CIS505 Project 3 - Final Documentation

Team: Code Damn It!

Shruthi Ashok Kumar

Adrian Diaz

Abhishek Ravichandran

OUTLINE OF PROJECT REPORT

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 sequence number of the message that it last acknowledged, the sequence number of the message that it last received and the timestamp at which it was last known to be alive. Apart from this, it contains flags to indicate whether it is the leader or not, whether it is holding elections or not, whether the user is still alive, whether it has joined the chat. A pointer to this structure called **local_user** points to all the information about the current user himself.
2. A linked list called **userListObj** which is nothing but a list of information about all the users. A pointer **head** points to this linked list i.e, always points to the leader of the chat.
3. A linked list called **messageListObj** which is nothing but a list of messages along with a message ID.
4. A structure of type sockaddr_in called **coordinator_socket** which stores the IP address and the port number of the leader.
5. Integers to keep track of last sequence number received and sent, local sequence number and last sequence number acknowledged.

- **Queues**

1. **Broadcast queue** : The leader maintains a broadcast queue. All the messages being sent to the leader are added to this queue and then multicast to all the users in the chat.
2. **Send Queue** : Each participant maintains a send queue which contains the messages that it wants to send to the leader for multicasting. The message from the send queue is removed as soon as the leader sends a “message-request-ack” for that message.

3. **Undelivered Queue** : The messages in the send queue are replicated in this queue. Messages are removed from this queue only when the leader multicasts the user's message and is received by the user to be displayed on the console. This queue is useful when there is a new leader elected and he has to first send all the messages that were not multicast by the previous leader.

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 sequence number keep track of who acked which message.	handle_msg() handle_message_ack()
ping-ack		The server acks the pings that it has received from the user.	handle_ping()
coordinator-info	A string comprising of all the details about 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	Message	An user sends the message to the leader. It can also indicate a “notice” being sent.	handle_msg_request()

MESSAGE TYPE/IDENTIFIER	**PAYLOAD**	**CONTEXT OF USAGE**	**FUNCTION**
message	A string comprising of the message, the ID of the sender, the sender's 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	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 and then sends it out to all the users.
- When a participant joins, the leader sends them a sequence number which is one more than the current sequence number.
- 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 ack 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
- 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.
- 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 payload “slowdownby” which tells the user to sleep for “slowdownby” seconds.
- 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, pops off the head message, 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 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 char
- 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 “noticesinBroadcastQueue” messages.
- As messages keep getting sent out from the broadcast queue, this variable keeps getting decremented.
- 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.
- 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 highest suggested sequence number and the highest sequence number seen.

- When any user has to send a message, it asks all the other users to suggest sequence numbers. The other users each propose a sequence number that is one more than the value of “highest_suggested” which is a global variable.
- The user who invoked the “seq_no_req” chooses the highest of all proposed sequence numbers and broadcasts the message along with this sequence number.
- The users’ variables such as “highest_suggested” and “highest_seen” get updated accordingly.

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