

Programming Assignment 3: Instant Messenger

CSE 220: Systems Programming

Introduction

In this assignment, you will implement the packet encoding and decoding for a basic instant messaging protocol. You will be using void pointers and pointer arithmetic, along with functions from `string.h` to accomplish this. You will learn about pointer arithmetic, raw memory access, and data serialization. This assignment only requires you to encode and decode the outgoing and incoming packets, the rest of the instant messaging application is already implemented for you.

Academic Integrity

As detailed in the course syllabus, academic integrity is important for the value and durability of your degree from the University at Buffalo. Here are a few reminders to help you maintain the integrity of this assignment.

- It is a violation of the course academic integrity policy to share this assignment document or details about this assignment with any student at UB not enrolled in CSE 220 during this academic semester, with any student at any other institution, or with anyone else without permission from your instructor. This includes homework- and note-sharing Internet sites such as Course Hero and Chegg.
- It is a violation of the course academic integrity policy to share *any code* from this assignment with any student at any time during or after this semester. You may discuss your code with course staff if necessary.
- It is a violation of the course academic integrity policy to seek assistance from other students or *any online resource not specifically approved by your instructor*. Forbidden resources include: Stack Overflow/Stack Exchange, GitHub/GitLab/BitBucket, students who have previously taken this course, *etc.*
- It is a violation of the course academic integrity policy to discuss implementation details with anyone except course staff.

Students found violating any of these policies, or any other academic integrity policy in the syllabus, will be sanctioned. Sanctions may include failure in the course or expulsion from the University. Make sure that you are familiar with the course academic integrity policies, as well as the policies of the department and University.

1 Getting Started

You should have received a GitHub Classroom invitation for this project. Follow it and check out the resulting repository.

As always, read over the entirety of this handout before starting the assignment. If you have not yet read Chapter 5 of *The C Programming Language* by Kernighan & Ritchie, you should also read that before starting. You will specifically find Sections 5.3, 5.4, and 5.5 to be very helpful.

You should read and understand all of the code in the `src` directory. You are *not* expected to read or understand any code in the `client` directory, although you may look at it if you wish.

Man page sections are noted as a number in square brackets after a keyword in this document. For example, `memset [3]` indicates that the manual page for the `memset()` function is found in section 3 of the Unix manual, and you can view it with the command `man 3 memset`.

2 Requirements

In this assignment, you must implement encoding and decoding of packets of data according to the formats specified below. Each packet is stored internally as a `void *`, according to a standard format as specified in Section 3.

Note that the formats in this assignment are **very specific**, and must be implemented precisely. This includes details such as white space and the value of padding bytes. Be sure not to include any extra characters in quoted strings, insert extra newlines or other formatting, or deviate from this specification in any way!

The standard defines several packet types:

- **STATUS**
This packet type sets your status on the server. It should be sent when the user input starts with `"/me"` followed by a space. Any input starting with `"/me"` followed by any other character is invalid.
- **LABELED**
This packet type tags another user. It is sent when the user input starts with `"@"` followed by at least one and no more than `NAME_SIZE` non-space characters. `NAME_SIZE` is defined in `src/serialize.h`.
- **STATISTICS**
This packet type is used to request and receive a set of basic statistics from the server. It is sent when the user input starts with `"/stats"` followed by the end of the input. Any input beginning with `"/stats"` followed by any other character is invalid.
- **MESSAGE**
This packet type is the most basic, it is just a normal plaintext message. It should be sent if the user input does not fall into any of the above categories.

The standard also defines a special packet:

- **REFRESH**
This packet is sent to the server by the client to request new messages to decode. It is sent every second and has no relation to user input.

Through this assignment, you will implement four functions:

- `int pack(void *packed, char *input):`
Parse input to determine the packet type and fill `packed` with the encoded input.
- `int pack_refresh(void *packed, int message_id):`
Fill `packed` with an encoded refresh packet according to the specification in Section 3.5.
- `int unpack(char *message, void *packed):`
Decode `packed` based on the packet type and fill `message` with the decoded string.
- `int unpack_statistics(struct statistics *statistics, void *packed):`
Decode `packed` according to the STATISTICS format specified in Section 3.4, filling `statistics` with the data.

Any input beginning with `"/"` that does not include one of the commands defined above is invalid, *but will never be tested*. You may use this to implement commands starting with slash for debugging if it is useful to you.

For all functions, the return value should be the integer value of the packet type as defined in `serialize.h`, or `-1` for invalid inputs. You should assume that any non-NULL pointer passed to a function is correctly allocated and of an adequate size. Other than that, you should not make any assumptions about the validity of inputs; you are responsible for validating them. This includes unreasonable or meaningless user input as well as malformed data to be unpacked. Some examples of invalid values will be described in Section 3.

3 Packet Formats

This section will describe the format of the different packets in memory. The packets are generally in the form described in the table below.

int	char[NAME_SIZE]	size_t[]	size_t	char[]
Packet Type	UBIT Name	Data Lengths	0	Data

The first half of the packet is constant among the different packet types. It starts with an integer storing a value associated with the packet type. This would be one of the packet types described above (REFRESH, MESSAGE, etc.), as defined in `serialize.h`. Following this is a character array of *exactly* NAME_SIZE bytes, which is also defined in `serialize.h`. This character array is valid if and only if it contains the sender's UBIT Name followed by ASCII NUL bytes.

The second half of the packet describes the actual data of the message. The "Data Lengths" field is an array of `size_t`, which varies in length based on the packet type. This list will always have a 0 value terminating it. For each non-zero value in this list, there is an associated string of characters in the "Data" field. To give a simple example, if the Data Lengths field held the array {2, 5}, and the Data field held the string "hihello", that would signify that there are two strings in the data of sizes two and five respectively, "hi" and "hello". The sum of all data lengths must not exceed MAX_MESSAGE_SIZE as defined in `src/serialize.h`. Note also that, due to ambiguity with the 0 terminating length value, *no valid packet can contain a field of length 0*.

Each of the specific packet formats follows. All of the specific formats other than the STATISTICS and REFRESH types are implementations of the general format described above.

3.1 Message

int	char[NAME_SIZE]	size_t	size_t	char[]
MESSAGE	UBIT Name	Message Length	0	Message

The message packet is the simplest data-carrying packet, holding only a single data field and its length. The data should be the entirety of the message input by the user *including any leading or trailing whitespace*. So if I were to input "My name is Peter" the encoded message should look like the following:

int	char[NAME_SIZE]	size_t	size_t	char[]
MESSAGE	"pagottes"	16	0	"My name is Peter"

Note that every character after the "s" in the UBIT Name field should be a NUL character, which you should remember has a integer representation of zero.

A message containing *only* ASCII space characters should not be encoded, and the pack function should instead return invalid.

When decoding this packet, the resulting string should be of the form "UBIT Name: Message", so the above packet would decode to "pagottes: My name is Peter". Note that there is both a colon and a space between the UBIT Name and the Message.

3.2 Status

int	char[NAME_SIZE]	size_t	size_t	char[]
STATUS	UBIT Name	Status Length	0	Status

This packet type is very similar to that of the message type; however, the data it contains differs. As previously mentioned, the status packet type is sent when the user input begins with `/me` followed by *one or more* space characters. When you encode this input you should only send the status itself, not the `/me` or **any** of the spaces between it and the next non-space character. If the user input was `/me says hi`, the data encoded should be simply `says hi`; this would be the same for a dozen spaces of separation as a single space. Like the message packet type, a status of only spaces should be considered invalid.

A status should be decoded nearly the same as a message, but it should not have a colon, so it would be "UBIT Name Status". If I were to send a status resulting from the user input `/me says hi`, it would be decoded by a client as `pagottes says hi`.

3.3 Labeled

int	char[NAME_SIZE]	size_t	size_t	size_t	char[]	char[]
LABELED	UBIT Name	Message Length	Target Length	0	Message	Target

The labeled packet is the most complex type to pack. You will recall that this packet type is sent when the user input begins with an "@", followed by one to no more than NAME_SIZE non-space characters. This string between the "@" and the first space is the "Target". Note that this string does not include the "@". The string starting from the first non-space character after the target is the "Message". The same rules on space characters from the previous subsection apply. Any input starting with "@" that does not match the input format for a labeled message is an invalid input.

If the user pagottes input the text "@elb the Offspring is basically ska but good"¹, it would be encoded as:

1. The integer LABELED
2. The characters in "pagottes" padded out to NAME_SIZE bytes
3. A size_t containing 39 (the number of characters in "the Offspring is basically ska but good")
4. A size_t containing 3 (the number of characters in "elb")
5. A size_t containing 0
6. The 39 characters of the message
7. The 3 characters of the target

When decoding, the string is expected to be of the form "UBIT Name: @Target Message". In the above example, it would be "pagottes: @elb the Offspring is basically ska but good".

3.4 Statistics

int	char[NAME_SIZE]	char[NAME_SIZE]	int	long	long	int
STATISTICS	UBIT Name	Most Active	Most Active Count	Invalid Count	Refresh Count	Message Count

You should immediately notice that the statistics packet does not conform to the general format mentioned at the beginning of the section. Instead, it stores data corresponding to the statistics struct defined in `serialize.h`:

```
struct statistics {
    char sender[NAME_SIZE+1]; /* Name of sender */
    int messages_count; /* Number of packets sent to the server */

    char most_active[NAME_SIZE+1]; /* User who has sent the most messages */
    int most_active_count; /* Number of messages sent by that user */

    long invalid_count; /* Number of invalid packets sent to the server */
    long refresh_count; /* Number of refresh packets sent to the server */
};
```

When encoding the statistics packet, none of this information is available to you, so you should only populate the packet type and your own UBIT Name. When decoding, you will decode this information into a statistics structure handed to the `unpack_statistics` function. The given client code will handle formatting the output printed to the screen.

¹Peter's musical discernment is known to be suspect. The Offspring is good, and they have some excellent ska tunes, but ska is king.

3.5 Refresh

<code>int</code>	<code>char[NAME_SIZE]</code>	<code>int</code>
REFRESH	UBIT Name	Last Message

The refresh packet is another packet which does not conform to the general case. It simply stores the packet type, your UBIT, and the ID of the last packet you received. This packet type is also special in the fact that you will only have to encode it, as it is never sent from the server to the client.

Because this packet type is used in the process of communication with the server, `pack_refresh()` is the first thing you should implement. If this function does not work correctly, you will not receive any messages.

4 Using the Client

When you start the chat client, you will be presented with a full-screen terminal application having a box at the bottom for user input. The chat client frequently calls `pack_refresh()` to send a REFRESH packet to the server to fetch incoming messages, which will be passed to `unpack()` to be displayed by your client. It will also accept user input and send that input to `pack()` when the user presses the Enter key.

To exit the chat client, press Control-C.

5 Testing

You are not given any direct tests for this assignment. Instead, you have the above specification and have been provided with a functional server to use for testing. You should open two terminals, starting the server in one and the client in another. To start the server you will run `./chat_server server_sock`. For the client, `./chat server_sock`. The argument `server_sock` should be a path to *the same file*, which will be *created by the server*. Using a file that already exists will probably result in an error. You should send at least one packet of every type from the client, and ensure that they are in turn displayed correctly on your client. You should avoid sending messages too quickly, however, as the server enforces a rate limit of 5 messages per second. If you exceed the rate limit you will be locked out for 5 minutes, although when you are running the server locally you may just restart it.

You will find that this may be somewhat difficult to debug, as it can be hard to tell if the error is occurring in your encoding or decoding functions. For this reason, you will find `gdb` very useful for debugging. Of specific interest to you will be the `x` command. This command dumps memory at a pointer, as described by the `gdb` help message:

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
    and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format. If a negative number is specified, memory is
examined backward from the address.
```

Defaults for format and size letters are those previously used.
Default count is 1. Default address is following last thing printed
with this command or "print".

You might use `gdb` to break at the end of your `pack` function and dump several bytes of memory using `x/20xb packed`, for example.

Because the chat client uses the entire terminal window, running gdb and the chat client in the same window is inconvenient. Fortunately, gdb has the capability of connecting to another process on the system. You can do this by running `gdb filename PID`, where `filename` is the name of the binary to debug, and `PID` is the process ID of the running process. See the gdb help, [ps \[1\]](#), and the gdb resources provided for the gdb lab for more information.

Once you have a working implementation you will be able to connect to a class-wide server on `stilben.cse.buffalo.edu`. This machine is reachable only from on-campus IP addresses. You can connect to it by first connecting to an on-campus machine such as `timberlake.cse.buffalo.edu`, or by using the UB VPN software. You may find this server helpful for finding edge cases you had not previously considered. To connect to the server, push your code to GitHub and do the following:

- In a terminal, connect to the machine `stilben` by running `ssh UBIT@stilben.cse.buffalo.edu`, where `UBIT` is replaced with your `ubit` name. Your credentials are your normal `UBIT` credentials. Remember that you may have to start your VPN or first `ssh` to an on-campus machine.
- Open your `pa3` GitHub repository on GitHub, and copy the URL from “Clone or download”.
- In your terminal, run `git clone URL` where `URL` is what you copied in the previous step, and enter your GitHub username and password if necessary. (You may need to set up SSH keys on `stilben`, particularly if you have two-factor authentication enabled on your GitHub account.)
- `cd` into your cloned repository and run `make`.
- You can now invoke the client with `./chat /home/chaterv/server/socket /home/chaterv/client`. There is already a running server, so you should not start one.

Please note that the shared server will be monitored and logged, and you should absolutely not: behave inappropriately, use foul or offensive language, harass or disrespect anyone, or discuss any implementation details on it. This also applies any messages you send when you are locked out due to exceeding the rate limit.

6 Guidance

When considering how you are going to implement this assignment, I *highly* recommend reading over Chapter 5 of K&R. You will also want to read the `string [3]` man page, which contains a description of all functions defined in `string.h`.

You will almost certainly find it helpful to draw out a few encoded packets of each type, with the appropriate values and padding in the appropriate places, and think about how you will implement code to produce that output. Consider what input the user would type to produce the packet.

Learn how to attach gdb to your client process sooner rather than later. You will certainly find it helpful.

7 Grading

This assignment is worth 5% of your course grade. Points for this project will be assigned as follows; 10% of the assignment score for the handout quiz, and the remainder from this breakdown:

Points	Description
2	Handout quiz (will not be reflected on Autograder scores)
8	Packets are correctly encoded
6	Packets are correctly decoded
2	Inputs are correctly validated in <code>pack.c</code>
2	Inputs are correctly validated in <code>unpack.c</code>