Unreliable Chat くコ8彡

CIS 505 Spring 2016 Project 3 Documentation

くコ8彡 Spencer Caplan pennkey: spcaplan
くコ:彡 Jordan Kodner pennkey: jkodner
くコB彡 Anupama Kumar pennkey: anupamak

User Guide くコ:彡

Command Line Arguments

The following sections explain how to start the program

Building the Code

Make has two options: all and clean.

The following performs a clean followed by a make all

```
$ make clean && make
```

Starting a New Chat

To start a new chat, specify only your desired port number and the ui flag. The ui flag may be set to 0 for default ui or to 1 for extra credit ui

Format. Note, all arguments are space delimited

```
$ ./dchat <my portnum> <ui flag>
```

Example: Start on port 5000, no extra credit ui

```
$ ./dchat 5000 0
```

Joining an Existing Chat

To join an existing chat, you need to specify the ip and port for some member of the existing chat

Format. Note, all arguments are space delimited

```
$ ./dchat <my portnum> <ip for existing chat> <portnum for existing cat> <ui flag>
```

Example: Join a chat at 123.456.78.900:4321, use port 6000, use extra credit ui

```
$ ./dchat 6000 123.456.78.900 4321 1
```

Using the Default UI

After entering command line arguments, splash text will be displayed and you will be prompted for your username. Enter a username without spaces, and press enter. If you created a new chat or joined successfully to an existing one, you will be able to send messages. Updates show up in cyan. Chat messages are in yellow. Ctrl-D to exit.

Using the Extra Credit UI

After entering command line arguments, splash text will be displayed and you will be prompted for your username. Enter a username without spaces, and press enter. The UI will start up and the bottom left window will be highlighted yellow.

Press tab twice to highlight the blue info window. Then press h to toggle help to display the help menu.

Enter messages from the input window. Scroll through messages in the message window. View status updates in the info window. Ctrl-D to exit.

##Code & Protocols ＜コB彡##########

Message Protocol

The following describes the format and types of packets sent over UDP

Packet Layout

Message packets will be no larger than 1K. This means that larger messages may need to be reconstituted from several packets.

Packet components are newline-delimited. This is safe given user input from the commandline.

Packet Format

```
<sender>\n
<uid>\n
<senderuid>\n
<message type>\n
<packet #>\n
<total # packets expected>\n
<message body>
```

1. `<sender>` Name of client who sent the packet
2. `<uid>` a uid for this packet or message
3. `<senderuid>` The uid for the client who sent the message (IP:PORT)
4. `<message type>` what kind of packet was sent
5. `<packet #>` 0 if this packet contains an entire message. Otherwise represents this packet's sequence within the message
6. `<total # packets expected>` How many packets make up the associated message
7. `<messagebody>` Payload to be delivered

For example, a short chat message sent from user UnreliableChatter:

```
UnreliableChatter
236149324^_^5
123.456.78.900:5000
CHAT
0
1
Hi, Guys. How are you doing?
```

Packet Types

The following types of packet are used in the protocol

- CHAT - a chat message meant for display
- SEQUENCE - a multicast sequencing message from the leader
- CHECKUP - a heartbeat message
- ELECTION - an election-initialization message
- VOTE - an election message indicating the sender should lead
- VICTORY - an election completion message
- JOIN_REQUEST - a message requesting to join the chat
- LEADER_INFO - information on the leader sent to a new joiner

- JOIN - a message announcing a new member
- EXIT - a message announcing that someone has left the chat
- QUORUMRESPONSE - a message for counting quora during elections and checkups
- CONFIRMDEAD - a message that votes someone has gone offline
- CONFIRMCOUP - a message confirming an election candidate

CHAT

A chat message is partitioned into enough 1KB packets to send the entire message body. Each packet shares a common UID. They differ in their packet numbers.

SEQUENCE

A sequencing message's UID corresponds to the message that should be sequenced. Its message body is the sequencing number that the clients should use.

CHECKUP

A checkup message contains the following possible payloads:

- "ARE_YOU_ALIVE" This indicates a message asking for the receiving client to respond to the sender with an "I_AM_ALIVE" message (heartbeat confirming the receiver's state of still being alive and active in the chat)
- "I_AM_ALIVE" This message is sent back to the sender of an "ARE_YOU_ALIVE" message to indicate that the receiver is alive and well within the chat program

ELECTION

An election message contains no additional information. Its message body is always "INITIATE_ELECTION" This indicates that each receiving client should set their `election_happening` global variable to true. That variable is in turn read by the checkup thread to trigger holding a new leader election.

VOTE

An vote message contains no additional information. Its message body is always "I_SHOULD_LEAD" The senderuid indicates who sent the message.

VICTORY

A victory message's message body is the uid of the sender i.e. the election winner. When a node receives a VICTORY message it updates the information for who the leader is.

JOIN_REQUEST

A join request's message body is the sender's ip and port (IP:PORT). This is sent so that the recipient knows where the new client is located. The IP:PORT format also represents the client uid.

LEADER_INFO

A leader info message body contains the username and uid (IP:PORT) of the leader.

JOIN

A join indicates that a new client has joined. It contains a list of username:IP:PORT combination, one for each client. The first client in the list is the new client. The second client in the list is the leader.

EXIT

An exit message indicates that someone has been deemed offline by vote and should be removed from the list of clients.

QUORUMRESPONSE

A quorum message body contains the uid of the client whose liveness status is being voted on.

CONFIRMDEAD

A confirm dead message body contains one of the following values:

- “YEP_THEY_ARE_DEAD” This indicates the sending clients belief that the clients currently being voted on is dead.
- “NO_THEY_ARE_ALIVE” This indicates the sending clients belief that the clients currently being voted on is alive.

CONFIRMCOUP

A confirm coup message contains no additional information. Its message body is always “LONG_LIVE_THE_KING.” This is used to update the global variable for each client that the leader election results have propagated to the new leader.

Algorithms

The following section explains the algorithms used in the chat

Sequencing

All CHAT and JOIN messages are totally ordered by a central sequencer, the leader.

CHAT and JOIN messages are multicast to every client, including the sender. When a client receives a packet corresponding to a CHAT or JOIN message, it creates a (possibly incomplete) message and places it in a list of unsequenced messages. Additional packets with the same uid are collated to complete the message.

When the leader receives a CHAT or JOIN, it performs the above actions then assigns a sequence number through a fair sequencing process (*see extra credit*). Once it determines the number, it multicasts a SEQUENCE message.

Upon receiving a SEQUENCE message, it the client has a complete message in its unsequenced messages list, it assigns it the sequence number and moves it to a holdback queue. It then prints all the messages in the queue with consecutive sequence numbers because these are guaranteed to be totally ordered. It stops at the first gap in the sequence numbers because it has not sequenced that message yet.

If a SEQUENCE message is received which does not correspond with any completed messages, it gets added to a list of unaccounted for sequencing messages. When the matching CHAT or JOIN eventually arrives, it is then sequenced immediately.

If the leader dies, everyone continues to receive CHAT messages (JOINing is suspended until the new leader is elected). This remain unsequenced until the new leader is chosen. The new leader then goes through its backlog of unsequenced messages and assigns sequences to them and multicasts the appropriate SEQUENCE messages

If all else fails, a message will be skipped with notification if its sequence number cannot be recovered.

Joining

When a new client wishes to join, it may contact any existing chat member with a JOIN_REQUEST messages. If a client which is not the leader is contacted, it will respond with a LEADER_INFO message containing information on how to contact the leader. The new client then automatically sends a JOIN_REQUEST to the leader. When the leader receives a JOIN_REQUEST, it marshals the info for all current clients into the message body of the JOIN message and multicasts it.

When the new client receives a JOIN message, it reads its own info (username:IP:PORT) off first. Then it reads off the leader's info. Then it reads off everyone else's info. It creates the appropriate data structures for each client this way.

If anyone else receives a JOIN, it reads off just the first client info since this corresponds to the new client. It then adds this info to the appropriate client data structures.

If the new client cannot reach the client specified in its command line args, like if the ip was entered incorrectly, it will timeout and exit. If there is no leader, nobody can join until a new leader is elected. This is governed by the timeout for the election and the joining timeout. Once the leader is elected, everything proceeds normally again

If the new client's info matches an existing client (like if a client died then the user manually rejoined before the client's death registered), it will never receive a JOIN response. It will timeout and exit. Retrying after a few seconds, when the death is registered, will be successful.

Checkup

Every three seconds each client multicasts a checkup "heartbeat" to every other client. This checkup process is run on a dedicated thread so as not to interfere with the external chat functionality.

Upon receipt of this checkup heartbeat clients will respond to the sender with a simple message over UDP indicating that they are still active. Each client keeps a tally of the number of checkup heartbeats which other clients have failed to respond to. If that tally exceeds three missed checkups then the offending client is presumed to be dead. In this way each client has a running count of the status of all other clients.

Since the leader node is responsible for sequencing and joining, it is also the only node which can initiate a call to remove a dead client from the chat. If the leader detects that a client has missed more than three checkups then it multicasts a message to everyone in order to establish a quorum. This is required in order to protect against a case in which a client is in fact alive but only the network connection between it and the leader is experiencing problems. When clients receive a quorum request from the leader they will respond in turn via UDP message with their current status of the accused-to-be-dead client. If the leader gets back at least 50% of responses which agree with the determination that a given client has died then it multicasts a message instructing all other clients to remove the dead node from their chat lists.

Election

When a non-leader node detects that the leader has died then an election must be held. First a quorum (see above section on Checkups) must be established that at least 50% of all nodes agree that the leader is non-responsive to checkups. If a quorum has been established then the client multicasts a message to remove the old, dead leader from their chat lists and then multicasts an ELECTION message to everyone. Upon receipt of an ELECTION message a client will set a global variable which is to be read by the checkup thread.

When the checkup thread reads that the election variable has been set then it temporarily stops broadcasting checkup heartbeats and begins new leader election. The new leader will be selected based on which active client has the a HOSTNAME:PORTNUM combination with the larger value according to the C function strcmp().

The process for leader election is that every node broadcasts a VOTE message with its HOSTNAME:PORTNUM indicating that it currently believes itself to have the largest value. When a node receives a VOTE message it compares the value to that of the node it is currently

deferent to (i.e. the node it currently believes to have the highest value). If the incoming value from a VOTE message is higher than its current deference then it updates its deference accordingly. Otherwise if its existing deference is higher than no action need be taken.

Clients default to assuming that their own value is the highest and will continue to broadcast VOTE messages every second until they receive an external VOTE message with a higher value than their own.

At each second interval each client checks the number of “votes” it currently has (number of other nodes which are deferent to it). If all active clients (since the old leader has already been removed) are voting for the same node then that node declares itself the winner of the election and multicasts a VICTORY message which includes its UID.

When a node receives a VICTORY message it iterates over the local linked list of CLIENTS. For each local client it updates the deference value to the new leader.

When the local copy of the node which represents the new leader is reached it is updated to know that it is now the leader. At this point a global variable coup_propogated is set to reflect that the new leader knows its status.

When clients read that the coup_propogate variable has been set the checkup threads return to the normal behavior of broadcasting heartbeat messages.

An additional condition which is handles within the election function is if other non-leader clients die in between the time of death of the previous leader and the end of the new leader’s election. In this case there exist clients which are still held in each node’s local CLIENTS list but which are non-responsive (and so no client will be able to achieve an apparent unanimity of votes). In this case there is a timer in the leader election such that if the process has taken longer than eight seconds then the conditions for a node broadcasting victory are relaxed to only greater than N/2 votes where N is the number of clients after removing the dead leader.

If more than half of the active clients die concurrently during the interim between the death of the old leader and the result of the election then a catastrophic network outage is considered to have occurred and clients will exit if the connectivity situation has not resolved itself within thirty seconds.

Code Layout

Files

The following relevant files are in the repository:

- **`clientmanagement.c/.h`** functions for maintaining the client list
- **`dchat.c/.h`** declaration of constants, types, global datastructures, main, functions for joining, checkups, global data structures:
  - **`packet_t`** type for data packets sent over UDP
  - **`chatmessage_t`** type for messages reconstituted out of packets
  - **`client_t`** type for client info
  - **`UNSEQ_CHAT_MSGS`** list chat messages that haven't been sequenced yet
  - **`CLIENTS`** list of clients
  - **`STRAY_SEQ_MSGS`** sequence messages that arrived before their corresponding chat messages
  - **`HBACK_Q`** holdback queue for sequenced messages
  - **`me`** client_t corresponding to me
  - Many global variables indicating state
  - Mutexes for these global variables
- **`llist.c/.h`** our thread-safe linked list implementation
- **`Makefile`** make file
- **`messagemanagement.c/.h`** functions for handling chatmessage_t
- **`messagingprotocol.c/.h`** functions for sending, receiving, and processing packets
- **`queue.c/.h`** our thread-safe priority queue implementation
- **`schedule`** milestone 1
- **`send_msg.h`** a function declaration for the ui
- **`unreliablesplash.c/.h`** ncurses ui implementation

Threads

Our implementation makes use of the following threads

- Main thread - starts up program
- Send thread - processes and sends user input
- Receive thread - receives and does light processing of UDP packets
- Checkup thread - performs checkups and elections
- Fair Sequencing thread - performs sequencing on the round robin data structures
- Animation thread - animates the logo if the extra credit UI is running

Constants and Assumptions

The following details the hard-coded constants are made:

- IPs are no more than 32 characters and in IPv4 format
- Usernames are under 64 characters
- Messages are under 7500 bytes (10 UDP packets' payload's worth)
- New clients fail to join after 5 seconds
- Checkups are every 3 seconds
- Clients are killed after 3 missed checkup "heartbeats"

- Quora for client removal are ended after 8 seconds
- After 8 seconds, elections become less strict. Instead of requiring unanimous votes, votes are only required from a simple majority. After 30 seconds from the start, a catastrophic outages is assumed
- Elections have a sleep interval of 1 second
- Fair sequencing happens every 50 milliseconds (can be changed to 5 seconds for demo purposes)

The following assumptions are made:

- It is not possible to join while there is no leader
- If more than half the clients die “simultaneously” this is considered a “catastrophic outage” and the chat cannot proceed is because half the clients are need to establish a quorum.
- If a chat message is missing a packet, it cannot be sequenced. This can only happen with very long messages and a dicey network.
- If the sequencer gets far ahead of a client, the client is forced to skip over unsequenced messages with a warning. This can only happen if multiple SEQUENCE packets are lost quickly.
- Messages sent during an election will not be sequenced (and thus not displayed) until after the election is over. This is necessary because the sequencing is done according to the leader. The new leader is able to sequence this messages since they are still collected.
- Sequencing messages can be received before their chat messages
- Only CHAT and JOIN messages are sequenced
- Usernames cannot contain spaces or newlines
- Pasting a newline into a message will cause two messages to be sent, broken on the newline
- The user cannot copy/paste in Extra Credit UI mode, and message length is bound by the size of the text entry window
-

##Extra Credit ＜コ：ミ##################

Point to other parts in the doc where extra credit tasks are explained

UI

See the User Guide for information on how to use the UI

The UI was implemented using ncursesw, the unicode-friendly variant of ncurses (＜コ:ミ). This is the standard C library for text window manipulation. See files `unreliablechat.c` and `unreliablechat.h` for the code.

Implementing the UI this way gave us a retro character that we felt would be lacking if we used a standard GUI library. The additional challenge with ncurses is that it's very bare-boned. There are no built-in functions for scrolling windows, text bubbles, ascii art, window focusing, or even backspacing. We had to implement all that. The title window is on its own thread which allows it to be animated.

Fair Sequencing

All sequencing in this project is fair sequencing. Each client has a fair sequencing thread (fair_sequence in `messagingprotocol.c`). When the client becomes the leader, it automatically enters the sequencing loop.

If the leader, the client places all completed unsequenced messages in individual lists, one for each client. It then loops through the individual client's lists in round-robin fashion with a short time delay. It then pulls off and sequences the first message from each client's list. This ensures fair sequencing.

Otherwise, sequencing proceeds with the same SEQUENCE protocol described earlier.

Message Priority

Fair sequencing only applies to CHAT messages. JOIN messages are exempt. They are sequenced immediately in FIFO when the leader receives them. This ensures that they will be immediately displayed rather than getting stuck in the round robin lists.