

# Project 1

CS/EE 555

Fall 2001

Due Oct 5, 2001

## Description

In this project you will implement the sliding window protocol (SWP), described in Chapter 2 of the textbook, using UDP packets to simulate the link layer. You will need to decide on details of the protocol such as the designing the header, choosing appropriate timeouts and deciding when to retransmit data and ACK packets, etc.

I have two primary goals in assigning this project:

- that you gain some experience with the techniques of protocol implementation
- and that you experience first-hand the effects of different link characteristics and protocol choices on protocol performance by seeking to optimize the protocol choices for several different links.

You will write 2 programs, a sender and a receiver. The sender will consist of a main program that reads command line arguments, configures your SWP implementation, then makes repeated calls to the function `p1_send` to send data across the link. Similarly, the receiver's main program will read command line arguments, configure the receiver's SWP implementation and make repeated calls to `p1_receive` to receive the data. `p1_send` and `p1_receive` provide the service interface for your protocol. Together with the things they call they will be your implementation of SWP. Your main programs should provide an independent check that all the data was successfully transferred and a mechanism for determining the performance achieved. At a minimum the latter should consist of determining the total time taken for the transfer, but it will be more interesting if you record the arrival time of each packet at the receiver and graph the total data transferred vs. time. Initially, your sender should send packets directly to your receiver, which may be on the same or a different machine. This will allow you to make sure that the basics of the protocol are working over a reliable connection. However for what you turn in, you will use a *link emulator* to connect the sender and receiver. The emulator will allow you to experiment with links that are less reliable, slower and have greater delay than the direct Ethernet connection.

Note that this structure is different from what is described in the book: in your implementation the sender sends data and receives acks. The receiver receives data and sends acks. Your protocol and implementation does not need to support both sending and receiving by a single program. This is intended to make it easier for you to get to the point where you can explore the implications of different link characteristics.

## Experiments

The required experiments involve sending 500Kbytes of data over 3 different emulated links.

- A 1Mbps link with a 100Ms delay and an error rate of 1 bit in  $10^5$ .
- A 100Kbps link with a 5Ms delay and an error rate of 1 bit in  $10^5$ .
- A 100Kbps link with a 100Ms delay and an error rate of 1 bit in  $10^4$ .

The error rates are very high but that's necessary to make the exercise interesting.

For each experiment you should choose the best possible operating parameters for your protocol implementation and report the observed performance over several trials. How does this compare with the performance you expect? Can you explain any differences? Do you wish you had made different design decisions? Is there a limitation of the operating system or programming language you are using that makes achieving good performance difficult?

### Directly connecting the sender and receiver

The sender and receiver may be on the same or different machines. The sender needs to know how to send data to the receiver. There are a couple of approaches you might take. You could decide that the receiver will always bind its socket to a particular port, but you run the risk in this case that someone else will be using the port when you want it.

Another approach is to call `bind()` in the receiver with an unspecified port so the system assigns the port. If you take this approach you will have to output the port number and pass it as a command line argument to the sender along with the hostname or address for the receiver. In this direct connection scenario, the sender's data packets travel from the sender port, SP, to the receiver port, RP, and the acks travel from RP to SP, all using UDP.

### Connecting via the emulator

When using the emulator, packets will travel from the sender to the emulator and from there to the receiver after the appropriate amount of time (if not dropped due to a simulated error). ACK packets return via the emulator, as well. To use the emulator you must request its service by sending a request packet from the SP to the well-known emulator port (WEP) telling it where your receiver is and what emulation parameters you want applied. The emulator will create a new, private emulator port (PEP) for your connection and send a reply from PEP to SP indicating that the emulator is ready. Thereafter, packets sent by SP to PEP are forwarded to RP (and look to the receiver like they came from PEP, according to the UDP header). Packets from RP to PEP are forwarded to SP (and look like they came from PEP according to the UDP header).

The details of how to establish an emulator session are captured in the file `emulator_connect.c`, available in `/net/cs555/assignments/project1`. The parameters for the three required experiments are preconfigured in the procedure `int em_connect(int param_set, int sp_socket, struct sockaddr_in *rcvr)`. `param_set` is 1, 2, or 3 corresponding to each of the experiments described above. `sp_socket` must be an appropriately initialized

AF\_INET, SOCK\_DGRAM socket that the procedure can use to send the emulator setup request and receive the response. This socket must also be subsequently used for sending the data, since it is only data from this socket that will be forwarded to the receiver. The `rcvr` parameter should be initialized with the address and port number of the receiver. On successful return it will contain the address and port number of the private emulator port that the sender should subsequently use as the destination for its data. It is suitable for use in `sendto` system calls. I suggest copying `emulator_connect.c` and the header files it uses to your project directory.

### What to turn in

- The following are to be placed in your `cs555` directory in SNIF lab. Everything goes below `/net/cs555/students/youruserid/project1/`. **Please follow the directions carefully about where to place things: it makes life a lot easier for Jin and me if everyone follows the same conventions about file placement.**
- In the `doc/` subdirectory: a design document that describes all the important decisions you made about the service and peer-to-peer interfaces of your protocol and how you came to make those choices: include any analysis and the results of any preliminary experiments. The design document should also describe the mechanism you used in the main program to independently verify that all the data had been transferred as well as the methods used to measure performance. This file may be ASCII text, MS-Word, HTML or PDF.
- In the `programs/` subdirectory: all the source code for your sender and receiver including a Makefile, and the executables for the sender and receiver.
- In the `report/` subdirectory: your report on each of the required experiments. Provide the command line used to invoke your sender and receiver for each experiment, your report on the expected and observed performance and the answer to each of the questions listed in the description of the experiments above. At the end of your report include summary of what you learned in doing this project. The report file (only one please) may be ASCII text, MS-Word, HTML or PDF.

### Notes

- Different parameters for the protocol will be appropriate for different link characteristics. Your protocol does **not** have to dynamically adapt to different link characteristics, but you will need to be able to configure appropriate behavior using command line switches. What is configurable and how you configure it is part of your design.
- You do not need to do fragmentation/reassembly below the `p1_send/p1_receive` interface. If you want to change the packet size you can do it in the main program.
- If you choose to gather per-packet data, there are few enough packets making up each experiment that you can record the data in memory and only write it out at

the end of the experiment. This removes one source of noise in your data relative to trying to write it out while the experiment is going on.

- You may use the programming language of your choice as long as I agree to it ahead of time. C, C++, Java and Python are already blessed. I'm inclined to agree to Perl and Visual Basic but I want to talk to you ahead of time if you are considering either of these or another programming language.
- Although you may choose to make your sequence number field large enough that it never overflows during these experiments, your code should correctly handle overflow of this field.
- The emulator simulates bit errors by dropping packets at the appropriate rate. You do not need to provide an independent checksum implementation for received packets – your code may assume that if a packet is received it is correct.
- The emulator instance that is created by calling `em_connect` will timeout and exit if it doesn't receive a packet from either the sender or receiver for a period of 10 minutes. This is to prevent the accumulation of useless processes on the emulator host. If the timeout period is too short for your debugging needs it could be raised. Let me know.
- You will need an additional procedure (in addition to `p1_send`) in your sender protocol that your main program will call to indicate that it is finished sending and to wait for any already-queued data to be acknowledged. You will need some way in your protocol to indicate the end of the data to the receiver.
- You will probably want to implement an event mechanism similar to that described at the end of Chapter 1. I've provided the header file for the event library I built for the emulator in `/net/cs555/assignments/project1/ev.h` as a model for how you might proceed.

### **Limitations due to the approach**

Because we are simulating the link layer with UDP packets, allowing the packets to get too large will result in packet fragmentation, which may well be confusing. The maximum size of packets used in your implementation should result in fragmentation.

Because we are running the programs over a real Ethernet that carries other traffic, we cannot control the service underlying the emulated link. That's a fact of life. However, given the relatively low speeds and long delays of the emulated link it should not be a major factor in the results you observe.

The delay, bandwidth and packet-drop emulator you are using is simulated by a real program running on a real machine and will not be perfect in its simulation of delays. However, it seems to be pretty good at providing delays within a few hundred microseconds of what is specified for nearly all packets sent through it. If you detect that this is not true please let me know.