

CIS505 Project 3 - Final Documentation

Team: Code Damn It!

Shruthi Ashok Kumar

Adrian Diaz

Abhishek Ravichandran

OUTLINE OF PROJECT REPORT

1. DATA STRUCTURES
2. MESSAGE TYPES
3. IMPLEMENTATION OF TOTAL ORDERING
4. DETECTING A PARTICIPANT LEFT/CRASHED
5. DETECTING A COORDINATOR LEFT/CRASHED
6. LEADER ELECTION IMPLEMENTATION
7. EXTRA CREDITS
 - a. TRAFFIC CONTROL
 - b. FAIR QUEUING
 - c. GUI
 - d. ENCRYPTION OF CHAT MESSAGES
 - e. MESSAGE PRIORITY
 - f. DECENTRALIZED TOTAL ORDERING
8. USER MANUAL

DATA STRUCTURES

- **Global variables**

1. A structure **user** which stores all the information regarding a particular user, like, user ID, user name, IP address, port number, the global sequence number of the message that the user last acknowledged, the local sequence number of the message last received from that user and the timestamp at which the user was last known to be alive. Apart from this, it contains flags to indicate whether the user is currently the leader of the chat, whether the user is holding elections, whether the user is still alive (used primarily during elections) and whether the user has already joined the chat. In addition, in the fair queuing implementation, each **user** instance also contains a queue corresponding to the message requests received from that user. In the traffic control implementation, each **user** instance additionally contains the number of messages received from that user within a certain time span, and the time each user must wait before sending the next message request. In the decentralized algorithm implementation, each **user** instance also contains the last sequence number suggestion made by that user and the highest message sequence number each user has seen. All this information is maintained by the leader of the chat. A pointer to an instance of this structure called **local_user** points to all the information about the local chat participant.
2. A structure **userListObj** containing within it a single **user** and a pointer to the next **userListObj** - used to represent a single user object within a linked list of users. A local pointer **head** of type **userListObj** represents this linked list and the first instance of **userListObj** in this linked list always points to the leader of the chat.
3. A structure **messageListObj** containing within it a string representing the text of a single messages, a message ID (used to prevent duplication on the sender's side), and a pointer to the next **messageListObj**. An instance of **messageListObj** thus corresponds to a linked list of messages.

4. A structure **queueObj** containing within it an instance of type **messageListObj** - representing the list of messages within the queue, a mutex for multi-threaded access, and a semaphore for efficient threaded production and consumption of messages in the queue.
5. A structure of type `sockaddr_in` called **coordinator_socket** which stores the IP address and the port number of the leader and a socket file descriptor **local_socket_fd** for the file descriptor of the participant's socket.
6. Integers to keep track of the last global sequence number received and sent, the local sequence number of the last message request fully acknowledged and the last local sequence number acknowledged.

- **Queues**

1. **Broadcast queue:** Every participant maintains a broadcast queue, but only the current leader of the chat uses it. The broadcast queue contains message requests sent to the leader by clients. The messages in this queue are then successively broadcasted to all users in the order in which they were inserted into the queue. A particular message is removed only when the leader has received acknowledgments ('ack-msg') from all the users in the chat that they have received that message and the leader will repeatedly send the message until these acknowledgments have been received.
2. **Send Queue:** Every participant maintains a send queue which contains the messages that it wants to send to the leader (we call these message requests), so that they can be sent to the entire chat. The message from the send queue is removed as soon as the leader sends a "message-request-ack" for that message, and until then, the participant repeatedly sends the message request to the leader.
3. **Undelivered Queue:** Every time a participant wishes to send a message request to the leader, the message request is added in both the send queue and the undelivered queue. Thus, the messages in the send queue are replicated in this queue. Messages are removed from this queue only when the leader has broadcasted the user's message to the entire chat and the user who sent the initial message request for that message receives it. This queue is useful when an election has taken place and the new leader requires all the

message requests that the chat participants sent to the previous leader that were not broadcasted to all the users before the previous leader quit. When this happens, the contents of the undelivered queue are flushed into the send queue (taking care of duplication by comparing message IDs).

- 4. Notification Queue:** (Used when implementing Message Priority) Every participant maintains a notification queue, but only the leader's notification queue is active. Every time the leader generates a notice stating that someone left or joined the chat, he inserts the notice into the notification queue. The messages in the notification queue are then added into the broadcast queue for broadcasting to all users in the chat.

MESSAGE TYPES

The following table explains the type of messages being exchanged within the system and their purpose.

MESSAGE TYPE/IDENTIFIER	PAYLOAD	CONTEXT OF USAGE	FUNCTION
join	A string that holds all the information about the new user like IP, port etc.	When a new user wants to join the chat, he sends this payload to the coordinator.	send_join_msg()
ack-msg	User ID, Message sequence number	User acknowledges the receipt of a message from the coordinator. The user ID and global sequence number keep track of who acked which message.	handle_msg() handle_message_ack()
ping	User ID	User pings the server in 3-second intervals to let the server know that it is still alive	pingServer()
ping-ack		The server acks the pings that it has received from the user.	handle_ping()

coordinator-info	A string containing the IP address and port of the leader.	When a new user contacts an existing member of the chat, this payload is sent to the newcomer.	
list	A string comprising of all the details about all the members in the chat	When a user joins, leaves or the leader changes.	
message-request	The message the user intends to send to the entire chat, the user's ID number and the local sequence number of the message.	An user sends the message to the leader. It can also indicate a "notice" being sent.	handle_msg_request()
message	A string comprising of the message, the ID of the sender, the sender's local sequence number and the global sequence number	Leader sends this payload to all members of the chat.	handle_msg()
quit	User ID	When a user quits the chat	handle_quit()
alive-req		When a user wants to check if other users with lesser ID than itself are alive while holding elections	handle_alive_req()

alive-yes		When a user acks to a “alive-req” that it is indeed alive	handle_alive_yes()
message-request-ack	Local message sequence number	Updates the last_local_seq_no_acked to the sequence number received as payload	
socket-used		Prints that socket is already in use.	

IMPLEMENTATION OF TOTAL ORDERING

- The leader maintains a sequence number counter. When any user sends the leader a message to be multicast, the leader assigns a sequence number to the message (only at the time of sending!) and then sends it out to all the users.
- When a participant joins, the leader assigns to them an ID number which is one more than the ID number of the last person who joined.
- Each user also maintains a record of which sequence number it has seen previously. When a participant receives a message from the leader’s multicast, it only displays the message if the sequence number is one more than the last seen sequence number.
- Each user has to acknowledge the receipt of the message from the leader. The leader will not multicast the next message in the queue unless all the users acknowledge the currently multicast message.
- Dealing with delayed or lost messages:
 - The leader keeps on sending the message to the user (with a slight delay between the message so as to not overwhelm the client) until it

receives an ack from the participant. Thus, even if one packet gets lost/dropped, the user is sure to get the packet after a slight delay.

DETECTING A PARTICIPANT LEFT/CRASHED

- Every user “pings” the leader every 3 seconds. The leader responds with a “ping-ack”.
- If the leader does not receive “pings” from any user 7 seconds, it declares the user to be dead/crashed and sends out a notice.
- The users keep track of time by updating the timestamp “time_last_alive”.
- A separate thread “pingThread” handles this functionality by invoking “pingServer”.
- The leader uses a thread “checkClientTimeStamps” to update the “time_last_seen” of the users and to check if the user has been silent for more than 7 seconds.

DETECTING A COORDINATOR LEFT/CRASHED

- When any user receives “ping-acks” from the leader for its “pings”, it updates the “time_last_alive” of the server.
- When no “ping-acks” arrive, the difference between the current time and the “time_last_seen” keep increasing and once it exceeds 7 seconds, the user thinks the leader is dead/crashed and calls for fresh elections to elect a new leader.
- The users use the thread “checkServerTimeStamps” to check the “time_last_alive” of the server and to determine whether the server has been silent for more than 7 seconds.

LEADER ELECTION IMPLEMENTATION

- We implemented the bully algorithm presented in class, with the difference that we are using the lowest user id (instead of largest) for priority, as a lower ID number corresponds to a user who joined earlier.

- When any user detects that the leader is no longer active, it will initiate the election process by sending “alive-req” to all the users who have ID less than its own from the list of users it has.
- The users who received “alive-req” will reply with a “alive-yes” to indicate that they are alive.
- These users proceed to hold elections among themselves and the user with the lowest ID (the person who has been in the chat for the longest time) gets elected as the leader.

EXTRA CREDITS

TRAFFIC CONTROL

- When any user sends more than 10 unique messages (uniqueness determined by looking at the message sequence number) within a duration of 3 seconds, the leader asks the user to sleep for 2 seconds.
- A separate thread “TrafficControlThread” handles this functionality and sends a message to the user with the header “traffic” and a payload with the number of seconds by which the client should slow down.
- The code can be found in “dchat_traffic_control.c”.

FAIR QUEUING

The code for fair queueing is in fair_dchat.c and was implemented with the following changes off our base code:

- We added message queues to each user struct.
- When a message request comes in, the coordinator adds the message to the corresponding user’s queue (instead of the broadcast queue)
- We have a thread running checkFairQueues which performs a round robin check on each user’s queue. If the user’s queue is non-empty, the thread pops off the head message from that user’s queue, and adds it to the broadcast queue.

GUI

The GUI was implemented with GTK+ (GIMP Toolkit) and written in c. It consists of a scrollable text buffer, text entry field, and exit chat button. We considered using other GUI's but wanted to keep the language consistent so it was easily integrated with the existing code base. The code is in `gtk_dchat.c` NOTE: to compile/execute this program your system needs to have the GTK library installed. The major required changes from the original `dchat.c`:

- We had to run the client on a separate thread and run GTK in main, as this was the recommended approach.
- The semaphore implementation was modified as we developed the GUI in mac, so we needed to use `sem_open`, etc.
- The client needed to wait until GTK widgets were initialized and rendered before printing anything to the buffer requiring additional semaphores.

ENCRYPTION OF CHAT MESSAGES

- All messages and notifications were encrypted using a substitution cypher that applies an offset to each character.
- The messages are encrypted before entering the `sendQueue`.
- When a client receives a message, it decrypts it before printing it to the console.
- The code can be found in `encrypt_dchat.c`. For testing purposes, you can run a client off of `dont_decrypt_dchat.c` to see that his messages are encrypted while all others appear normal.

MESSAGE PRIORITY

- All “notices” are added to the “notificationQueue” rather than the broadcast queue.
- A global variable keeps track of the number of notices in the broadcast queue.
- A separate thread “checkNoticeQueue” keeps checking the notice queue and as soon as there are notices, it appends these to the broadcast queue after the notices that are already in the broadcast queue - this is made possible by the

“noticesInBroadcastQueue” global variable, which we increment after adding each notice to the broadcast queue. Also, as long as there any messages in the broadcast queue, this variable is kept at a value of at least 1. This is to account for the fact that we are implicitly considering the message at the top of the broadcast queue as a notice and do not touch it, as the broadcasting thread might have already assigned a sequence number to this message and started multicasting this message.

- As messages keep getting sent out from the broadcast queue, this variable keeps getting decremented, until it reaches 1. It is made 0 when there are no messages left in the broadcast queue.
- In this way, the broadcast queue consists of all the notices first and then all the messages. Since this is a FIFO queue, the notices get sent out first before the messages, thus implementing priority.
- The code can be found in “dchat_msgp.c”.

DECENTRALIZED TOTAL ORDERING

- The code can be found in “decentralized.c”.
- The base code has been modified so that there is no leader and the user structure now holds data about the sequence number suggested by that user for the current message, and the highest message number that user has seen (only used during initialization - the joining user sets the local value of ‘highest_suggested’ using this value.
- When any user has to send a message, it asks all the users (including itself) to suggest sequence numbers by sending out sequence number requests. The other users each propose a sequence number that is one more than the value of its local “highest_suggested” count, and increments this. Note that this happens only for unique sequence number requests - duplicate sequence number requests are dealt with by appending each sequence number with a local sequence number. The user who receives the sequence number request only responds if this request is greater than the last sequence number request received.

- The user who requested the sequence number chooses the highest of all proposed sequence numbers and broadcasts the message along with this sequence number.
- As a user receives a message, it updates its “highest_seen” value appropriately. If the highest seen is greater than the highest suggested thus far, it updates its highest suggested value to the highest seen value.

USER MANUAL

COMPILING THE BASE CODE

- Go to folder “regular_credit”
- Run “make dchat”

STARTING THE CHAT

```
./dchat Bob
```

JOINING THE CHAT

```
./dchat Alice 172.17.11.24:8000  
./dchat Tom 172.17.11.125:8002
```

TRAFFIC CONTROL

- Go to folder “extra_credit”
- Run “make dchat_traffic_control”
- ./dchat_traffic_control Bob
- ./dchat_traffic_control Tom 172.17.11.125:8002

MESSAGE PRIORITY

- Go to folder “extra_credit”
- Run “make dchat_msgp”
- ./dchat_msgp Bob

- `./dchat_msgp Tom 172.17.11.125:8002`

ENCRYPTED MESSAGES

- Go to folder “extra_credit”
- Run “make dchat_encrypt”
- `./dchat_encrypt Bob`
- `./dchat_encrypt Tom 172.17.11.125:8002`
- For test case to see user who doesn’t decrypt
 - Go to folder “extra_credit”
 - Run “make dchat_dont_decrypt”
 - `./dchat_dont_decrypt Bob`
 - `./dchat_dont_decryptt Tom 172.17.11.125:8002`

FAIR QUEUEING

- Go to folder “extra_credit”
- Run “make dchat_fair”
- `./dchat_fair Bob`
- `./dchat_fair Tom 172.17.11.125:8002`

DECENTRALIZED TOTAL ORDERING

- Go to folder “extra_credit”
- Run “make dchat_decentralized”
- `./dchat_decentralized Bob`
- `./dchat_decentralized Tom 172.17.11.125:8002`

GUI - NOTE: THIS REQUIRES YOUR SYSTEM TO HAVE GTK

- Go to folder “extra_credit”
- Run “make dchat_gui”
- `./dchat_gui Bob`
- `./dchat_gui Tom 172.17.11.125:8002`