Table of Contents

1. About this Guide ..................................................................................................................... 3
2. Scope of this document .......................................................................................................... 4
3. System Overview .................................................................................................................... 5
4. System Components ............................................................................................................... 6
5. The Chat Server ..................................................................................................................... 7
6. The User Agent ..................................................................................................................... 14
7. The End User ........................................................................................................................ 18
1. ABOUT THIS GUIDE

This document provides the complete and consistent protocol specification for a fault tolerant chat application. The chat application consists of three modules, namely: Chat Servers, User Agents and End Users. The goal of the system is to provide a fault tolerant distributed application.
2. SCOPE OF THIS DOCUMENT

The scope of this document is to specify the following:

⇒ Protocol for communication between Chat Servers and User Nodes
⇒ Message and command formats
⇒ Message and command sequencing
⇒ Design constraints and assumptions
3. **SYSTEM OVERVIEW**

The following diagram shows the basic overview of the system:

Circular Nodes: Chat Servers

Square Nodes: User Agents and End Users

The following important points must be noted:

1. Each User Node consists of a User Agent and a End User.
2. Users participate in Chat Rooms within single Chat Servers.
3. Users can have peer-to-peer communication with User Nodes in other Chat Servers.
4. Chat Servers can communicate with each other. The communication includes:
   a. Discovering neighboring Chat Servers,
   b. Sending/requesting users list,
   c. Searching for specific users.
4. **SYSTEM COMPONENTS**

The system consists of three modules:

⇒ The Chat Server a.k.a. the server,
⇒ The User Agents,
⇒ The End User.

The next sections describe these modules in detail.
5. **The Chat Server**

The basic block diagram for a chat server is as follows:

The Chat Server will have the following threads: A Command Server thread and a Receiver thread will start executing as soon as the Chat Server application is started and will continue execution till the Chat Server module is terminated. The Command Server thread will listen to commands from the Users. The Receiver thread will listen to unicast messages from the other Chat Servers and other Users. Another thread will be the Sender thread that will be called to send unicast messages to Users as and when needed and will terminate after sending the message to the intended recipient. Another thread will listen to messages from other servers at a particular multicast address (Let it be 'Add'). Another thread will send multicast messages to the address 'Add'. A Timer thread will be used to make any socket receiver wait for a specific period of time, before terminating itself. A Polling Sender Thread will send unicast messages to all its users after every specified interval of time to give the message that it is still alive. A Polling Receiver Thread will receive messages from its users confirming that they are alive. The following sequence of events happen when a new server is created.

The name of the server is specified when you run the server on the command prompt. The format followed would be:

```
java server <servername>
```

Server name could be optional on the commandline. We can have a configuration file at (say) conf/server.conf which stores various parameters used by the server. The server displays the address information of its Command Processor thread on the console once
it starts. This is to facilitate prospective users to join the server. The server then goes through the **Server Discovery Protocol** which is explained on the next page.

Once the server starts executing, it waits for any users to join. When a user joins/leaves the server, it sends unicast messages to its neighbors specifying the updated list of its users. The server maintains the following databases-

1. The information of users connected to the server.
2. The information of Chat Rooms and its users, as in project 2.
3. A neighbors list listing all the neighboring servers.
4. A list of all the users connected to all its neighbors.

The server manages Chat Rooms as it did in the project 2.

The following sequence is followed when a server comes up:
The description of the various modules is as follows:

**Server Discovery Protocol:**

1. Each server would listen to a specific multicast address and a specific port.
2. When a server comes up, it broadcasts the server name. If any other server detects name collision, it responds to the server to change the name.
3. If it does not receive collision message within specified time, it will generate a random number. Then, it will broadcast a message “I am alive” along with the random number generated.
4. When another live server receives this message, it will generate a random number.
5. If the received random number comes in the range of the local random number, then the new server will be added to the neighbor list. Further, this server will inform the new server about the successful operation and will send its IP address, port and user list.
6. If the new server does not receive response from anyone, it will keep on generating random numbers and broadcasting the message until at least one server responds back.
Node Discovery Protocol

1. When a user gives command

\texttt{search \ <username> \ [<senderIP\texttt{Address}] \ <port>] \ to \ the \ server \ it \ is \ connected \ to, \ the \ server \ will \ search \ its \ own \ user \ list.}

2. If the user name is found, then it will reply to last IP address and port extracted from the received command, with the following response

\texttt{search\_response \ <user\_globalname> \ <user\_IP\_address> \ <user\_port>}

3. If the username is not found in its user list, the server will forward the request \texttt{search \ <username>}, appending its own IP address and port #, to each of its neighbor server.

4. When a server receives message \texttt{search\_response} it just forwards the message to the last IP address and port # extracted.
Polling Module

1. The server listens on a specified IP address and port for the poll messages from the user node. This is used by the server to keep a track of the status (online/offline) of all the users connected to it.
2. When it receives a poll message, it replies with the “I am alive” message. This is used by the users to keep a track of the status (online/offline) of the server.

Command Processor

The command processor receives all the command from the user node in the form of message. Message packet corresponding for each command have the following structure

```
command [argument if any] [<alias> <senderIPAddress> <port>]
```

The terms in [ ] are appended by the message sender.

The following commands are processed by the server:

createroom:

Whenever the server receives a command to create a new chat room, it must first check for any existing rooms with the same name. Also it must check if the requesting user is not participating in any other chat room by comparing the user agent's receiver thread IP address and port number. If no such room is found and the user is not part of any other chat room then it creates a new room and also generates a unique seed for that room. Next it picks up a multicast address from the range 230.0.0.1 to 230.0.0.255 that is not assigned to already existing chat rooms. It then enters this tuple value <chatroom name, seed, mcast address> in its chat room database / table. Also it maintains a list of users participating in each chat rooms. If the chat room does not exist it creates that room and immediately adds that user in the user list for that chat room. It then sends back to the user a reply message specifying the chat room seed, multicast address for that chat room. If the chat room creation fails because it already exists, server returns failure message back to the user's receiver thread address.

On successful ChatRoom creation, the server responds with a createroom response which has the following format:

```
createroom <chatroomname> <seed> <multicastaddress>
```

where, chatroomname is the name of the ChatRoom that was created, seed is the seed value associated with this ChatRoom, multicastaddress is the multicast address of this ChatRoom.
joinroom:

When the server receives a `joinroom` command, it first checks if that user is already participating in any existing chat room or not. This check is done by comparing the user agent's talk receiver IP address and port number in the list of users for any existing chatroom. If the user is already participating in any existing chat room, server responds back by a failure message specifying the name of the chat room the user is already participating in. Also if the alias is already used by some other user agent in that chat room user list, the server responds back indicating name collision to the end user agent further advising it to change its self alias and try again. Otherwise it returns back a message specifying the chat room seed and the corresponding multicast address. Server also then adds this user to the list of users for that chat room.

On success, the server responds with a `joinroom` response which has the following format:

```
joinroom <chatroomname> <seed> <multicastaddress>
```

where, `chatroomname` is the name of the ChatRoom that the user joined, `seed` is the seed value associated with this ChatRoom, `multicastaddress` is the multicast address of this ChatRoom.

leaveroom:

When the server receives a `leaveroom` command, it first checks if that user is already participating in that particular chat room or not by comparing the IP address, port number fields of the existing users for that chat room. If the user is currently participating in that chat room then the server removes its entry from the list of user for that chat room. It immediately changes that chat room unique seed, again immediately communicates to the remaining user of that chat room the new chat room seed (using UDP unicast messages) in turns. It then indicates to the original user that it has been successfully removed from that chat room. Otherwise it returns back a failure message. If the server changes the unique seed for that chat room it also updates the chat room table with the new seed value. Further if after removing the requesting user from the chat room successfully, if the chat room becomes empty, the server must destroy that chat room session by removing that room details from the existing chat room table (this is different from the user list table for each chat room).

On success, the server responds with a `leaveroom` response which has the following format:

```
leaveroom <message>
```

where, `message` is the success/failure message.
ulist:

When the server receives a **ulist** command from the user agent, it sends back the list of all users specifying their alias name along with the IP address and the port number for that chat room regardless of the requesting user participation status in that chatroom. If no such chat room exists, server returns back an error message.

On success, the server responds with an **ulist** response which has the following format:

```
ulist <message>
```

where, **message** contains all the users present in that ChatRoom.

listrooms:

When the server receives a **listrooms** command, it responds back with the list of all existing chat rooms names to the requesting user-agent talk receiver address. All the responses sent back to the agent by the server is done by spawning the sender thread.

On success, the server responds with a **listrooms** response which has the following format:

```
listrooms <message>
```

where, **message** contains all the ChatRooms present on the server.

search:

When the server receives a **search** command, it will invoke the Node Discovery Module. The user will send the search command with the following syntax:

```
search <username> [senderIPAddress <port>]
```

where, **senderIPAddress** and **port** refer to the ipAddress and port of the user node.

On success, the server responds with a **search_response** response which has the following format:

```
search response <user globalname> <user IP address> <user port>
```

where, **user_globalname**, **user_IP_address** and **user_port** are the global name, IP Address and port of the user the user searched for.
6. **The User Agent**

The block diagram for the user agent is as follows:

![User Agent Diagram](image-url)
The sequence followed when a user agent comes up is as follows:

![Diagram of sequence](image)

The description of modules is as follows:

**Fault Tolerance Manager**

1. The Server Alive Sender Thread will send “Are you alive” message to the server at every fixed time interval.
2. When the Server Alive Receiver thread receives the message from the server “I am alive”, it spawns a Timer Thread.
3. When the timer expires, it means that no message had been received within the timeout period and the server had crashed. Then, the Timer thread performs the **migrate** functionality.
4. The Timer thread will read the neighbor list of the server and will select a server at random to connect to with its global name.
Command Processor

The Command receiver thread receives messages:

⇒ When the agent (command receiver) receives a `createroom`, `joinroom`, `leaveroom`, `ulist`, `listrooms` command at its command server thread, it spawns a unicast sender thread passing it the command and the corresponding parameters. The unicast sender thread then appropriately creates a UDP message packet on the lines of project 1 containing necessary information for the server which includes the self-alias information as well as unicast receiver thread's ip address and port number and sends it to the server's receiver thread IP address and the port number.
⇒ If the agent's command receiver thread receives the `csend` command, it spawns a mcast sender thread passing to it the message as the argument. The mcast sender thread creates a new multicast message packet to send to the chat room specified by the multicast address global variable. The constructed packet's data portion must contain the unique chat room seed followed by the message in that order. It then sends that packet on the multicast channel and terminates after successful operation, otherwise it should output an error message indicating send failure.
⇒ When the agent receives `setalias` command, it sets the alias name and updates the alias table entry.
When it receives **alias** command, it updates the alias table accordingly.

⇒ When it receives **send** command, it send unicast message to the peer specified.

⇒ When it receives **remove** command, it updates the alias table accordingly.

⇒ When it receives **list** command, it displays the alias names from the alias list.

⇒ When it receives **exit** command, it terminates all the running threads and exits the system.

P2P receiving thread receives messages:

When the agent's receiving thread receives a message from the server, depending on the message type it takes appropriate action

⇒ If the received message is in response to the **createroom** or **joinroom** command, if it is a error message it just displays the message, otherwise it updates the global chat room name variable, chat room unique seed variable and multicast address variable to reflect the most recently joined chat room details. Also it spawns a new mcast receiver thread enabling the agent to start receiving messages on that multicast channel from other users in that chat room.

⇒ If it receives a message in response to the **leaveroom** command, if the message is an error message it just displays the message otherwise it terminates the mcast receiver thread.

⇒ If it receives a message in response to **ulist** or **listrooms** commands, it just displays the message, possibly after parsing it to separate out different fields / components.

⇒ If the message received is due to change of the unique chat room seed by the server then it immediately changes the global unique chat room seed to reflect the new seed.

⇒ If the received message is the neighbor list, then it stores the neighbor list locally.

⇒ If the received message is a message from a peer, then we display the message in a specific format.

⇒ If the received message is a **search_response**, then it displays the details of that user on the console. Incase multiple records are returned, it is the users responsibility to choose the correct user node it wants to talk to.
7. **The End User**

The block diagram for the user agent is as follows:

The valid commands and their syntax for the end user will be:

- **alias** `<alias name> <IP address> <port>`
  This command is used to define named aliases.

- **createroom** `<chat room name>`
  This command creates a chat room and names that chat room

- **csend** `<message>`
  This command sends a message to current chat room’s multicast address

- **exit**
  This command is used to terminate the system.

- **joinroom** `<chat room name>`
  This command allows a particular user to join the specified chat room

- **leaveroom** `<chat room name>`
  This command allows the user to leave the specified chat room

- **list**
  This command lists all the current existing aliases.

- **listrooms**
  This command lists all existing chat rooms

- **remove** `<alias name>`
  This command is used to remove an existing alias.
search <user name>
This command searches for the specified user in its user list, if not found, it forwards the search request to other servers that are connected to it, and when any matches are found it lists all matches along with the IP address and port of those matches.

send <alias name> <message>
This command sends a message to the specified alias.

setalias <alias name>
This command allows to set one's own alias name.

ulist <chat room name>
This command lists all the users that present in the specified chat room.