cs154-2014 Project 5: Instant Message Server (p5ims)

Due date: 11:59pm Thursday June 12

You will implement an instant message (IM) server for this project that routes instant messages between connected friends. The University of Chicago Computer Science department created this project, with significant updates specifically for cs154-2014.

1 The IM System and its Terminology

We first give an overview of the main components of the IM system.

The purpose of the IM system is to allow **users** to communicate instant messages with their **friends**. Each user runs his own copy of the **client** software (which we provide to you, with source), which creates a TCP/IP connection to the single central IM server (which you implement). Clients stay connected to the server for as long as they are running. All information about the IM system that a user may send or receive is through the client's TCP/IP connection to the server.

Each user is identified by a unique **username**, which is created in the IM system by **registering** the username using the client software. Users then **login** to the server, at which point we say that the client is **bound** to the user using it: from the standpoint of the server, the client side of the TCP/IP connection is identified with the logged-in user. Users may login to the server through only one client at a time, and each running client can have at most one active user. If the running client has no active (logged in) users, it is an **unbound client**, which may be used only to register new usernames.

A user can participate in instant messaging only if he or she is logged into the IM server, that is, he or she is an **active** user. If not logged in, the user is **not active**. Two users become friends (or confirmed friends) by mutual agreement; one user sends (via the IM server) the other a **friend request**, and the request is either accepted or declined.

The IM system works because the communication is governed by a specific **protocol**. A source of potential confusion in our terminology is that while the IM server facilitates exchanging **instant messages** between friends, information in the IM system is sent between client and server in units called **protocol messages**. As the implementer of the IM server, the kind of message you will be dealing with mostly is the protocol message, so in this document we use "message" to refer to protocol messages, and will use 'instant message" to refer to the communication between friends.

Protocol messages have an in-memory representation, but they are transmitted by encoding the message as an ASCII text string that ends with a newline (`'\n'`), sending the string over the TCP/IP connection, and then parsing the string on the receiving side. The strings sent over the client-server TCP/IP connection (whether from client to server or vice versa) are called **protocol strings**. Message structure is defined in the next section in terms of symbols that we write as upper-case words like "`OP`" or "`FRIEND_NOT`", but these are not literally in the protocol strings: the symbols are encoded as *numbers* in the strings, analogous to the numeric response codes used in SMTP or HTTP (discussed in class).

In addition to providing you the IM client, we provide you an "`imp`" library with structs for representing protocol messages, functions for generating strings from messages and parsing messages from strings, and a header file that defines constants like `IMP_OP_REGISTER` and `IMP_ERROR_NOT_FRIEND`. After you read through the project description and `imp.h`, you will be able to recognize these constants as identifying elements of the protocol messages. Your server code will have to implement the server side

of the IM protocol; the `imp` library just makes some of the low-level parts easier. When the text below refers to constants like `IMP_NAME_MAXLEN` and `IMP_IM_MAXLEN`, these are defined in `imp.h`. Your code should "`#include <imp.h>`" and refer to the constants by the same names. That is, your code should work correctly even if the graders change these values in `imp.h` prior to compiling your server.

The protocol itself is defined in the next section 2, and the the `imp` library is described in Section 3. The behavior and operation of the server is described in Section 4, and its evalution in Section 5.

2 The IM Protocol

Previewing the various functions that your server will need to perform will provide some context for understanding the types of protocol messages. Your server will support: registering a new user name, letting a user login to (and logout from) the server with an IM client, creating and removing friend relationships with other users, sending instant messages to friends, and notifying users of changes in their friends' logged-in status. In response to receiving a request for some operation (like logging in or sending a friend request), your server should perform the operation if possible, and then generate a response containing an acknowledgement. Otherwise, the server should generate an appropriate error message. There are thus four types of messages: `OP`, `ACK`, `ERROR`, and `STATUS`.

2.1 User operations

The protocol has three operations for telling the server about users: register, login, and logout. For each of these operations, we list first the message that the client sends to the server, and then the various messages that your server should send to the client and to other connected clients.

2.1.1 Protocol messages for registering a user

- `OP REGISTER <userName>`: (unbound client to server) A client sends a register message to add a new user `<userName>` to the system. User names are strings of non-white-space printable ASCII characters (for every character `c`, `isspace(c)` is zero and `isprint(c)` is non-zero), with length between 1 and `IMP_NAME_MAXLEN` (inclusive). This message may only come from an unbound client.
- `ACK REGISTER <userName>`: (server to client) If user name `<userName>` is valid and not already registered, the server adds the user name to its database and sends the client this acknowledgement. The client is still unbound and `<userName>` is not yet active.
- `ERROR USER_EXISTS <userName>`: (server to client) If user name `<userName>` is already registered, the server sends this error message.
- `ERROR CLIENT_BOUND <userName>`: (server to client) The server uses this error message to say the `REGISTER` is invalid because the client is already bound to active user `<userName>`.

If the client sends a register request with an invalid user name, the server should respond with the general-purpose error message:

- `ERROR BAD_COMMAND`

There are other situations in which the server sends this error message, described below.

2.1.2 Protocol messages for login

- `OP LOGIN <userName>`: (unbound client to server) An unbound client sends this to log in with the existing `<userName>` into the server.
- `ACK LOGIN <userName>`: (server to client) If user `<userName>` exists in the server's database and is not logged in already, the server sends the client an acknowledgement. The client is then bound, and user `<userName>` is then active.
- `STATUS <friendName> <friendStatus> <IMP_ACTIVE_NOT or IMP_ACTIVE_YES>`: (server to multiple clients) In response to user `<userName>` logging in, after the server sends `ACK LOGIN`, the server then sends zero or more **friend status messages** to `<userName>`, each one describing a confirmed, pending, or requested friend `<friendName>` of `<userName>`. When user `<userName>` logs in, the server also sends friend status messages about `<userName>` to all active and confirmed friends of `<userName>` (not to pending or requested friends this time).
- `ERROR USER_DOES_NOT_EXIST <userName>`: (server to client) If the user `<userName>` in the `LOGIN` message does not exist in the server's database, the server sends this error message.
- `ERROR CLIENT_BOUND <userName>`: (server to client) The server uses this error message to say the `LOGIN` makes no sense because the client is already bound to active user `<userName>`.
- `ERROR USER_ALREADY_ACTIVE <userName>`: (server to client) The server uses this error message to say that user `<userName>` is already active: logged in through another client.

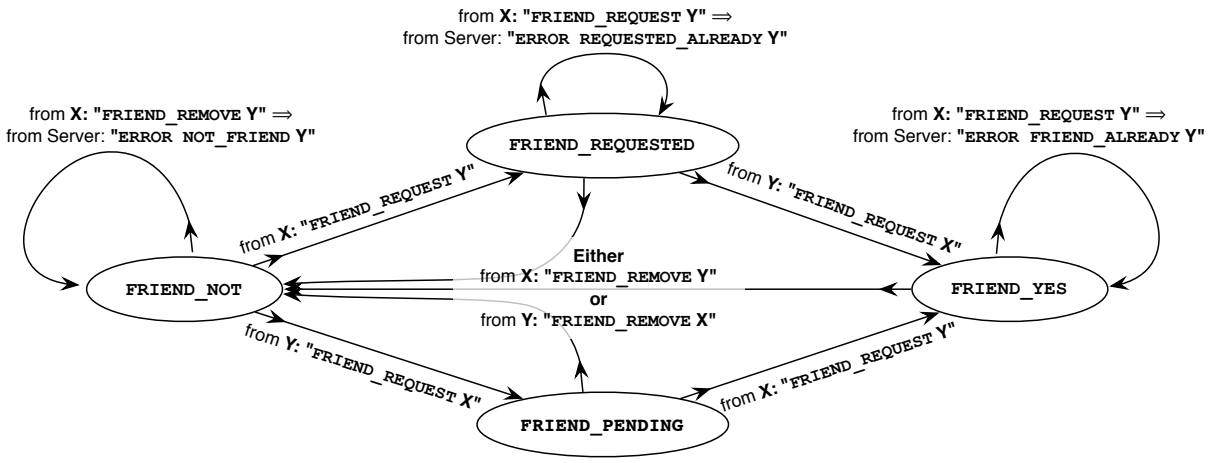
2.1.3 Protocol messages for logout

- `OP LOGOUT`: (client to server) The bound client's active user is logging out of the server.
- `ACK LOGOUT`: (server to client) The client is not bound, and the user is no longer active. The client and server remain connected, however (the same way they were when the client first started, prior to a `LOGIN`).
- `STATUS <userName> <friendStatus> <IMP_ACTIVE_NOT or IMP_ACTIVE_YES>`: (server to multiple clients) When an active user `<userName>` logs out, the server sends updated friend status messages to all active friends of `<userName>`.
- `ERROR CLIENT_NOT_BOUND`: (server to client) If the client is not bound, the server sends this message to the client to say that there is no user to logout.

In addition to `OP LOGOUT`, the server should treat a disconnect from the client (server's `read()` returns 0) as equivalent to a logout: the previously logged-in user must no longer be considered active.

2.2 Friend operations

Two users become friends by adding each other to their friend lists, via a sequence of protocol messages. Users can also remove a friend from their list. As graphical context for the following description of friend states and messages, here is a state diagram of user **X**'s friend state with respect to another user **Y**. The state transitions are labeled with protocol messages that the server receives from X or Y (or more precisely, the clients bound to users X or Y).



State diagram of **Y**'s status as a friend of **X** (these values are stored in the representation of **X** to describe **Y**). Edges between friend states are labeled with the message from the client used by either X or Y. Loops associated with error messages are labeled with message from X, and response from server.

In the IM system's representation of user X, it should record (either explicitly or implicitly) one of four possible states for a different user Y:

- `FRIEND_NOT`: X and Y are not friends and are not in the process of becoming friends. This is the initial state of all users with respect to each other.
- `FRIEND_REQUESTED`: X has requested to be friends with Y, but Y has not yet responded to X
- `FRIEND_PENDING`: Y has requested to be friends with X, but X has not yet responded to Y
- `FRIEND_YES`: X and Y are friends and may send each other instant messages

A user can not be friends with himself or herself.

Friend state is changed and communicated by the following messages:

- `OP FRIEND_REQUEST <friendName>`: (client to server) From the client's active user, who wants to become friends with `<friendName>`.
- `OP FRIEND_REMOVE <friendName>`: (client to server) From the client's active user (say, `<userName>`) who wants to cease being friends with `<friendName>`, or cancel a prior friend request from `<userName>` to `<friendName>`, or decline a pending friend request from `<friendName>` to `<userName>`.
- `STATUS <userName> <friendStatus> <IMP_ACTIVE_NOT` or `IMP_ACTIVE_YES>`: (server to client) The server sends these to update both clients involved in a friend transaction about a change in their friend state. As shown above, these messages are also sent by the server when users log in or out.
- `OP FRIEND_LIST`: (client to server) The client's active user wants to see a list of all friends and pending friends. The server responds with the same set of friend `STATUS` messages as would be generated if the user had just done a `LOGIN` (see above).
- `ERROR NOT_FRIEND <userName>`: (server to client) Server's response to a client's `FRIEND_REMOVE` request about a user `<userName>` that is not a friend or a pending friend.
- `ERROR USER_DOES_NOT_EXIST <userName>`: (server to client) Server's reponse to a client's `FRIEND_REQUEST` for a name `<userName>` that is not a registered user.
- `ERROR REQUESTED_ALREADY <friendName>`: (server to client) Server's response to a client's `FRIEND_REQUEST` when the client user already has a `FRIEND_REQUEST` pending with `<friendName>`.

- `ERROR FRIEND_ALREADY <friendName>`: (server to client) Server's response to a client's `FRIEND_REQUEST` when the client user is already friends with `<friendName>`.

If user X sends a `FRIEND_REQUEST` or `FRIEND_REMOVE` for the same user X, the server should respond with:

- `ERROR BAD_COMMAND`

In response to receiving an add or remove friend operation that is not in error, your server should make the appropriate state transition and send one friend status message to the source of the request and another to the target of the request, if the target is active. The state transition involves changing the friend state for **two** users. For example, suppose X and Y are users that are not friends. If the server receives from X a request to add friend Y, it should (1) change X's friend status for Y to `FRIEND_REQUESTED`; (2) send to X an updated friend status message for Y; (3) change user Y's friend status for X to `FRIEND_PENDING` and finally (4), if Y is active, send Y a friend status message describing X (so that Y knows there is a pending friend request to respond to).

2.3 The IM operation

Once two users are friends, they can send instant messages to each other, via the server (this is not a peer-to-peer messaging system):

- `OP IM <userName> <message>`: This message can go both from client to server and vice versa. In the first case, a client is requesting to send a message from the client's user **to** friend `<userName>`. In the second case, the server is forwarding a message **from** `<userName>` to a client's active user. Messages are strings of between 1 and `IMP_IM_MAXLEN` (inclusive) printable ASCII characters in which `' '` is the only allowed white space (`isprint(c)` must be non-zero for all characters `c`, and `isspace(c)` implies `c==' '`).
- `ACK IM <userName>`: (server to client) If the server receives a request from the active user X to send an instant message to user <userName> who is indeed a friend of X and also active, then the server should forward the instant message to <userName>, and send this acknowledgement to X.
- `ERROR USER_DOES_NOT_EXIST <userName>`: (server to client) Error message from the server if intended IM recipient `<userName>` is not a registered user.
- `ERROR NOT_FRIEND <userName>`: (server to client) Error message from the server if intended IM recipient `<userName>` is not friends with the sender.
- `ERROR USER_NOT_ACTIVE <userName>`: (server to client) Error message from the server if intended IM recipient (and friend) `<userName>` is currently not an active user.
- `ERROR IM_FAILED <userName>`: (server to client) Error message from the server in case there was some other situation, not covered above, that prevented sending the IM to <userName>.

If the server receives a friend add, friend remove, or friend instant message request from an unbound client (without an active user), the server should answer with:

- `ERROR CLIENT_NOT_BOUND`

If the client sends a message that is malformed or invalid in any way not already covered above, the server should answer with:

- `ERROR BAD_COMMAND`

A final note about the protocol strings coming from the server. An individual protocol string is terminated by the `'\n'` newline character. The server can also pack multiple protocol strings into a single string and `send` them all at once to the client; this is appropriate for the multiple friend status messages sent in response to `OP LOGIN` or `OP FRIEND_LIST`. The individual constituent protocol strings (including the last) should still have a `'\n'` terminator.

3 The `imp` protocol message library

We provide a library, called `imp`, with functions to parse protocol strings, generate protocol strings, generate protocol messages, and dispatch to (call) a function based on the protocol string or message type. Your code will send and receive protocol strings over TCP connections between the client and the server. The library also has a function for generating a human-readable version of the protocol string that can be printed for display and debugging purposes. For consistency, and as an example of good C coding practices, all the symbols in the library intended for your use begin with "`imp`" (and other symbols used internally begin with "`_imp`"). Also, all the `enum` and `#define` values defined in the `imp.h` begin with "`IMP`".

You are responsible for reading through `imp.h` carefully to understand what functions are provided by the library and how they can be used. The library uses an approximation of subtyping in C, via a set of structs that all have a "type" identifier as the initial field, followed by a `char *userName` field that may be used in all message types. The generic message struct is `impMsg`:

```
typedef struct {
  impMsgType_t mt;   /* message type */
  char *userName;    /* used for all types of messages */
} impMsg;
```

See `imp.h` for the definitions of the `impMsgOperation`, `impMsgAck`, `impMsgError`, and `impMsgStatus` structs, which you use to represent specific protocol messages, depending on the type. Pointers to the generic protocol message can be cast to/from the appropriate specific type. To convert an existing protocol (in-memory) message to a string to transmit over the network, use `impMsgToStr()`, or to create the protocol string directly (without the intermediate `impMsg`) use `impStrNew()`. To parse a given protocol string (such as that received from the server) into an `impMsg`, use `impStrToMsg()`.

Note that functions in `imp` that return `impMsg *` and `char *` (but not `const char *`) are using dynamic allocation. It is your responsibility to free that space when you are done with it, with `impMsgFree()` for `impMsg *`'s and `free()` for `char *`'s.

4 IM Server Operation

We provide (in your `p5ims` directory a reference server implementation `ims-ref`, for which the usage information is:
`./ims-ref [-h] [-v] -p port# -d dbFile [-i saveInterval]`
From the commandline you specify which "`-p`" port the server should listen on, which "`-d`" file the user database should be read from at initialization time, and the "`-i`" interval with which the user database file should be written (for simplicity, it is written whether or not are any changes). The `main.c` that we provide for you already parses these command-line options, including setting `impSaveInterval` from the "`-i`" option. The verbose debugging messages that you (may, if you want to) implement do

not have to match those produced by the reference server. The `ims` you submit should not introduce any new command-line options.

We are aware of some bugs in our "reference" implementation, and we reserve the right to update the `ims-ref` (in your individual and group repos) as we fix them. We won't expect your implementation to be any more conformant than `ims-ref` as of 72 hours before the deadline.

You test your server by using the text-based IM client `txtimc` to talk to your server, and to a reference implementation of the server that we provide, called `ims-ref`. If you have questions about subtleties of how the server will respond to some particular sequence of events, you should try it with the reference implementation and copy its behavior (though not its bugs ☺). If the reference server was started on the same machine with `./ims-ref -p 8080 -d db.txt`, and you can talk to it with `./txtimc -s localhost -p 8080`. We suggest that you open multiple shell windows, one for the server, and one each for multiple clients.

The IM server is concurrent, and accepts multiple incoming connection requests on its listening TCP port. A new thread should be generated for each connected client. Representing and updating the information about users and their friend relationships has to be done in a reliable manner, so use `pthread_mutex` functions. The server is stopped by stopping the process (e.g. `^C` at the terminal).

After its command-line options have been parsed, if the server fails to start for any reason, mathen the server should print to the standard error a descriptive error message that includes the (exact) string "`SERVER_INIT_FAIL`". For example, to handle an error in the call to `bind()`, the server could do a:
`perror("SERVER_INIT_FAIL: bind failed");`
See `man perror` in case this function is unfamiliar.

4.1 User database

Your server needs to maintain a persistent database of the registered users and their friend relationships (but not whether they are logged in). This means updating an in-memory representation as users connect and do things, as well as periodically storing the database to a file on disk (this can be the job of another thread). You should save the database to disk **every `impSaveInterval` seconds**; `impSaveInterval` is a global in the `imp` library. We are not using any established database management system; you store the information with C structs of your design in memory, and with a simple file format on disk, described next. The default `FRIEND_NOT` relationship between two users should not be explicitly represented; you represent only when users are friends (`FRIEND_YES`) or are becoming friends (`FRIEND_REQUESTED`, `FRIEND_PENDING`). The in-memory database also has to record whether each registered user is active or not (`IMP_ACTIVE_NOT` or `IMP_ACTIVE_YES`), while the database saved to file should not reflect being active.

The C library function `tempnam`, which generates a unique temporary file name, and `rename`, which can be used to change the name of a file, are useful for minimizing the chances of corrupting the on-disk database. You will want to write a function that generates the textual representation for a user's friend information, and another that parses that representation. Use these functions when you read the database on start-up, and when you periodically refresh the database.

4.2 User database file format

The provided `db-example.txt` is a human-readable example of an IM server database file:

```
4 users:
```

```
alice
- david
- bob requested
- carol
.
bob
- alice pending
- carol
.
carol
- bob
- alice
.
david
- alice
.
```

After indicating how many users there are, there is one section per user, starting with the user name, listing the friends and friends-in-progress on lines starting with "`-`", and ending the section with "`.`". Note that established friends (`FRIEND_YES`) are simply listed, while friends-in-progress are listed with their name followed by " `requested`" or " `pending`". The database records both sides of a friend relationship, even though this is redundant (this mirrors how you will likely want to record the information in memory). In the situation described by this database, `alice` has requested to be friends with `bob`, but `bob` hasn't responded yet.

The other provided database `db-empty.txt` represents having no users at start-up:

```
0 users:
```

Feel free to create you own `db.txt` files based on these examples for your testing and development.

In the database file, the top-level ordering of users, and the per-user ordering of friends, is *arbitrary*. You may choose to copy what the reference server implementation does (record users and friends in their registration order), or adopt some other ordering. Our grading will be invariant with respect to this ordering.

5 Grading

We will grade your project based on the compilation and execution of the "`ims`" program that should be built by running "`make`" in your "`p5ims`" directory. The program must compile cleanly with `-Wall -Werror`. Your server must be able to initialize itself with any database file that conforms to the format described above: we may start it with `db-empty.txt`, or `db-example.txt`, or something else entirely. Do not spend time doing careful error checking and writing error messages to describe how the database may be corrupt: you can safely assume that the database file provided will conform to the file format. We will start your server listening on a port of our choosing (specified with the `-p` option); if the server cannot initiate a listening socket on that port, you must generate an appropriate `perror` message.

Scripts for doing grading will be released shortly. You should prioritize your effort according to the following approximate allocation of points:

- **Persistent database:** 20 points
- **Register (`OP` and `ACK`) a user:** 10 points
- **Login (`OP` and `ACK`) a user:** 10 points
- **Logout (`OP` and `ACK`) a user:** 10 points
- **Add a friend:** 10 points
- **Remove a friend:** 10 points
- **Generating all appropriate friend `STATUS` messages**: 10 points
- **Send an IM (`OP` and `ACK`) to a friend:** 10 points
- **Generating `BAD_COMMAND` and other error messages**: 10 points

You can use this point allocation as a way to organize your work on building the server so that you can get some credit even if your server is not working completely. The database part is worth more than the others because it is through changes in the database file that we can confirm the registration of new users, and the formation and removal of friend relationships, even if the IM functionality is broken.

As with previous projects, **if your code does not compile, then there is nothing for us to grade**, so you get a zero.