CIS 505: Software Systems
Project 3 (25% credit)
Due: Apr 24 2016 at 11:59pm (no late submissions accepted)

You can discuss high-level concepts with other teams, but your team is required to write your own code. We will be running the MOSS analyzer to detect any form of copying, including those copying from previous years. All offenders will **be referred to the Office of Student Conduct (OSC) for first time offence**, regardless of severity of violation. Offenders may be suspended for one semester by our university. In addition, if any solutions or code were obtained from a student who has taken this class previously, that student will similarly be referred to OSC and receive a similar set of points deductions, resulting in a change of grade. Students who have graduated but are found out to have supplied code or solutions to this batch of students will similarly be dealt with under our school's academic integrity guidelines. If you are unsure whether you can use an existing library or data structure, consult with the teaching staff.

1. Overview

In this *three person* group project, you will design and implement a fully distributed text-based group "chat" system under Linux. Your system will allow arbitrary size groups of Internet users to send and receive messages to the group in (approximately) real time. Users in a group can type messages at any time, and the messages typed by the various group members should be received by all members in the same order. The system will use UDP datagrams via the standard socket interface to exchange messages.

Users can either *start* a new chat group (thereby becoming the initial member) or *join* an existing chat group (by connecting to any member of a the group they want to join). Any user can leave the group they are in at any time, even if they started the group. When a member leaves, the chat continues with the remaining members (if any are left).

Your system will need to implement (on top of UDP) a *fully-ordered multicast protocol* with one group member serving as a *sequencer* (with a scheme to elect a new sequencer if the current one leaves or crashes). That is, you'll be solving many of the problems we'll be discussing in class, including error detection and leader (sequencer) election. You can use C or C++, and your system should run (and will be tested) on the *speclab* cluster.

Note that a *significant* part of this project is designing the protocol details; this project description gives the basic functionality required, but it will be up to you to decide on the specific message formats, data structures, and so on, as well as building, documenting and testing the code.

2. Functionality

The system should be implemented as a single Linux executable, "**dchat**", which will serve as the chat client for each user (both for starting new chats as well as for joining existing chats).
To start a new chat group, the user invokes the "dchat" client as:

```
$$ dchat USER
```

Where, USER is the name or nickname of the user. After starting, dchat should print out the IP address and UDP port on which it is listening. For example, if "Bob" wants to start a chat:

```
$$ dchat Bob
Bob started a new chat, listening on 192.168.5.2:7432
Succeeded, current users:
Bob 192.168.5.2.7432 (Leader)
Waiting for others to join...
```

This would indicate that the chat is now active, with Bob's client listening for messages on IP address 192.168.5.2, UDP port 7432.

To join an existing chat, the user invokes “dchat” as:

```
$$ dchat USER ADDR:PORT
```

Where (as before), USER is their own nickname and ADDR and PORT are the IP address and UDP port number of a currently chatting user (which can be *any* user in a chat, not necessarily the user who started the chat). When invoked this way, dchat should print out its own IP address and port and attempt to join the chat specified. If successful, it should print a list of the current users in the chat, with their IP addresses and port numbers. For example:

```
$$ dchat Alice 192.168.5.2:7432
Alice joining a new chat on 192.168.5.2:7432, listening on
192.168.5.81:1923
Succeeded, current users:
Bob 192.168.5.2:7432 (Leader)
Eve 192.168.8.200:8333
Alice 192.168.5.81:1923
```

If there is no active chat user listening on the IP address/port specified, dchat should simply fail and exit:

```
$$ dchat Alice 192.168.5.2:7432
Alice joining a new chat on 192.168.5.2:7432, listening on
192.168.5.81:1923
Sorry, no chat is active on 192.168.5.2:7432, try again later.
Bye.
$$
```

Once the chat starts, every user should see every message from everyone else, prefixed with their nickname, as well as status updates when events happen (such as when new users join the chat or existing member leaves the chat, a new leader being elected, etc.). For example, a typical chat dialog might look like this:

```
$$ dchat Bob
Bob started a new chat, listening on 192.168.5.2:7432
Succeeded, current users:
Bob 192.168.5.2.7432 (Leader)
Waiting for others to join...
```

```
NOTICE Eve joined on 192.168.200:8333
Hi Eve.
Bob:: Hi Eve.
Eve:: Hi Bob, I guess it's just us
NOTICE Alice joined on 192.168.5.81:1923
Hello Alice!
Bob:: Hello Alice!
Eve:: Nice to meet you Alice.
Alice:: Hi everyone
Eve:: Now there are three of us here.  I have to go soon though.
Alice:: I love distributed systems.
Eve:: Me too.  Distributed systems are the best thing in the world.
Now Eve, I think you might be exaggerating.  There's a little bit
more to life than distributed systems. Cryptography, for example.
Alice:: They sure are! My life is complete now that I've taken
CIS505.
Bob:: Now Eve, I think you might be exaggerating.  There's a little
bit more to life than distributed systems. Cryptography, for
example.
NOTICE Eve left the chat or crashed
Alice:: I guess it's just us now, Bob.
....
```

(You might notice that the messages from Bob -- "Hi Eve.", "Hello Alice!", and "Now Eve..." -- each appear twice, first when Bob typed them in as local input to the client and then again as they were output when they were actually sent to the group. Also, notice that the last message from Bob was delayed to be delivered after a message from Alice, to ensure global ordering.)

Observe that to join an existing group's chat, the user must know the IP address and port number of some current user in the group. (This is an example of a "naming" problem.) How someone finds this information is *outside* the scope of the project (that is, you don't need to provide a directory service of active chats or anything like that; you can just assume the users will find the address of a current member somehow).

The messages sent to the group should be displayed in the same order by all active users (that is, all users that the current leader considers to be in the group), but there is no need to replay previous messages to new users when they join. The chat system should deal sensibly and robustly with lost messages and with clients that crash and/or leave (e.g., the system should detect crashed or disconnected group members and remove them from the group within a reasonable time).
All chat output should be sent to stdout, and all input read from stdin. Exit the chat by indicating EOF on stdin (control-D).

3. Protocol and Implementation

The underlying network protocol you'll use will be UDP, which, you will recall, is an unreliable unicast datagram delivery protocol. Everything else must be implemented on top of that. In particular, you'll need to implement a distributed, fully ordered multicast protocol (based on a sequencer), deal with leader election, and detect and deal with failure (both of message delivery and of other clients).
Initially, the user who started the chat will serve as the sequencer, but it may be necessary to switch that role to another client if, for example, that user leaves the chat.

Note that in addition to the possibility that a UDP message might be dropped (or duplicated) by the network, users (and their clients) can come and go at any time. That is, any chat client (including that of the current leader) might simply disappear (by crashing or by exiting) at any moment. You will have to design a protocol that deals with failure gracefully, and allows the chat to continue for the remaining users.

Design and implement a fully ordered multicast with a central sequencer/leader (chosen from among the current group members). The leader will be responsible for sequencing messages as well as determining who the current active clients are, detecting eliminating any inactive/crashed users from the chat (and making sure that the remaining clients have an up to date list of those currently in the chat). Your protocol should have each client communicate with the current leader from time to time. If any client detects that the leader is not responding, they should call for an *election* of a new leader. (This means that even though clients route their messages through the leader, they still need to maintain a list of all the other current chat clients, to allow them conduct an election when required). Any deterministic scheme for choosing the election winner is fine (e.g., winner is whoever has the highest IP address / port, etc.).

A new client should be able to join a chat by connecting to *any* currently active client, even one that is not the current leader. One way to implement this is to advise the new client of the IP address and port of the current leader, if the client that was connected to is not itself the leader, and then have the new client contact the leader directly.

The design details -- message formats, data structures, protocol states, etc. -- are up to you (there is no requirement that your client inter-operate with other teams, just with your own system). The design is a very significant part of the work in this project, and it can be challenging to do this well. You are designing a fairly complex protocol, with many functions, and with many different kinds of messages that each client must be prepared to receive, parse, and act on. So don't put the design off until the last minute, and certainly think about the protocol and architecture well *before* you start writing code.

Feel free to put reasonable limits on the number of simultaneous chat users, the maximum message size, etc., but be sure to document what these limits are.

Your client should be implemented in C or C++ and run on the speclab cluster. No elaborate user interface is required (just text on stdin and stdout). It should be implemented robustly, detecting and handling gracefully unexpected events and errors.

4. Protocol Analysis

Because your system runs over UDP, it must deal with the possibility that messages will be lost, re-ordered, or duplicated by the network, and may be delivered in different orders for different clients. However, because you will be implementing and testing your system on machines connected to a local area network, these conditions will likely be relatively rare as you test your code. But your system should still handle these events robustly, and you must document the different things the network might do to each UDP datagram and the mechanisms you've implemented to recover from each kind of network error.
Your code submission must include detailed documentation describing the various messages you are sending in your protocol, the various network errors that might occur, and how you recover from them. If there are conditions you do not recover from, document those as well.

5. Extra Credit

There are several enhancements you can add to your assignment for extra credit. In all cases, if you implement an improved component, you do not need to implement the simpler version described above.

Doing extra credit is entirely optional. We offer extra-credit problems during the semester as a way of providing challenges for those students with both the time and interest to learn and pursue knowledge in a better depth. **You should only attempt the extra credits after you have completed the regular portions of the project.** We recommend that you only start working on extra credits after you have finished the regular credits.

To get full credit for the extra credits, please make sure you write suitable test cases (which can be in the form of a modified client generating a particular workload) that will help you in demonstrating the extra credit.

If you are worried that the extra credit will interfere with your regular credits, you can submit multiple versions using different team member's Pennkey.

5.1 Traffic Control (+10%)

In data communications, it is important to manage the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. In this project, you can design and implement a mechanism that controls the incoming message load at the sequencer side. First, you should have a means to detect traffic congestion, which is defined by yourself. Then, you should have methods to slow down message traffic and alleviate the congestion condition.

5.2 Fair Queuing (+10%)

When traffic is not heavy in the chat system, it is reasonable to let the sequencer order incoming messages based on a First In First Out (FIFO) queuing strategy. However, when many nodes are sending a large of number messages simultaneously and traffic gets heavy, fairness should be considered. Try to implement a Fair Queuing algorithm (e.g. Round-Robin) used by the sequencer that allows multiple group members to fairly share the capacity of the network.

5.3 Message Priority (+10%)
Priority of messages can also be considered in ordering incoming messages at the sequencer side. For example, messages regarding membership updates and election requests shall have a higher priority than the normal chatting messages, and should be broadcast immediately. Try to implement a mechanism that assigns priority to each message and order messages based on that. The details are for you to decide.

5.4 Encrypted chat messages (+5%)
You can implement a basic form of encryption-decryption e.g. the substitution cipher. Assume that the nodes already have knowledge of the key. Simple encryption schemes based on bit-shifting will not be acceptable.

5.5 Decentralized total ordering (+15%)
Instead of using the centralized sequencer, use a decentralized total ordering algorithm, using the clients as participants. You can assume that the elected leader is not used here.

5.6 Chandy/Lamport's 'snapshot' algorithm (+15%)
Implement the snapshot algorithm taught in class in which you can initiate a global snapshot from any of the members in the chat system. You are free to decide the specifications for initiating the snapshot. To fully test this algorithm, you should generate workloads that generate continuous sending/receiving of messages. The algorithm should capture correct set of messages at any client, and also in-flight messages (i.e. the channel states) capturing the differences between snapshots.

5.7 GUI (+10%)
Develop a GUI of your choice in any programming language you prefer and link your distributed chat system library to it.

6. Submissions and due dates

You'll be submitting *three* different milestones, but your grade will be based primarily on the final working code and documentation. The first two milestones are intended mainly to keep you on track, give you feedback, and give you an opportunity to correct any design problems early on. Detailed information on how to submit each milestone will be provided as the due date approaches.

Milestone 1: March 28 at 11:59pm (Monday)

Create a github/bitbucket PRIVATE repository. Add all your team members plus your assigned TA to the repository. The assigned TA will be announced on Piazza. You need to decide on a schedule of work with a rough division of labor. You should checkin a short (1 page or so) document (named 'schedule') into your repository that outlines a planned schedule. This can be a PDF, Word, or text document. We will not hold you to this preliminary plan -- you'll likely change things as you move forward, but you should have a realistic schedule that demonstrates that you've thought about what you'll be doing.

WARNING: if your repository is public, you will receive ZERO credit for milestone 1.

By this point, you should also have designed the basic protocol (messages types, data structures, protocol states), and have a detailed implementation plan that describes the various modules in your software and who is responsible for which.

For this milestone, please include also write a detailed design specification describing the protocol the clients will use (the formats of the UDP messages, when they are sent, etc). This will typically require at most of 4 pages of text. Also, you should include a detailed software design description that identifies who is doing what.

You may find you want to deviate from your original plan as you move forward in your implementation. You also may find you need to change your protocol from your original design. There is no grade penalty for changing your design or protocol as long as your original design reflected a reasonable degree of planning and effort and the reasons for your changes are sensible. Be sure to document any changes you make from the design as you proceed. Please check in this report into your private repository.

Milestone 1 is worth 7% of your project grade. No submission is required. Your assigned TA will look at your repository to do the grading after the deadline.

Milestone 2: April 11 at 11:59 PM (Monday)
For this milestone you will have to meet your group's assigned TA with your entire team to demo the following basic features:

- The chat system should be able to handle addition and deletion of the members.
- The chat system should be able to send and receive messages across the members.

You can email your TA to setup a time when you can meet him/her anytime between March 28th to April 11th.

Milestone 2 is worth 8% of your project grade.

Final Milestone: April 24 at 11:59pm (Sunday)
The final milestone is your code and documentation. You'll submit the code and documentation (describing the protocol as implemented plus a "user manual" that describes how to use your system). We have scheduled a final demonstration on **April 25** to see your team's project demo after you've submitted.
The final milestone is worth 85% of your project grade.

7. Submission guidelines (for final milestone only)

7.1 What to turn in

For this project place all required folders and files into a folder called proj3_pennkey (where pennkey is selected from any one of the team members. The submission by one of the team members will hold for the entire team. You will need all of the following items to receive full credit for this project.

• Source code: Submit all the source files. The code should be well commented.

• Project Report: The report should mention the names of the team members along with their pennkeys. It should describe the details of your final implementation (design decisions,

assumptions etc.). If you have attempted any of the extra credit sections, mention which ones. Also, include any specific instructions that may be required to run the program. Remember that if your implementation does not work during the demo, the report will play a crucial role in deciding your final grade for the project. Please do not paste your entire code in the report.

- Please include Makefiles for your submissions.

7.2 Turning it in
Use the command **"turnin –c cis505 -p proj3 <folder>"** to submit the project.

You can use **"turnin -l -c cis505"** to see whether project is current open for acceptance at any time. To make sure your submission is successful, you can use **"turnin –c cis505 –v –p proj3"** to check the status of your submissions.

If you want additional information for "turnin", there is a turnin man page which explains the usage in more detail, including some additional arguments. And the below link might help.
http://manpages.ubuntu.com/manpages/maverick/man1/turnin.1.html