CSE 5462 Project: Spring 2014

Instructor: Adam C. Champion
Project Webpage: `http://www.cse.ohio-state.edu/~champion/5462/proj`

1 Overview

The goal of this project is to implement a TCP-like reliable transport layer protocol using the unreliable service provided by UDP and then write a simple file transfer application to demonstrate its operation.

A. Function Calls to be Implemented: The arguments and return values of these function calls need to exactly match those of the corresponding function calls for UNIX socket programming. To implement these functions, you can use any UDP-related function calls.

- `SOCKET()`
- `BIND()`
- `ACCEPT()`
- `CONNECT()`
- `SEND()`
- `RECV()`
- `CLOSE()`

The focus of the project is on data transfer. Most of the TCP functionality will be implemented in the `tcpd` (TCP daemon) process, which is equivalent to TCP in Linux that runs in the background. These function calls will require communicating with the local `tcpd` process. The communication between the application process and the local `tcpd` process can be implemented with any inter-process communication mechanism such as UDP sockets. UDP communication within a machine can be assumed to be reliable.

Write a simple file-transfer application that uses your TCP implementation. Note that you will need this program for testing your TCP implementation. The file-transfer protocol will include a server called `ftps` and a client called `ftpc`. Start the server using the command

```
ftps <local-port>
```

Start `ftpc` with the command

```
ftpc <remote-IP> <remote-port> <local-file-to-transfer>
```

The `ftpc` client will send all the bytes of that local file using your implementation of TCP. The `ftps` server should receive the file and then store it. Make sure that after receiving the file at the `ftps` server you either give the file a different name or store it in a different directory than the original since all the CSE machines have your root directory mounted. Otherwise, you will overwrite the original file.

The file-transfer application will use a simple format. The first 4 bytes (in network byte order) will contain the number of bytes in the file to follow. The next 20 bytes will contain the name of the file. The rest of the bytes to follow will contain the data in the file.

To simulate real network behavior, all communication between the two machines will pass through local `troll` processes. `troll` is a utility that allows you to introduce network losses and delay. More details on `troll` are provided later.

The steps for transferring a file from machine $M2$ (client machine) to machine $M1$ (server machine) are as follows:

(a). Start the `troll` process and the `tcpd` process on machines $M1$ and $M2$.

(b). On machine $M1$, start the file-transfer server `ftps`. It will make the function calls `SOCKET()`, `BIND()` and `ACCEPT()`. `ftps` will then block for a client to connect.

(c). On machine $M2$, start the file-transfer client, `ftpc`. It will make function calls `SOCKET()`, `BIND()`, and `CONNECT()`.

(d). Normally the `CONNECT()` should initiate TCP handshaking between the two `tcpd` processes. But in this project you are not implementing TCP handshaking. Thus `CONNECT()` is a null function.

(e). The buffer management for this connection will be done in $\texttt{tcpd}_{M2}$.

(f). `ftpc` will read bytes from the file and use the function `SEND()` to send data to `ftps`. The `SEND()` function call will need to send these bytes to the local `tcpd` process. The `tcpd` process will then store these bytes in a circular buffer.

`SEND()` should be implemented as a blocking function call. It should not return until all bytes in the buffer passed in the argument is written in the `tcpd` buffer.

(g). The buffer management functions will then take bytes from the buffer and create packets.

(h). Upon receiving the first byte, `tcpd` on $M1$ will unblock the `ACCEPT()` call. The `ftps` application will then make calls to `RECV()` to receive data.

`RECV()` should not return until at least one byte is read from the `tcpd` buffer. However, if multiple contiguous bytes are available they should be read to fill up the buffer up to the maximum size specified in the argument.

(i). After sending all the bytes of the file, `ftpc` closes the connection. The `CLOSE()` function call will initiate closing the connection.

(j). On receiving all the bytes of the file, `ftps` will close the connection using the `CLOSE()` function.

B. Connection Setup: As you are not implementing TCP handshaking, the connection setup steps are fairly simple:

- Initially the application process on $M1$ calls `ACCEPT()` and blocks. The `ACCEPT()` function in turn will send a message to the local `tcpd` and wait to hear back when a client has successfully connected.
- The `CONNECT()` function on $M2$ is an empty (or null) function.
- On reception of the first data packet, $\texttt{tcpd}_{M1}$ sends a message to the process waiting on `ACCEPT()`.

These steps are shown in Figure 1.

C. RTT Computation: Implement Jacobson's algorithm for computing RTT and RTO.

D. Checksum Computation: The CRC (Cyclic Redundancy Code) checksumming technique should be used for computing the checksum.

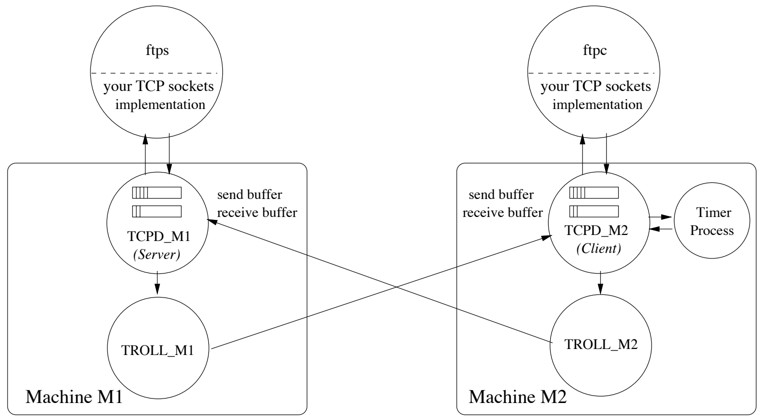

Figure 1: Connection setup

E. Packet Formats: The TCP packet structure should be strictly followed for both TCP and ACK packets. Instead of TCP's cumulative ACK, the ACK packet will acknowledge the data packet just received. Note that each packet will be ACKed.

F. Timer Implementation: Each data packet after transmission will require a timer to be started. When the timer runs out, the packet will need to be re-transmitted. Since a large number of packets may be in transit at any given time, a large number of timers may be simultaneously running.

Instead of using explicit timers for each packet, you will implement the timers using a delta list. More details on delta lists are available on the project website.

The delta list must be maintained in a separate process called the *timer process*. When a new timer needs to be started, a message is sent to the local timer process, indicating how long the timer needs to run for, which port the timer process should send notification upon expiry, and the byte sequence number of the packet for which this timer is being started.

G. Buffer Management and Sliding Window Protocol: Implement the selective repeat algorithm. Use a fixed window size of 20. You are not required to implement slow-start, congestion control, or flow control algorithms.

The send and receive buffers will be circular buffers. Use a 64 KB sending buffer and a 64 KB receiving buffer and a 1,000 byte MSS.

Buffers/arrays in other processes should not store more than 1 MSS worth of data.

H. Connection Shutdown: Implement the exact state diagram for shutdown of a TCP connection. All the data structures related to the socket will be deallocated. However the buffer management function should make sure that all data has been acknowledged before deallocating the data structures.

2 `troll`

In the CSE network it is hard to artificially create real network scenarios (lossy links, packet garbling, etc.) Use the `troll` utility to control the rate of garbling, discarding, delaying or duplication of packets. All packets will first go through a local `troll` process running on the same machine, where they will be subject to delay, garbling and/or drops. The packets will then be forwarded to the intended destination.

Test the `troll` program using the `totroll` and `fromtroll` programs. Source code for the two programs is available on the project webpage. Here is an example of how you can test `troll`'s functionality (illustrated in Figure 2):

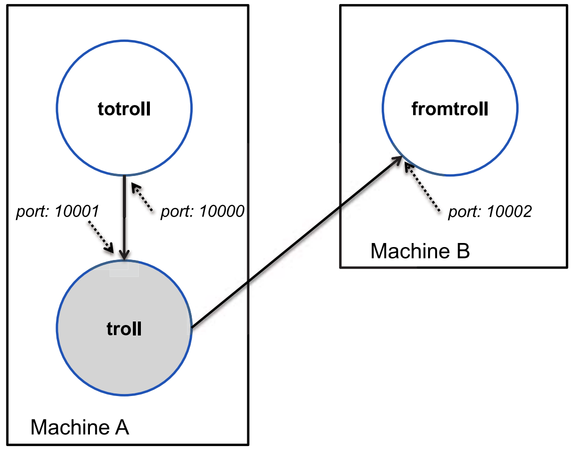

Figure 2: `troll`

- On machine A, start `troll` to communicate on port 10001 using the command:

  troll -C B -S A -a 10002 -b 10000 10001
  or
  troll -C A -S B -a 10000 -b 10002 10001

- On machine B, start `fromtroll` to listen on port 10002 using the command

  fromtroll 10002

- On machine A, use `totroll` to send short messages via `troll` to `fromtroll` on machine B using the following command:

  totroll A 10001 10000

`totroll` and `fromtroll` are the two ends to communicate with each other through `troll`. You can designate either of them to be the client and the other to be the server. A and B in the commands above have to be replaced by the IP addresses of the corresponding machines. You can use a combination of the following commands to find the IP address of your machine: `nslookup`, `hostname`, `ifconfig`. You can choose any two machines to run the `fromtroll` and `totroll` programs. Note that the particular port numbers used in the example above may be unavailable if some other process is using it.

3 Testing and Developing the Project

This project has several components. Debug and test each function carefully before integrating it into your code. Start your work *very early* as certain components such as the sliding window protocol may take a long time to debug.

4 Checkpoints

Group Formation (due Jan. 16, 2014, in class): Write your and your partner's name in the signup sheet to be passed in class.

Project Proposal (due Jan. 30, 2014, in class): The proposal must include details on the circular buffer, packet formats (formats for all packets exchanged between `tcpd`, `ftpc`, `ftps`, `troll`, and `timer` processes), timer process, checksumming algorithm, RTT/RTO computation, description of operations within the implementation of all `monospaced` functions, and connection shutdown. Details of various data structures and when/how they are updated must be included for each of those modules. In addition, present a clear timeline and work distribution plan with specific internal milestones. Proper planning with time allocation for debugging and testing is crucial for successful execution of the project.

Your proposal needs to be typed using software such as MS Word, FrameMaker, LaTeX, etc. Use 11 pt font with single spacing, single column. The recommended length is 6–8 pages.

Checkpoint Demo and Submission (due Mar. 6, 2014): Be creative in how you demonstrate the modules' correct functionality (see table).

In addition to the demo, you need to submit your code using the `submit` utility by 11:59 p.m. on Mar. 6, 2014. Use the following command for submitting the code:

```
submit c5462aa lab6 <code-directory-name>
```

Your code directory needs to contain a `README` file that describes all the C files in your directory. Also, indicate how to test the functionality of each module. The code directory must contain a `Makefile`. If you resubmit the files, submit both the code and the final report again, as each invocation of `submit` deletes all files of the previous submission for the same lab. Read `man submit` for clarification. Only one person from each group needs to submit the project.

Final Demo (Sat., Apr. 19, 2014): The final demonstrations will be held in Caldwell 112 on Saturday, Apr. 19, 2014 from 2–5pm. There will be a signup sheet for each team.

Final Report and Code Submission (due Apr. 21, 2014 at 11:59 p.m. EST): Submit the final version of your code and the final project report by this deadline. The final project report should include details on the implementation, discussion on all the features of your program (including any extra features beyond what is required), optimizations, and possible enhancements for the future. The report should only include code fragments if they are absolutely essential.

The report needs to be typed using software such as MS Word, FrameMaker, LaTeX, etc. Use 11 pt font with single spacing. The report needs to be 6–10 pages long.

Submit your code and final report using the `submit` utility. Use the following command for submitting the final project code and the final report:

```
submit c5462aa lab7 <code-directory-name> <final-report-file-name>
```

Your code directory must contain a `README` file that describes all the C files in your directory. Also, indicate how to run your program. The code directory must contain a `Makefile`. If you resubmit the files, submit both the code and the final report again, as each invocation of `submit` deletes all files of the previous submission for the same lab. Read `man submit` for clarification. Only one person from each group needs to submit the project.

5 Miscellaneous

- **Team Formation:** You can either work by yourself or in a group of two (recommended). To the extent possible, please make sure that your group partner is a motivated and hard-working student. A significant portion of the grade depends on the project. You will be identifying your individual contributions in the project. If the contributions are significantly different, each team member may get a different score.
- **Platform:** Use the `stdlinux` system (*not* the testbed) for all your implementation.
- **Delayed Submissions:** Late demonstrations, code submissions or report submissions are *not* eligible for any points.

Table 1: Schedule and grading rubric

Deadline	Item	Point Details	Points
In class, Jan. 30, 2014	**Proposal**		**6**
	Circular buffer	1	
	Packet formats (for all packets exchanged between `tcpd`, `ftpc`, `ftps`, `troll`, and `timer` processes)	1	
	`timer` process	1	
	Checksumming algorithm	1	
	RTT and RTO	0.5	
	Description of operations within the implementation of all `CAPITALIZED` functions	0.5	
	Connection shutdown	0.5	
	Timeline and work distribution	0.5	
Mar. 6, 2014, in Caldwell 112, 4–7pm	**Checkpoint Demonstration**		**14**
	Delta timer	7	
	Checksumming	7	
Submit by 11:59 p.m., Mar. 6, 2014	**Well-documented code**		**2**
Apr. 19, 2014, in Caldwell 112, 2–5pm	**Final Demonstration**		**22**
	RTT computation	4	
	Circular buffer management	6	
	Shutdown	2	
	Successful execution with `troll` on both machines (with garble 25%, destroy 25%, duplicate 25%, reorder, exponential decay with mean 10 ms)	10	
Submit 11:59 p.m., Apr. 21, 2014	**Final Report**		**4**
	Project overview and details on each process	1	
	Detailed description of each project component (circular buffer, `timer`, checksumming, RTT/RTO, packet formats, implementation of `SOCKET()`, `BIND()`, `SEND()`, `RECV()`, and `CLOSE()`)	1	
	Detailed description of how to compile, run, and test the program	1	
	Possible future extensions to make this program more efficient and to add more features	1	
Submit 11:59 p.m, Apr. 21, 2014	**Well-documented code**		**2**
	TOTAL		**50**