

Project 1b: Sliding Window Protocol and Error Detection

Assigned: 2022-01-17 8:00 AM Pacific Time

Due: 2022-01-31, 10:00 PM Pacific time

Overview

This project builds on Project 1a. You will add an error detection mechanism to detect corrupted frames, and implement the sliding window protocol to improve performance. To make this document easier to read, the sections that have changed for Project 1b are shown in bold.

Sender hosts must transmit messages typed in at the command line to a corresponding receiver host. Messages can and will be dropped **or corrupted** in flight. You are responsible for designing and implementing a protocol to ensure that messages eventually reach their destination host.

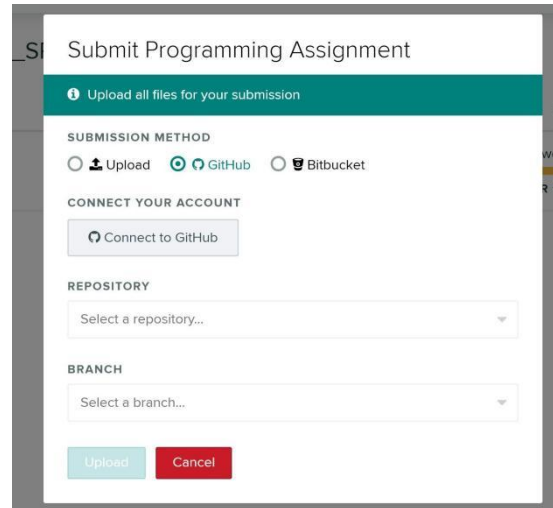
General Instructions

1. All code must be written in C or C++.
2. All submitted code should be accompanied by a brief design document in the README file describing, at a 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.

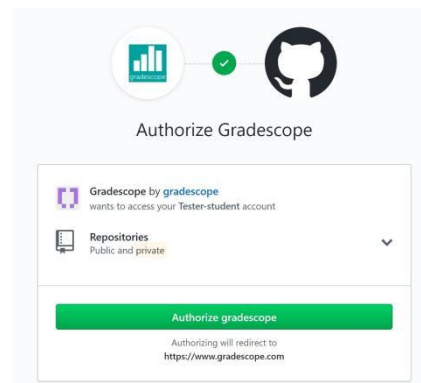
Submission Instructions

1. **Use the same GitHub repository as you used for Project 1a.**
2. Please add your name and PID into the README file.
3. Once you have modified the files, commit your changes with your completed solution
 - a. `git add *`
 - b. `git commit -m "Your commit message"`
4. Push/sync the changes up to GitHub
 - a. `git push origin master`
5. Submit your repository to the Gradescope website
 - a. Submit through GitHub:
 - i. Push your completed code to your GitHub repository

- ii. On Gradescope, choose “GitHub” as submission method



- iii. Authorize Gradescope to access your GitHub repository by clicking “Connect to GitHub”



- iv. Repository menu: select your project 1a repository
- v. Branch menu: select the `master` branch (unless you are certain that you pushed your code to some other branch you created)

Project Specifications

As stated in the introduction, the objective of this project is to design and implement a protocol to ensure the reliable, in-order delivery of messages between hosts (threads). The communication channel between the hosts (threads) will be **lossy and unreliable**, meaning that some messages will be dropped **or corrupted (e.g. bits will be flipped at random)**. To overcome this, you will implement acknowledgements (to detect if messages are received correctly) and retransmission (to resend messages that are lost).

Getting started

The goal of this project is to guarantee the reliable, in-order delivery 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:

1. **Use the same GitHub repository as you used for Project 1a.**

2. Compile the code by typing: `make`

You should now have a `tritontalk` binary in the same folder as the skeleton code.

3. To see the full list of command line options and a corresponding explanation, please type:

```
./tritontalk -h
```

4. 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.

5. 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
```

6. 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.

7. You should see the message printed to the screen. However, note that both receiver threads print this message.

```
<RCV_0>:[hello world]
```

```
<RCV_1>:[hello world]
```

You need to make sure that ONLY `RCV_1` prints the messages.

By default, no messages will be dropped **or corrupted**. You can enable message dropping using the `-d` option to specify the probability that a message will be dropped, **or enable message corruption using the `-c` option**.

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 framing, **error detection**, and retransmission to 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`). However, we will be overwriting the `input.h`, `input.c`, `communicate.h`, and `communicate.c` files after you have submitted your project.

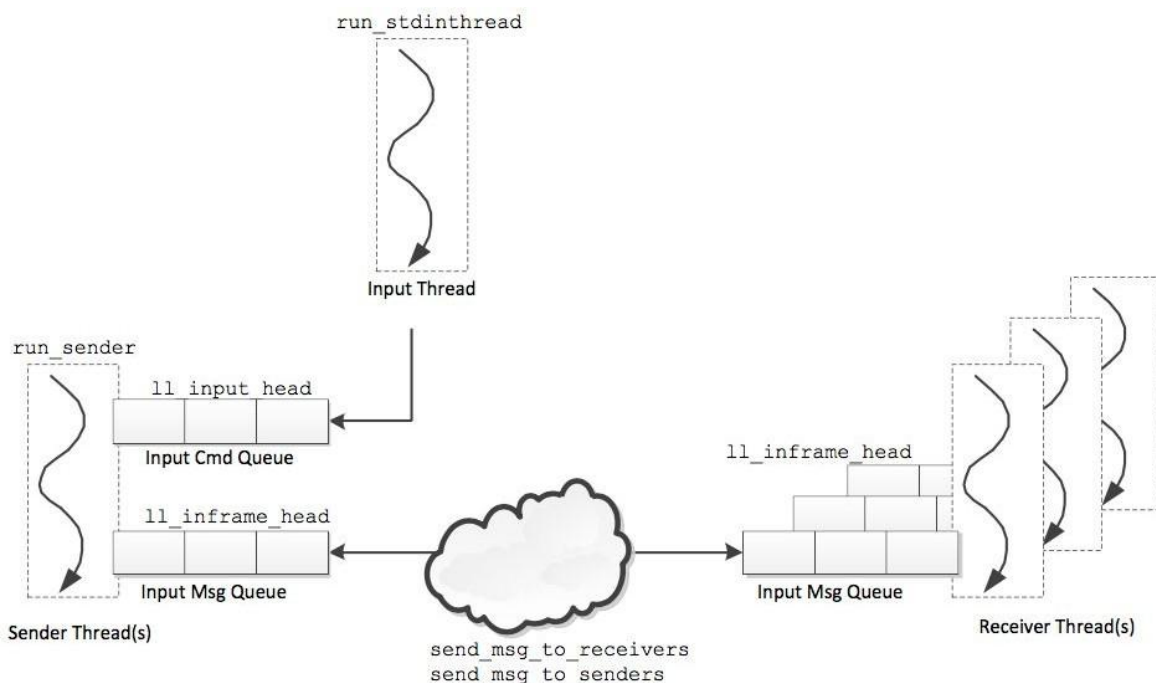


Figure 1: A diagram illustrating how the stdin, sender, and receiver threads interact

Tasks

The following are suggestions for implementing the sliding window protocol:

1. Divide the messages into frames: You should divide all messages communicated between the senders and receivers with some type of framing. This will make it possible to detect errors in frame-sized portions of the message, and retransmit the frames that do contain errors.
2. Create a frame header format: Think about what attributes should be included in the frame header to meet the goals of the protocol. 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. This means that when a sender sends a message, all receivers get a copy of the message.
3. Implement Receiver Acknowledgments: When a message arrives, the intended receiver should respond to the sender that it has received the corresponding message.
4. Implement message retransmission: After acknowledgments have been implemented, your senders should be able to detect that a message has been lost. The next task is you should add the functionality of retransmitting frames that have been lost. To test retransmission, you should use the `-d` option when running `tritontalk` to specify the probability that messages will be dropped. For example, running `./tritontalk -s 1 -r 2 -d 0.3` will cause 30% of messages to be dropped. Your senders should retransmit the dropped messages.
5. **Add sequence numbers: Frames should have sequence numbers. This will allow you to have multiple outstanding frames, and also allow the receiver to acknowledge having received specific frames.**
6. **Implement sliding window protocol: At this point, this should be fairly straightforward to implement. 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. Like in project 1a, a sender may communicate with only one receiver at a time, but a receiver must be able to handle frames from multiple senders at the same time.**
7. 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 characters than can fit in one frame, your senders should partition the input string into multiple frames.
The receiver should reassemble the input string and print it out all at once after all the

frames have arrived.

8. **Sender/Receiver Buffering:** No sender or receiver should send more than 8 frames without receiving a reply. You should make sure that if a user types a long message at the command line, 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 frames between each sender/receiver pair).
9. **Detecting Corrupted Frames:** Your code should detect and ignore corrupted frames. Chapter 2 in P&D has details on how to detect corruption.

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 text per frame.
2. You should use an 8-bit unsigned integer (`uint8_t`) for the sequence/ack numbers in your frame header. 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 after either value reaches 255, your sequence/ack numbers SHOULD wrap back around to 0, and your implementation SHOULD continue to function correctly.
3. If a message is lost or corrupted in transit, your senders/receivers should retransmit it after waiting more than 0.085 seconds but less than 0.1 seconds. We recommend using a nominal timeout duration of 0.09 seconds.
4. 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 too large to fit into 8 messages, 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.
5. A sender may finish handling one command from the input thread (sending all frames to the appropriate receiver and getting all expected acknowledgments) before starting on the next command.
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.