

CS485/ECE440: Project

Rolf Riesen

December 2, 2008

1 General

- **Follow the instructions below very carefully!** Name the files exactly as spelled out in the text below, follow the rules on file formats, etc. With this many students, we need to automate the process of grading as much as possible. For every minute we spend on each homework, we lose an hour! Please be considerate.
- This project is **due December 11, 2008**.
- Submit via e-mail to `riesen@cs.unm.edu` (Mail it before class on the 11th.) Your subject line must say: “Project Submission” (and nothing else).
- Keep your answers (and programs) succinct.
- Obey the University rules on plagiarism. In particular, do use libraries and the web to find information you need to answer the questions, but do not copy whole answers or programs. Reference your sources. The work you turn in must be *your* work.
- This is not a team project. You may discuss your ideas with other students, but the code must be designed and written by yourself.
- Your programs must be written in C or C++.
- Name your files as specified in the exercise below.
- Your programs must compile without warnings under gcc with the flag `-Wall` set.
- Provide a Makefile that compiles your programs assuming gcc and the gnu libraries are installed on the system.
- Create a compressed tar archive with the Makefile and the source code only (no `.o` files, executables, or READMEs).
- Name the tar archive `Project.tar.gz` The archive must contain a Makefile, a file named `miami.c`, and a description of your implementation in a PDF file named `Project.pdf`. There should be no other files.
- Make sure your full name appears in the body of your message. Some email aliases do not easily translate to your name in Loboweb.

- Do not put any requests or messages into the body of your email. It is quite possible that we will never look at the body of your message; just the attachments. If you have questions or comments, send them in a separate email with a subject line other than “Project Submission”
- Your program code must be clean and readable. Be consistent with you indentation, use descriptive names for your variables and constants, etc. Make sure the code you send is plain ASCII text and formatted the way you intended. This is especially true if you are using an IDE which often produce messy source files. Use a plain text editor to look at your files. Poorly written code, even if it works, will loose some points.

2 Overview

Your task for this project to implement a transport layer protocol called MIAMI. In particular, you are given an API for a asynchronous message passing protocol that you must implement on top of UDP. MIAMI makes some guarantees that UDP alone cannot provide. It is your task to implement those reliability guarantees. This seven-function API is designed to implement reliable message passing between two application processes.

The code fragment below sends two messages from host1 to host2.

```

1 char buf1[BUF1_SIZE];
2 char buf2[BUF2_SIZE];
3
4 miami_init("host1", 55043, "host2", 61200);
5 handle1= miami_tx_start(buf1, BUF1_SIZE, 12);
6 handle2= miami_tx_start(buf2, BUF2_SIZE, 13);
7 while (!miami_tx_done(handle1)) {
8     /* Wait */
9 }
10 while (!miami_tx_done(handle2)) {
11     /* Wait */
12 }
13 miami_finalize();

```

The next code fragment would be executed on host2 and receives the messages. Note how tag matching is used to select particular messages for particular receive buffers.

```

1 miami_init("host2", 61200, "host1", 55043);
2 handle1= miami_rx_start(buf1, BUF1_SIZE, 13);
3 handle2= miami_rx_start(buf2, BUF2_SIZE, 12);
4 while (!miami_rx_done(handle1)) {
5     /* Wait */
6 }
7 while (!miami_rx_done(handle2)) {
8     /* Wait */
9 }
10 miami_finalize();

```

3 The API

The API is defined and described in a file named `miami.h` which is available on our web site.

4 Implementation

During initialization, inside `miami_init()`, your implementation must determine how much data can be sent in a single UDP datagram between the two participating hosts. It is then your task to fragment larger messages into the appropriate number of blocks to send across. Try a few datagram sizes and settle on the largest value that works.

When sending and receiving fragments that are part of a larger message, you must keep track which ones you have received, and which ones are still outstanding. That means each datagram must carry some of your protocol information, such as a sequence and a message number (multiple messages may be in flight simultaneously). While it is unlikely that datagrams will be reordered, it is quite possible that some of them will get lost. Your implementation must handle and recover from such events.

You may assume that the network is fairly reliable and that errors occur in bursts. That means it is probably simplest to resend the whole message in the case of an infrequent error. Note, though, that your implementation may have to send the message several times before it successfully makes it to the other end.

You may also assume that if UDP successfully delivers a packet to your implementation, that the data contained in the payload is correct; i.e., you do not need to do any further checksum or CRC checking beyond UDP and IP.

MIAMI guarantees message ordering. If the send of message A starts before that of message B, and both of them have the same tag, then the first receive posted must receive message A, and the second receive posted must receive message B. This is independent of the length of these messages. Message A could be very long, while message B could be very short.

Your implementation should be space and performance efficient. It should not be excessive in its use of storage. For example, you should leave messages in the buffers provided by the user and not copy them to temporary space before sending them. Likewise, if a message starts coming in, but no receive buffer has been posted for it yet, you should wait to receive more data until you have a user provided buffer to store it. A secondary goal should be to send and receive messages as quickly as possible.

Remember that a non-blocking API allows multiple messages to be in flight at any time.

5 Error checking

Your implementation should do some reasonable error checking. For example, calling `miami_tx_done()` with a receive instead of a send handle, should return an error; not crash the program.

6 Documentation

In a PDF file named `Project.pdf` you should describe how your implementation works and state any shortcoming; i.e., unimplemented parts. In particular, describe the wire protocol

you have implemented; i.e. describe the messages you send back and forth to initialize the system and transfer a long message from one host to another. Describe the case when the receive has been posted at the time of the send, and when the receive is posted later. I expect the description will require about one page of text. Be thorough, yet concise.

7 Program

You will turn in a source code file named `miami.c` and a Makefile. Running the Makefile will produce a `.o` file that can be linked with an application that uses the MIAMI API.

8 Grading

Any violations of the items in Section 1 will cause you to lose some points. The amount will depend on how much time it costs me to work around your decision not to follow my requests.

About 60 points will be given for a minimalist implementation that compiles, and implements all seven API functions. For this initial implementation you can ignore tag matching and then non-blocking behavior of the initiating send and receive functions. the corresponding completion functions (`miami_tx_done()` and `miami_rx_done()`) then become no-ops.

Additional points are earned by implementing matching, and the asynchronous aspects of the API.