

if current degree <= minimum degree
then
    P = 100%
else
    allowance = max (maximum degree - current degree, 1)
    maxlevel = maximum degree - minimum degree
    P = allowance / maxlevel
r = get random number between 0 and 100; (0 <= r < 100)
if r < P
then
    Send a connection offer message

```

We prefer to have all nodes with minimum connectivity degree respond with connection offers. This is because we do not expect to see any node with a lower degree under normal condition and we want to make sure that the new server receives a connection offer under all

circumstances if there are other servers working in the network. The probability of responding back approaches zero as the server approaches the maximum degree. However the probability is never set to zero to make sure that we never get isolated subsets of servers which are connected only to each other with maximum connectivity degree.

**Solution B:** The initial proposal recommends making some servers unreachable from some subsets. We can adopt that solution as well, however we prefer to make it possible for all servers to connect to each other to minimize the probability of creating islands.

## 6. Search Protocol

The search is implemented as follows. When an agent sends a search request to the server, the server creates a search message which includes the following parts:

1. Alias to be searched
2. Unique id of the query
3. List of servers who have seen the request: initialized to server's own name/ip+port
4. List of matches found: initialized to empty

The server then sends this message to all of its neighbors. However it sends the message to the neighbors one at a time and waits until it receives their reply (or a timeout) before moving to the next neighbor. This is equivalent to doing a Depth-First-Search traversal of the server graph. The rationale behind this decision is that we want to make sure that no server is going to receive the search request more than once. Considering that we may have cycles in our server graph, if we send the message to all neighbors at each level, then the receivers can not tell which other servers have already processed this message and may try to pass the request to some server which has already sent a reply.

When a server receives this search request, it adds its information to the list of servers who have seen the request. It does that by modifying the server list in the header of the message. It then searches its internal alias database to see if it can find any matches and if it finds any, it adds them to the body of the message. It then uses the same algorithm to pass the request to all of its neighbors, provided their information is not already present in the header of the message. When a server receives all the replies from the neighbor servers it has contacted, it merges all the matches and all server lists in a search results messages which it sends back to the original server.

Each search request or reply is accompanied by a unique query id which is originally generate by the original server which initiated the search. These query ids are used to differentiate different search query and responses that might be going around the network simultaneously. Each server keeps two lists of searches. The first list reflects the searches that have been

originated at the local server. The server is awaiting responses from its neighbors on these searches so that it can respond back to the agent. This list also maintains the information on the agent that originally requested the query. The second list is about the searches that have been passed over to this server and the server is awaiting responses from its neighboring server so that it can respond back to the server which passed this query to this server. In this case, the server keeps the contact information of the parent node who passed the search down to this server.

It is debatable that it might be more efficient if we just ask each server to respond back directly to the original server which initiated the search request. However the original project proposal recommends the transitive approach. In addition, if the original server were to receive results directly from all other servers it could not tell when it has received all replies.

## 7. Handling Name Collisions

Since in this version of the application we are introducing a new agent-to-server message which signifies the request of an agent to connect to the server, we can handle alias name collisions at the time that the server receives this request. If the server detects an alias collision in its internal alias database, it's going to let the client know using a special reply. Otherwise, it sends an acknowledge message back to the client to let it know that the connection has been successfully established.

We also need to make sure that neighbor servers do not share the same name. It is very costly, if not impossible, to establish uniqueness of server names in the whole network. Therefore we only require that neighbors do not have name collisions. In the protocol messages everywhere we are using ip+port as the unique identifier for servers. If a server receives an announce message or a connect request message from another server with the same name, it does not reply/accept the connection.

When a server crashes, its clients will try to join to one of the neighbor servers. But the clients are going to use a different type of connect message, called tempconnect. In the tempconnect message the client also sends the name of the original server to which it was connected. The main difference between a tempconnect and a connect is that if a server receives a tempconnect request, it is supposed to accept the client even when it detects that the new client's alias is already in use by some other client connected to it. To resolve the name collision in this case, the server prefixes the new client's alias with the name of the original server to which the client was connected. A special character, like dot (.) is going to separate the server's name from the alias name. (Thus we should enforce the rule that server names can not contain this character.) The client itself does not need to change its local alias.

Servers need to be aware of this condition when performing distributed searches. If a server

receives a search request for an alias, it must try to match the query against the original alias of the temporary clients, and not the prefixed name.

## 8. New Client Commands

Since the task of switching back and forth between different servers is carried out automatically by the agent, we don't need to have a new command for the chat client for this purpose. We can just show informative messages on the agent console whenever we move to a new server. However if we also want to let the end user manually instruct the agent to connect to a different chat server after a connection has been established, we can implement the following command syntax:

**connect** <server ip> <server port>

And we need a command for performing searches:

**search** <alias>

## 9. New Message Formats for Server-to-Server communications

In this section we give the details for the format of new types of messages that are exchanged between the servers in the system. The messages that are already defined in Project 1 and Project 2 are not repeated here. We use this convention that if a word appears in italicized font, it means that the actual word is part of the message. It's while words and phrases enclosed in <...> are placeholders.

### 9.1. Server discovery and handshake messages

Server Discovery messages are received by the discovery management thread. Please refer to the server discovery section for an explanation of these messages. The port numbers here are the port number on which the inter-server communication thread is running.

Announce message:

**announce** <own ip address> <own port number> <own name>

Connection offer message:

**offer** <own ip address> <own port number> <own name>

Connect request message:

**conn** <own ip address> <own port number> <own name>

connect acknowledged message:

**connack** <own ip address> <own port number> <own name>

## 9.2. Search request and results messages

When sending a query message, each server sets its own ip address and port number in the fourth and fifth fields. Fields that begin with # contain information on servers that have already processed this particular search query. Fields that begin with \$ contain information about the matches that have been found so far. We are also including the actual matched alias along with each result, because in the future we might be interested to perform more complex searches, like regex-based searches and in that case it is important to get the alias names for each of the matches. Here is the complete format of the query message:

**query** <unique id> <search alias> <own ip address> <own search thread port>  
 #<ip1>#<port1> #<ip2>#<port2> \$<alias1>\$<ip1>\$<port1>  
 \$<alias2>\$<ip2>\$<port2>

The format of the search results message is quite similar to the search query message. The only difference is the initial op code in the message.

**results** <unique id> <search alias> <receiver ip address> <receiver search thread port> #<ip1>#<port1> #<ip2>#<port2> \$<alias1>\$<ip1>\$<port1>

## 9.3. Failure detection messages

These message are quite simple.

Ping message:

**ping**

Pong message:

**pong**

## 9.4. Neighbor info messages

We have four types of message in this category.

Neighbor list message is sent by servers to let their neighbors know about all other neighbors they are connected to. This message is sent as soon as the server detects a change in the list its neighbors.

***neighborlist*** <own name> <own ip address> <own port> #<neighbor ip1>#<neighbor port1> #<neighbor ip2>#<neighbor port2>

The next type of message is called clientlist which is also periodically sent to neighbors to let them know about the agents that are currently connected to this server.

***clientlist*** <own name> <own ip address> <own port> #<client ip1>#<client port1>#<client alias1> #<client ip2>#<client port2>#<client alias2>

The third message type is used by a server to query its neighbor about the clients that it used to have. This message is sent when it recovers from a failure to find its former clients.

***clientsquery*** <own name> <own ip address> <own port>

After receiving this message, the neighbor server responds with a clientsresults message which is quite similar to the clientslist message:

***clientsresults*** <own name> <own ip address> <own port> #<client ip1>#<client port1>#<client alias1> #<client ip2>#<client port2>#<client alias2>

## 10. New Message Formats for Agent-to-Server communications

In this section we give the details for the format of new types of messages that are exchanged between the servers and agents in the system. The messages that are already defined in Project 1 and Project 2 are not repeated here. We use this convention that if a word appears in italicized font, it means that the actual word is part of the message. It's while words and phrases enclosed in <...> are placeholders.

### 10.1. Establishing Connection

When an agent first starts up, it sends a connect request to a chat server. The contact information for the chat server is given on the command line. Here is the format of the connect message. The port number is the port number of the server response handler thread running on the agent.

Agent to Server: **connect** <alias> <ip address> <port>

Also when an agent wants to disconnect from a server, it sends the following command:

Agent to Server: **disconnect** <alias> <ip address> <port>

We also need a special connect message for the case when a client detects that its server is no longer available and decides to temporarily connect to one of the neighbor servers. We differentiate this type of connect request from the normal one, because in this case the neighbor server is supposed to accept the connection even when there is an alias conflict.

Agent to Server: **tempconnect** <alias> <ip address> <port> <original server name>

If there is not any problem with a client's connection, for example there is no alias conflict with a connect message, then the server responds with a acknowledge message:

Server to Agent: **connack**

If there is an alias conflict, the server responds with a conflict message, which informs the agent that the connection was refused due to an alias conflict.

Server to Agent: **connack aliasconflict**

## 10.2. Search requests and responses

When the agent receives a search command from the client, it sends a search query message to the server to which it is connected at that time. Here is the format of the search query message:

Agent to Server: **query** <search alias> <ip address> <port> <alias>

The first alias is the alias in which the user is interested. The last alias is the agent's own alias. The server performs a distributed search as described in the previous sections and when it gathers all the results, sends the data back to the client in a search results message, which has the following format:

Server to Agent: **results** <search alias> \$<alias1>\$<ip1>\$<port1>

### 10.3. Failure detection messages

The ping message is sent by the client periodically to determine whether the server is still alive.

Ping message:

Agent to Server: *ping*

Pong message:

Server to Agent: *pong*

### 10.4. Connection restoration message

When a server recovers from a failure, it tries to call back its former clients. This is done using the special clientrecall message. Here is the format of the clientrecall message:

Server to Agent: *recall* <ip address> <port>

The IP address and port information belongs to the server. Upon receiving this message the client is expected to terminate its current connection to the current server by sending a disconnect message and then connect to the old server using a connect message.