CSCI 4273/5273: Network Systems
Fall 2014
Programming Assignment 4

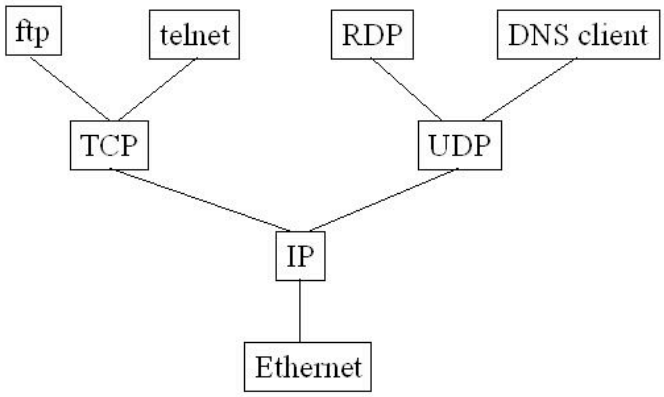
Due Date: 12/05/2014

Goal: The goal of this programming assignment is to evaluate two network implementation models: *process per protocol model* and *process per message model*. The assignment consists of implementing these two models and comparing their performance under some specific operating conditions. It requires familiarity with `pthreads` thread package and Unix system calls such as `pipe()` and `gettimeofday()`. You will use the Threadpool and Message libraries you implemented in programming assignment 3. A sample implementation of these libraries is provided.

Grade: 15% of your final grade is allocated for this programming assignment.

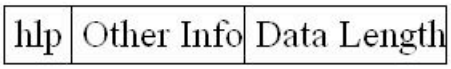
Team: You may work on this assignment in teams of up to two students.

An important guideline in designing a network system is that we *design for performance*. Performance is as important as correctness in building a network system. When designing a network system, one of the first questions you must ask yourself is *Where are the processes*?. An answer to this question will significantly affect the performance of your implementation. In this assignment, you will experiment with two implementation models: (1) *Process per protocol model*: Each protocol is implemented by a set of one or more threads; and (2) *Process per message model*: Each message is shepherded by a thread. You will implement the following protocol graph:



You will implement only a skeleton of this protocol graph, where outbound messages move down from higher-level protocols to lower-level protocols, and inbound messages move up from lower-level protocols to higher-level protocols along a protocol stack. Every protocol must implement two functions: message encapsulation (attaching/stripping headers) and multiplexing/demultiplexing. You will not implement any other protocol-specific functions. Use the following protocol IDs to identify a protocol: Ethernet: 1; IP: 2; TCP: 3; UDP: 4; ftp: 5; telnet: 6; RDP: 7; DNS client: 8.

Format of protocol headers of course depends on the protocol being implemented. For this assignment, assume that protocol headers of all protocols contain three fields as shown in the figure below. Here, *hlp* is the protocol ID of the higher-level protocol. So, the value of *hlp* in the protocol header of Ethernet will be 2 (protocol ID of IP), value of *hlp* in the protocol header of IP will be either 3 (protocol ID of TCP) or 4 (protocol ID of UDP), and so on. The size of the *Other Info* field is protocol dependent: Ethernet: 8 bytes; IP: 12 bytes; TCP: 4 bytes; UDP: 4 bytes; ftp: 8 bytes; telnet: 8 bytes; RDP: 12 bytes; and DNS client: 8 bytes. Finally, *Data Length* is the size of the message, not including the header.



Process per protocol model

In this model, a protocol is implemented by one or more threads. As a message moves up or down a protocol stack, it is passed from one protocol (thread) to another using an IPC mechanism provided by the operating system. In this assignment, use Linux pipes for passing messages between different protocols. In particular, each protocol implementation provides two input pipes: (1) *send pipe* to receive data from a higher-level protocol, and (2) *receive pipe* to receive data from a lower-level protocol.

For an inbound message, a thread receives the message and writes it on the receive pipe of the Ethernet protocol. The Ethernet protocol thread reads this message from its receive pipe, processes it (strips the Ethernet header), and then writes appropriate information on the receive pipe of the next higher-level protocol (IP) in the protocol stack. An IP protocol thread in turn reads information from its receive pipe, processes it (strips the IP header), and then writes appropriate information on the receive pipe of the next higher-level protocol (TCP or UDP) in the protocol stack. This process continues until that message is received by an application-level protocol thread. The application-level protocol processes the message (strips its header) and then prints it out. In general, a protocol thread will write one item in the receive pipe of its higher-level protocol: pointer to a Message object.

For an outbound message, an application thread running on top of an application-level protocol, e.g. ftp, writes appropriate information on the send pipe of the application-level protocol (ftp). An application-level protocol thread reads information from its send pipe, processes it (attaches its header), and then writes appropriate information on the send pipe of the next lower-level protocol in the protocol stack. This process continues until that message is received by an Ethernet protocol thread. The Ethernet protocol processes the message (attaches its header) and then sends it out on the network. In general, a protocol thread will write two items in the send pipe of its lower-level protocol: its protocol ID and a pointer to a message.

Your implementation should maintain an array of file descriptors for send pipes and another array of file descriptors for receive pipes, and index these arrays using protocol ids.

Process per message model

This model treats each protocol as a passive piece of code and associates threads with messages. Each protocol exports two operations: `send (int` protocol_id, `Message` *msg) and `deliver (Message` *msg). A thread pool of 25 threads is maintained to handle incoming messages arriving at the network level. Use your ThreadPool library for this.

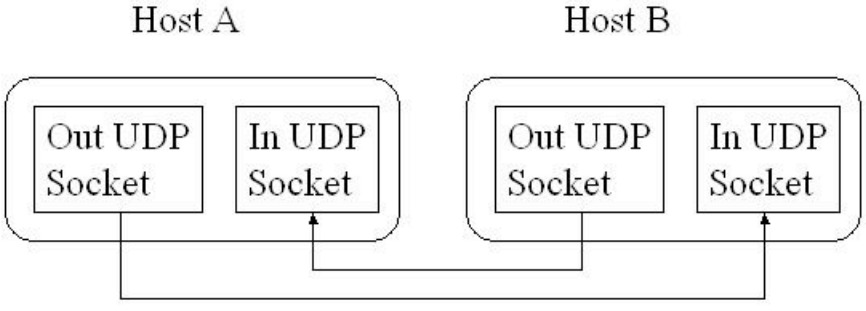
For an inbound message, a thread receives the message at the network level and then dipatches a thread from the thread pool to process the message. This dispatched thread from the thread pool shepherds the message up a protocol stack by invoking the `deliver()` function of the Ethernet protocol. The `deliver()` function processes the message (strips the header), and then invokes the `deliver()` function of the protocol at the next higher level in the protocol stack. In turn, this `deliver()` function invokes the `deliver()` function of the protocol at the next higher level in the stack after processing the message, and so on. The `deliver()` function of the application-level protocol processes the message (strips its header) and then prints it out.

For an outbound message, an application thread running on top of an application-level protocol, e.g. ftp, invokes the `send()` function of the application-level protocol. This function processes the outbound message (attaches its header), and then invokes the `send()` function of the protocol at the next lower level in the protocol stack. In turn, this `send()` function invokes the `send()` function of the protocol at the next lower level in the stack after processing the message, and so on. The `send()` function of the Ethernet protocol processes the message and then sends it out on the network.

Your implementation should maintain an array of function pointers for `send()` functions and another array of function pointers for `deliver()` functions, and index these arrays using protocol ids.

Network simulation

To evaluate these models, simulate a network of two hosts A and B that are connected to each other by two UDP links: one for incoming messages and one for outgoing messages.



To send a message out on the network a thread writes that message on the `Out UDP Socket`. In case of process per protocol model, this is the thread that implements processing of outbound messages. In case of process per message model, this is the application thread. To receive a message from the network, use a separate thread that reads messages from the `In UDP Socket`. In case of process per protocol model, this thread writes that message on the receive pipe of the Ethernet protocol. In case of process per message model, this thread dispatches a thread from the ThreadPool object to shepherd the message up the protocol stack.

Applications

Implement one application (one thread) per every top-level protocol (four applications in total) on both hosts. These applications repeatedly execute the following code 100 times:

```
create a message (message size: 100 characters)
send message
   • the application thread writes to the send pipe of the
     application-level protocol in process-per-protocol
     models
   • the application thread invokes the send() function of
     the application-level protocol in process-per-message
     model
sleep for a few milliseconds
```

Evaluation

Measure the total time required to complete all four applications (100 messages sent by each of the four applications from two hosts and all 400 messages received by each host) under three different experimental setups: (1) Both hosts are running process-per-protocol implementations; (2) Both hosts are running process-per-message implementations; and (3) One host is running process-per-protocol implementation and the other host is running process-per-message implementation.

Assignment Submission

1. Submission deadline is Friday, December 05, midnight. No late submissions will be allowed, unless there is a valid excuse.
2. Submit a single zip file via the submission link on Moodle. Your zip file must contain all your source code including all three libraries, Makefile and README.
3. DO NOT include any object files in your submission.
4. In the README file, provide the following information: Your name; instructions on how to compile and run your program; current status of your program: whether it compiles or not, known bugs/limitations/unusual features, what parts of the program work, etc.; any other information that will be useful in grading your program. In addition, include the times you measured for the three scenarios.