Project 1

October 9, 2014

Spec version: 0.1
Due Date: Wednesday, November 5th, 2014

1 Introduction

The sliding window protocol (SWP) is one of the most well-known algorithms in computer networking. SWP is used to ensure the reliable, in-order delivery of packets across unreliable links.

For your first project, you will be tasked with implementing a version of SWP for communication between threads. There are two types of threads you are responsible for implementing: *senders* and *receivers*.

Senders must transmit messages typed in at the command line to a corresponding receiver. Messages can and will be either dropped or corrupted in flight. You are responsible for ensuring that messages eventually reach their destination.

1.1 General Instructions

1. All code must be written in C/C++.
2. All submitted code should be accompanied by a brief design document, describing, at high level, how your code works, and what each major function does.
3. Please make sure your code is clean, well-formatted, and well-documented.
4. Your code must compile and run on department machines. We cannot grade programs that work only on Windows.

5. Please do not hesitate to contact the TAs or post questions to Piazza.

1.2   Submission Instructions

1. Log in to a campus machine
2. `cd` into the directory containing your code.
3. Run: `make submit`.
   This will tar up everything in the directory, create an archive file called `project1.tgz`, and run `turnin` on that file.
4. Please see Lab Instructions Manual for specific details.

1.3   Late Policy

You will lose a grade for every day the project is late. Assuming the project is out of 100 points (this is subject to change), if you submit the project one day late (meaning anywhere from 1 second to 23 hours, 59 minutes, and 59 seconds after the deadline), the highest grade you can get is an 89, two days late a 79, and three days late a 69. Any project submitted after 3 days will receive 0 credit.

2   Project Specifications

As stated in the introduction, the objective of this project is to ensure the reliable, in-order delivery of messages between threads. The communication channel between the threads will be lossy and unreliable, meaning that messages will be dropped or corrupted (e.g. bits will be flipped at random). To overcome this, you will implement a version of the sliding window protocol (SWP). Please read the sections 2.4 and 2.5 in P&D for further details regarding the algorithm.

2.1   Getting Started

The goal of this project is to guarantee the reliable, in-order deliver of messages between the *sender* and *receiver* threads. The sender will take messages from `stdin` (that the user has typed in at the command line), then direct them to the appropriate receiver. The sender will call `send_msg_to_receivers`, which will broadcast the message to **ALL** the receiving threads. The receiver merely needs to output the messages to `stdout` via `printf`.

Here is a brief overview of how to get started using the skeleton code.

run_stdinthread

Input Thread

run_sender

ll_input_head

Input Cmd Queue

ll_inframe_head

Input Msg Queue

Sender Thread(s)

ll_inframe_head

Input Msg Queue

Receiver Thread(s)
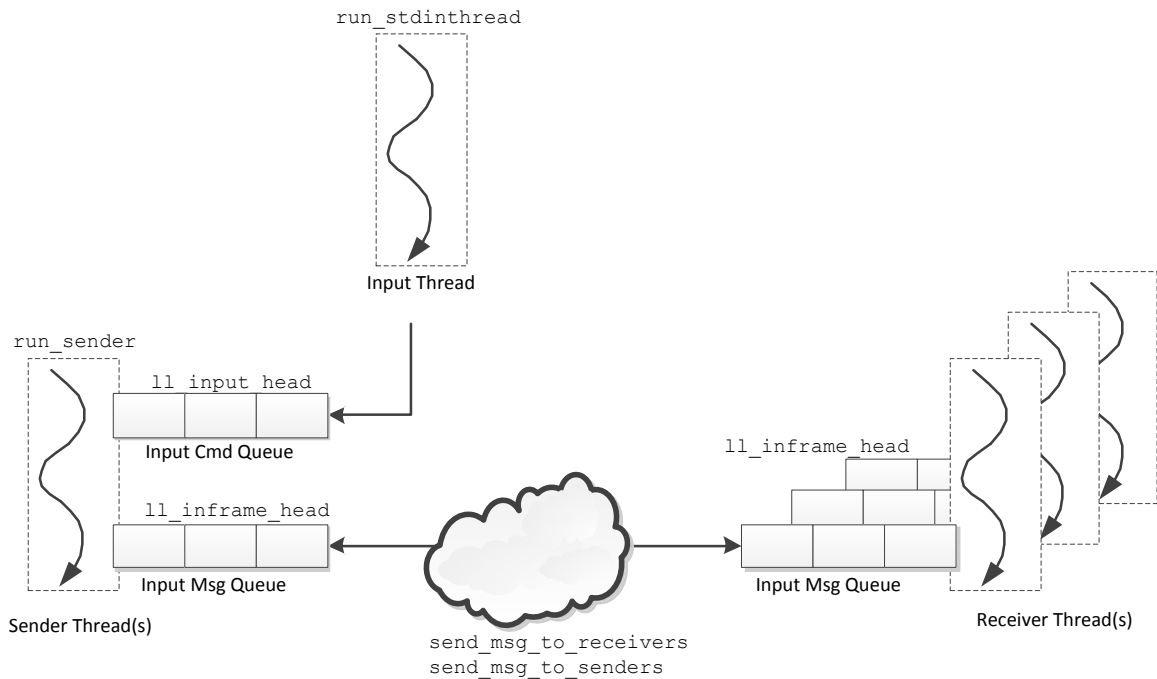
send_msg_to_receivers
send_msg_to_senders

Figure 1: Diagram illustrating how the stdin, sender, and receiver threads interact.

1. Use the prep command to begin working on cs123f assignments: `prep cs123f`

2. Please copy over the tar file into your directory and untar it:
   `cp ../public/projects/project1.tgz .; tar xzvf project1.tgz`

3. Please compile the skeleton code by typing: `make`
   You should now have an `tritontalk` binary in the same folder as the skeleton code.

4. To see the full list of command line options and a corresponding explanation, please type:
   `>./tritontalk -h`

5. You can start up the skeleton code with the following command:
   `>./tritontalk -s 1 -r 2`
   The command line options `-s` and `-r` specify the number of sender and receiver threads to run concurrently.

6. You should see the following:
   ```
   Messages will be dropped with probability=0.000000
   Messages will be corrupted with probability=0.000000
   ```

```
Available sender id(s):
send_id=0
Available receiver id(s):
recv_id=0
recv_id=1
```
7. You can now send messages between the sender and receiver threads. Type in the following command:
   ```
   msg 0 1 hello world
   ```
   This command intuitively says: have the sender with id 0 send the "hello world" message to the receiver with id 1.
8. You should see the message printed to the screen. However, note that both receiver threads print this message.
   ```
   <RECV_0>:[hello world]
   <RECV_1>:[hello world]
   ```
   You need to make sure that **ONLY** `RECV_1` prints the messages.

2.2   Breakdown of Included Files

The provided skeleton code consists of the following files:

1. `main.c`: Responsible for handling command line options and initializing data structures
2. `common.h`: Houses commonly used data structures among the various source files.
3. `communicate.c`: Takes care of transporting messages between the sender and receiver threads.
4. `input.c`: Responsible for handling messages inputted by the user (e.g. `msg 0 0 hello world`).
5. `util.c`: Contains utility functions, namely, all of those for the provided linked list implementation.
6. `sender.c`: Contains the skeleton code for the sender threads.
7. `receiver.c`: Contains the skeleton code for the receiver threads.

You are responsible for modifying the `sender.c` and `receiver.c` files to incorporate in concepts from the sliding window protocol that handle the lossy, unreliable links. You may modify *any* of the other files above, and add any additional files as necessary (also taking care to change the `Makefile`). **HOW-EVER**, we will be overwriting the `input.h`, `input.c`, `communicate.h`,

and `communicate.c` files after you have submitted your project.

2.3   Frame and Behavioral Specifications

1. The `char *` buffers communicated via `send_msg_to_receivers` and `send_msg_to_senders` should be, at most, 64 bytes. Please refer to `MAX_FRAME_SIZE` in `common.h`. For example, suppose that, in your frame specification, you use 16 bytes for the header on each frame. This leaves you with a payload size of 48 bytes, meaning that you can only transmit 48 characters worth of inputted text per frame.

2. You should **NOT** use more than a 8 bits (`unsigned char`) for the sequence/ack numbers inside your encapsulating frame. In your design document, please detail the fields (that is, list the field names and data types) used within your frames. We **WILL** be testing sequence/ack number wrap around, meaning that when either value reaches 255, your sequence/ack numbers **SHOULD** wrap back around to 0, and your SWP implementation **SHOULD** continue to function correctly.

3. If a message is lost in transmit, your senders/receivers should retransmit it within 0.10 seconds.

4. If a message is corrupted in transmit, your senders/receivers should retransmit it within 0.10 seconds.

5. A receiver cannot buffer more than 8 messages from a *particular* sender. For example, if you type in a string at the command line that is more than 64 bytes (see above restriction on `MAX_FRAME_SIZE`) * 8, your sender should send up to 8 messages, then queue the remaining bytes. The corresponding receiver should acknowledge having received some subset of the 8 messages before the sender can continue sending.

6. We will not be strict about checking to see whether you free dynamically allocated memory (via `malloc` or `new`), but please, do your best to call `free` when appropriate. Memory leaks are bad, and this is good practice for when you finally enter the real world.

7. To reiterate, we will be overwriting all `input.*` and `communicate.*` files.

8. Please send any debugging output to standard error. Substituting `printf("msg", ...)` with `fprintf(stderr, "msg", ...)` should send the output of this method call to stderr.

2.4 Tasks/Hints

The following are suggestions for implementing the project.

1. **Create a Frame Format:** You should encapsulate all messages communicated between the senders and receivers in some type of frame. Think about what attributes should be included in the frame to meet the goals of the project. For example, receivers should be able to tell when a message is intended for them. Remember, any communication between senders and receivers is broadcast based, meaning that, for example, when a sender sends a message, *all* receivers get a copy of the message. Again, we cannot emphasize enough, read the SWP section in P&D.

2. **Receiver Acknowledgments:** Implement acknowledgments, That is, when a message arrives, the intended receiver should respond to the sender that it has received the corresponding message.

3. **Retransmitting Messages:** The next task you should look into is retransmitting lost packets. Now that acknowledgments have been implemented, your senders should be able to detect that a message has been lost.

4. **Sequence numbers:** The next task you should look into is adding sequence numbers to the frames. This will allow you to have multiple outstanding frames, and also allow the receiver to acknowledge having received specific frames.

5. **Implement SWP:** At this point, this should be fairly straightforward to implement. Again, please read section 2.5 in P&D. Note that the SWP implementation described in the book only covers a single receiver and sender pair. One way to emulate this is to start up `tritontalk` with only 1 receiver and 1 sender. We recommend taking this approach first, to ensure that your SWP implementation is well tested before moving on. **However**, each of your senders will need to be able to send messages to each of the other receivers. You can try tackling this requirement first, but it will make testing more difficult.

6. **Implement Message Partitioning:** Recall that any transmitted frame cannot consist of more than 64 bytes. When a string is typed at the command line with more than 64 characters, your senders should partition the input string into multiple frames.

7. **Sender/Receiver Buffering:** Recall from the overall project specification that no sender or receiver should buffer more than 8 messages. You should make sure that if a user types an incredibly long input string at the command

line, that the sender takes the necessary steps to not overrun the receiver (and vice versa), while at the same time keeping the communication channel well utilized (that is, allowing for up to 8 in-flight messages between each sender/receiver pair).

8. **Detecting Corrupted Packets:** The next task you should tackle is detecting when a message is corrupted. You should be familiar with methods for discerning corrupted packets. If not, reread Chapter 2 in P&D.

9. **(Optional, for extra 10% credits) Implement Messaging Multiple Receivers:** You might have already done this, but your senders should have the ability to maintain SWP state on a per receiver basis. That is, if more than 1 receiver is present in the system, each sender should be able to send a message to each of the receivers (meaning that, for every sender, you will have to maintain state for **each** receiver). Similarly, receivers should maintain state for each sender.