# CIS 505 - Final Project Documentation

Andrew Righter
andrew.righter@gmail.com

Michael Collis
mcollis@seas.upenn.edu

04/22/2013

## 1 Project Summary

dChat is a fully distributed group chat system built on top of a multicast UDP protocol that runs on any POSIX-compatible command line interface. The goal of the project was to guarantee a total ordering of messages sent by chat participants, given the inherent unreliability of the links being used for communication and the possibility of node crash faults. dChat is based on a sequencer-client model. In the event of the failure or exit of the sequencer node, dChat implements an election protocol to select a new message sequencer. Furthermore, dChat is designed to tolerate and recover from UDP-based failures (dropped packets, duplicate packets, incorrect ordering of packets).

Due to the difficulty of completing the assignment in the allotted timeframe with only two team members, not all the features described herein are fully implemented in the submitted version of dChat. The code is commented to indicated where such features would be implemented should additional time have been available.

## 2 User Manual

### 2.1 Compiling dChat:

Move the dChat files to the desired destination directory and, on the command line, run *make all* to compile dChat. The program consists of a single multi-purpose executable that works for both creating and joining chat rooms.

### 2.2 Creating a new chat session:

Any dChat user can create a new chat session and invite friends to join in. Run the dChat executable with your requested chat nickname (see below for example usage). To permit

other dChat users to join your new chat session, you must share the IP address and port number printed when the chat session has been started successfully.

Snippet 1: Creating a new chat

```
1 ./dchat NICKNAME // Starts a new chat on the default network interface
```

If there is a problem initializing the socket on which dChat communicates with other clients, or the required memory allocation operations fail, the user will be informed that a new chat could not be created at this time and dChat will exit.

## 2.3 Joining an existing chat session:

In order to join an existing chat session, run the dChat executable with your requested chat nickname and the IP address and port number another user currently participating in the chat session in question (See below for example usage).

Snippet 2: Joining an existing chat

```
1 ./dchat NICKNAME IP_ADDRESS:PORT // Joins the chat on the specified IP address and port
```

If there is no currently active dChat session on the IP address and port or if the connection is refused for some reason (see Handling network issues), the user will be informed that the chat session could not be joined and dChat will exit.

## 2.4 Exiting a chat session:

Any chat participant may choose to leave at any time, even if are the creator of the chat session or your node is currently the message sequencer (the distinction between client and sequencer will not be visible to end users in any case). Pressing CTRL-D on the command line (sending the EOF ASCII character) will disconnect you from the chat session.

The chat session persists until the last participant leaves and users may rejoin the session at any time, as long as it is still running.

## 2.5 Handling network issues:

Users operating inside a LAN subject to NAT by a local gateway may experience connection problems attempting to create or join a chat group located on a separate subnet. This issue arises from the automatic local IP detection mechanism employed by dChat that allows users to join existing chat without needing to know their own local IP address.

Additionally, certain ports may be blocked or reserved by the user's operation system, or by that of the remote user to which the local user is attempting to connect.

# 3 System Architecture

All chat participant nodes operate the executable and multithreaded process structure regardless of whether or not the node is the sequencer. Each process operates 4 threads across 2 processes at any given time. Running the program starts the main thread which forks and executes the registration of the RPCs with the system. This process continues to run in the background listening for RPC invocations by client nodes. Unless the node is the sequencer, no clients register with this process. The RPCs are only invoked by clients on the IP address and port number of the sequencer, as indicated by the list of chat participants disseminated to all users when a new user joins the chat and when a user exits. The client process operates three threads to handle its responsibilities: Main thread for initializing data structures, spawning additional threads and listening for user input; message handler thread for listening for incoming UDP socket connections from the sequencer and handling new messages and requesting undelivered messages; heartbeat thread responsible for periodically checking that the sequencer is still active and handling election of a new sequencer.

## 3.1 Leader/Sequencer:

The leader is responsible for ordering incoming messages from each client and sending them out via multicast UDP sockets to all members of the chat group. It must maintain a buffer of recent messages and be able to retransmit messages based on their sequence number when requested by clients for whom the initial UDP delivery failed. It is also the responsibility of the leader to detect and eliminate exited/crashed users in the chat group. The leader also performs all responsibilities required of client nodes; from the user's point of view, the identity of the sequencer is transparent.

## 3.2 Client:

A user can either create a new chat group or enter an existing one. If the user starts a new group, that user is then marked as the leader/sequencer and handles all sequencer functions. If the leader leaves the chat at any time, an election is called among current active users to elect the next leader/sequencer. To handle the case of a crashed leader node, any client that doesn't receive a heartbeat response (see next subsection) from the leader in a given amount of time initiates a leader election to choose a new leader/sequencer.

In addition to these basic chat group management functions, the client is responsible for displaying chat messages relayed from the sequencer in the correct order, according to the numbering assigned to the message. This is accomplished with a holdback queue. It also maintains a list of the other group participants, used to implement the leader election protocol.

### 3.3 Heartbeat:

Each client implements a heartbeat thread that calls an RPC at regular intervals to make sure that the sequencer is still live and responding to requests and also to inform the sequencer that it is still in the chat session and responsive. In the event that a heartbeat isn't heard from the sequencer for a short time, the heartbeat thread calls an election by multicasting a message to this effect to all the other chat participants.

### 3.4 External Libraries:

RPC protocols were specified in XDR[1], and compiled into client and server stubs with RPCGEN.

Chat client multithreading is implemented using the POSIX pthreads library to spawn worker threads to handle input from the user, socket connections from the sequencer, and heartbeat messaging and sequencer elections[2].

## 4 Protocol Design

The chat functionality is implemented by having clients make calls to the registered RPC service on the servicer's open port to register their arrival in the session, deliver messages, request redelivered messages and send heartbeat messages. Messages are delivered to clients over synchronous UDP socket connections to the message handler threads.

### 4.1 RPC Functions:

| | |
|---|---|
| JOIN(): | sends a message to the leader announcing the client's request to join the session |
| SEND(): | delivers a user's input message for sequencing and broadcast to the leader |
| EXIT(): | notifies the sequencer of a client's wish to exit the chat session |
| REDELIVER(): | requests the sequencer redeliver a previously undelivered message, identified by sequence number |
| HEARTBEAT(): | sends a heartbeat message to the sequencer indicating liveness and receives an acknowledgement |
| SHUTDOWNSERV(): | shuts down the RPC listening process and destroys the outstanding data structures |

[1]RFC-1832 XDR: External Data Representation Standard (https://www.ietf.org/rfc/rfc1832.txt)
[2]POSIX Threads Programming (https://computing.llnl.gov/tutorials/pthreads/)

## 4.2 Data Structures:

We impose a number of constraints on the lengths of user inputs to dChat to better meet the limitations imposed by the UDP multicast protocol. The goal was to enable each client message to be reliably sent in a single UDP packet by relying on the IP specification requirement that all hosts accept datagrams of 576 bytes[3].

Snippet 3: RPC constants

```
1  // Maximum permitted length of an individual message (including null−terminator)
2  #define MAX_MSG_LEN 512
3
4  // Maximum permitted user nickname length (including null−terminator)
5  #define MAX_USR_LEN 32
6
7  // Maximum length of IP address string
8  #define MAX_IP_LEN 32
9
10 // Number clients initially provisioned for in the sequencer's data structures
11 #define INITIAL_CLIENT_COUNT 8
12
13 // Number of most recent user messages retained by the sequencer
14 #define MSG_BUF_SIZE 256
```

Client information is packaged and sent to the sequencer when a new participant joins a chat session. This information added to the sequencer's participant list and multicast to all participants. This broadcast also occurs when a participant exits the chat. The goal is to ensure that all chat clients have an accurate record of the other participants in the event that the sequencer exits or fails.

Snippet 4: RPC Client record structures

```
1  // An individual username (31 character max)
2  typedef char *uname;
3
4  // An individual IP address record (31 character max − in practice, the longest possible
5  // string is XXX.XXX.XXX.XXX [15 characters, plus null−terminator]
6  typedef char *hoststr;
7
8  // An individual chat client record
9  typedef struct cn {
10         uname userName;
11         hoststr hostname;
12         int lport;
13         int leader_flag;
14 } cname ;
```

[3]RFC-791 Internet Protocol Specification (https://tools.ietf.org/html/rfc791)

```
15
16 // List of chat clients
17 typedef struct cl {
18         struct {
19                 u_int clientlist_len;
20                 cname *clientlist_val;
21         } clientlist;
22 } clist;
```

## 4.3 Message Formats:

All messages sent between clients and the sequencer (with the exception of join and exit commands described above) are standardized into a single message struct. Text messages from other chat users intended for display on the client's terminal are distinguished from system messages indicating, for example that the sequencer has quit the chat, and that an election needs to be held.

Snippet 5: RPC Message structures

```
1  // Message string input by the client (511 character max)
2  typedef char *msg_send;
3
4  // Message struct packaging information between the client and
5  // sequencer or among clients during an election
6  typedef struct msg {
7          msg_send msg_sent;
8          uname user_sent;
9          int seq_num; // Set to 0 initially when the client transmits the message
10                        // only filled in when the sequencer applies an ordering
11         int msg_type; // Indicating if the message is a Text or System message
12 } msg_recv;
```

Text messages received over UDP multicast from the servicer are put into a binary-heap-backed holdback queue that is used to order messages by their sequence number. The message handling thread peeks at the front of the holdback queue and, if the queue isn't empty, checks that the message has the next sequence number it is expecting. If not, the missing message(s) is retrieved from the sequencer with a call to the REDELIVER RPC. Redelivered and correctly-ordered messages are printed to the client's console in strictly increasing order of their sequence number.

System message are handled on the fly by the message handling thread, updating the list of users and initiating a sequencer election as required.

All clients sent heartbeat messages to the sequencer thread at predetermined intervals. These are used to indicate the liveness of both the client to the sequencer and of the

sequencer to the clients. These are invoked with an RPC that sends an unsigned 32-bit integer that is incremented by the sequencer on receipt and returned. The client then increments this indicator variable again and sends it at the time of the next heartbeat. At the default rate of heartbeat message delivery, a dChat session won't run into an overflow issue for 204 years, which seems sufficiently robust for our purposes.

## 4.4 Sequencer Election Protocol:

A sequencer election can come about in a number of ways:

1. Sequencer chooses to exit the chat
2. Sequencer node fails, causing RPC calls from clients to timeout
3. Client nodes are unable to contact the sequencer for heartbeat or message delivery

In the case of the last scenario, the sequencer responds to the message calling for an election with an assertion of its position as the leader, thereby terminating any further election messages.

The election protocol works as follows: A client node initiates an election by multicasting a message to all the other clients in its local copy of the participant list. In the message, the client suggests a username to be the next sequencer. If any of the other clients still have a heartbeat connection to the existing sequencer, they respond with the *cname* object corresponding to the sequencer, and do not acknowledge the initiator's suggestion. Otherwise, the other clients will respond either by agreeing with the sequencer choice made by the initiator or by suggesting an alternative. The initiator sends out a confirmation multicast message with the *cname* object of the new sequencer elected by a plurality of the clients. Ties are handled by arbitrarily choosing one or the other. Upon receipt of a confirmation naming a client as the new sequencer, that client updates its local copy of the participant list to reflect the previous sequencer's exit and its own new status as leader. This updated list is then broadcast to all existing participants, and the clients update their RPC connections to reflect the new IP address and port of the sequencer. All subsequent messaging proceeds as before once the new sequencer is in place.

By default, clients will always suggest the sequencer to be the earliest joining client in the participant list that is still active.

It is also possible for further clients to exit while the next servicer is being elected. In the case that an election is underway and the client receives a shutdown signal, the new sequencer will remove the terminated client from the list of active participants because no further acknowledgments or heartbeat messages will be received. If the exiting node is elected as the new sequencer, the election will continue as the client in question will no longer be active to send out the acknowledgement of its new role.

# 5 Fault Tolerance

dChat is designed to be tolerant of a number of different failure modes. The nature of the links among the clients is inherently unreliable, and so the application needs to be sensitive to and tolerant of the loss and reordering of packets. Moreover, it is impossible to tell if a client's lack of response is due to a link failure or a crash fault.

## 5.1 Client node or link failure:

If a client is not heard from via the heartbeat protocol for a period of time, the servicer will declare the client to have exited from the chat and broadcast an updated chat participant list.

A client that has been declared to have exited but is simply suffering from a lossy connection will automatically rejoin the chat when the user sends a message. The client process will issue a SEND(message) request to the sequencer, only to be told that its username is not in the list of chat participants. On receipt of this error message, the client will reissue a join request, handle the response and resend the user's message.

## 5.2 Sequencer node or link failure:

In the event of a timeout on client invocation of sequencer RPCs, the clients will retry up to 5 times before reporting failure. Failure of an RPC to the sequencer will trigger an election.

If the failure is simply the result of a temporary link failure, as described above, the sequencer will reassert itself and halt the election from taking place.