CS 536 Fall 2019

Lab 2: Basic Socket Programming and Small-footprint Reliable Transport [250 pts]

Due: 09/30/2019 (Mon), 11:59 PM

Objective

The objective of this lab is to practice basic network programming using datagram and stream sockets. You will also program a simple stop-and-wait reliable file transport app using SOCK_DGRAM sockets where network protocol design, implementation, and testing come into play. This lab is an individual effort. Please note the assignment submission policy specified on the course home page.

Reading

Read chapter 2 from Peterson & Davie (textbook).

Problem 1 [50 pts]

1.1 General background

Modify Problem 2 of lab1 so that server and client run on different machines -- e.g., one on a machine in LWSN B148 and the other on a Linux PC in HAAS G056 -- and use datagram sockets to communicate. A socket is a type of file descriptor (Linux/UNIX distinguishes 7 types including regular file, directory, FIFO, socket) which is primarily used for sending data between processes running on different hosts. The hosts may run different operating systems (e.g., Linux, different UNIX flavours, Windows) on varied hardware platforms (e.g., x86 and ARM CPUs). Our lab machines run Linux over x86 PCs.

A socket can be of several types. In this problem, you will use datagram (SOCK_DGRAM) sockets that invoke the UDP (User Datagram Protocol) Internet protocol implemented inside kernels. We will discuss the inner workings of UDP when we study transport protocols. In the confines of user space app programming, all that you are doing is replacing the FIFO in Problem 2 of lab1 with a SOCK_DGRAM socket to achieve inter-process communication across different hosts, i.e., network communication. To identify a process running on a host such as PC, server, router, smartphone under the governance of an operating system (some devices may support network protocols but not run a multi-programming kernel), we need to know how to reach the host -- its IP (Internet Protocol) address -- and which process we wish to talk to given by its port number. We cannot use process ID (PID) to identify a process as PIDs are allocated to user processes by an operating system as it sees fit, not per request of the app programmer. A 16-bit non-negative integer, called port number, serves as an alias of PID to which every process wishing to engage in network communication must bind to.

Port numbers 0-1023, called well-known port numbers, cannot be used by app programs. Port numbers 1024-49151, referred to as registered port numbers, are usable by app processes. However, it is considered good practice to avoid using them to the extent meaningful. In many instances, networked client/server apps -- the same goes for peer-to-peer apps which are just symmetric client/server apps where a host is both a server and a client -- are coded so that they do not depend on specific port numbers which facilitates portability and robustness. We will do the same in our labs when feasible. The host on which a destination process runs is identified by an IP address. Although IPv6 (version 6) with 128-bit addresses is partially deployed and used, IPv4 (version 4) with 32-bit addresses remains the dominant protocol of the global Internet today. Unless otherwise noted, we will use IPv4 addresses. Although we loosely say that an IP address identifies a host, more accurately, an IP address identifies a network interface on a host device. Hosts that have multiple network interfaces (e.g., a smartphone has WiFi, Bluetooth, cellular, among other interfaces) may have multiple IP addresses, one per network interface. Such hosts are called multi-homed (vs. single-homed). A network interface need not be configured with an IP address if there is no need to speak IP.

Most network interfaces have unique 48-bit hardware addresses, also called MAC (medium access control) addresses, with IP addresses serving as aliases, i.e., logical addresses that are configured by an operating system. In socket programming, we use IP addresses, not hardware addresses to facilitate communication between processes running on different hosts. IP addresses are translated to hardware addresses by a kernel before delivery over a wired/wireless link with the help of network interface hardware and its software (called firmware). We will study the workings of their interaction inside kernels when investigating the network layer. Lastly, to facilitate human readability of IP addresses, a further layer of abstraction is implemented in the form of domain names. For example, www.purdue.edu is mapped to IPv4 address 128.210.7.200 where the four decimal numbers specify the 4 byte values of a 32-bit address. This translation operation is facilitated with the help of a distributed database system called DNS (Domain Name System). Many of the aforementioned features and capabilities are accessible via socket system calls in user space.

1.2 Specific implementation constraints

When re-implementing the FIFO-based client/server app of Problem 2, lab1, to use sockets, both server and client make socket() system calls to allocate a file descriptor (i.e., socket descriptor) of type SOCK_DGRAM. A socket descriptor is but a handle and must be further configured to specify who the communicating parties are, and possibly other properties. After socket(), bind() is called by both server and client to bind to an IP address (a specific network interface in case a host is multi-homed) and a port number on their local hosts. One way of binding to an IP address is to use INADDR_ANY in a field in the address structure, struct sockaddr_in. After an address structure is adequately populated, it is passed as an argument to bind(). In your implementation, use the Linux command, ifconfig -a, to determine the IPv4 address (following the dotted decimal notation) assigned to Ethernet interface eth0 (e.g., 128.10.25.208 on pod2-3.cs.purdue.edu) and provide it as a command-line input to your server process, call it remote-command-srv.

When binding to a port number, we will let the operating system choose an unused number, referred to as allocating an ephemeral port number. We do so by specifying 0 in the port number component of the sockaddr_in struct. When populating message related fields, keep in mind that big endian (also called network byte order) is used in networks whereas x86 under Linux uses little endian to represent data. You must ensure proper conversion is effected. For our purposes, that means IP addresses (4 bytes) provided in dotted decimal form and port numbers (2 bytes). We will leave the payload untouched since the only application layer field (see Problem 3) whose value is interpreted by a receiver as a multi-byte data type is a 1-byte sequence number field. If we were to use an unsigned integer (4 bytes) to implement an application layer sequence number field and the receiver ran on a host obeying big endian, then changing the byte order of the 4-byte unsigned int before handing off to a kernel using sendto() may be necessary. After bind(), use getsockname() to query the port number assigned to the server process by Linux and print it to stderr. You will need the value when running the client process. In our lab machines, the file /proc/sys/net/ipv4/ip_local_port_range specifies the range of port numbers that Linux will dynamically assign.

On the client side, let the client process bind to an ephemeral port number. Provide the server's IP address and port number as command-line arguments of the client binary, remote-command-cli, followed by the command to be remotely executed. For example,

```
% remote-command-cli 128.10.25.208 50000 ls -l
```

The client uses system call sendto(), instead of write(), to transmit the request to the server process following the same format as in Problem 2, lab1. The server calls recvfrom(), instead of read(), to receive a message from a client. A major semantic difference between FIFO and SOCK_DGRAM socket communication is that in the latter message delivery is unreliable. That is, any damage encountered by the client's message along its journey results in a dropped/list message and no delivery to the server process. In this version of the networked remote command execution app, the server will not provide the client with an acknowledgment of message receipt. After transmitting a request, the client terminates as in Problem 2 of lab1. Create directory lab2/v1/ under your home directory on our lab machines and deposit your source code including Makefile. Test that your code works correctly by sending request messages from multiple clients running on different hosts. After testing, make sure to terminate your server process. Otherwise it can become a vulnerability if exposed to hackers. Provide adequate comments/annotation in your code.

Problem 2 [70 pts]

2.1 Using sliding window SOCK_STREAM

Create a subdirectory v2/ under lab2. Modify your code of Problem 4, lab1, so that bi-directional FIFO communication between client and server on the same host is replaced by bi-directional message communication using SOCK_STREAM sockets. SOCK_STREAM instructs Linux to use TCP (Transport Control Protocol) to transport bytes between sender and receiver. TCP implements reliable communication through ARQ using the sliding window protocol discussed in class. Unlike SOCK_DGRAM, SOCK_STREAM supports a reliable byte stream abstraction between sender and receiver via read() and write() system calls. When holes develop (i.e., bytes go missing), Linux implementing sliding window patches the holes so that user processes are shielded from its effects.

2.2 Server side system call sequence and actions

As much more work is required by SOCK_STREAM than SOCK_DGRAM sockets to run sliding window (and other protocol actions we will discuss when studying transport protocols), SOCK_STREAM sockets incur significantly more overhead. After calling socket() to allocate a SOCK_STREAM socket descriptor, the server calls bind() analogous to SOCK_DGRAM in Problem 1. After bind(), the server calls listen() to mark the socket descriptor as passive, waiting on connection requests from clients. The second argument of listen() specifies how many connection requests are allowed pending. For our purposes, 5 will suffice. After listen(), the server calls accept() which blocks until a client connection request arrives. When a client request arrives, accept() returns a new socket descriptor that can be used to communicate with the client while the old socket descriptor (i.e., the first argument of socket()) is left untouched so that it can be re-used to accept other connection requests. Technically, the socket descriptor following listen() specifies that it is of type SOCK_STREAM and what IP address and port number it has been bound to. The client's IP address and port number remains unspecified since a connection request has not been processed by calling accept(). A socket descriptor in this "half limbo" state is called a half-association.

When accept() returns because a client connection request has been received, the old half association socket descriptor is left untouched. Instead, a second socket descriptor is allocated which is a duplicate of the old descriptor (same action as dup2() but internal to the kernel) but with the client's IP address and port number filled in. The new socket descriptor is now a 5-tuple

(SOCK_STREAM, server IP address, server port number, client IP address, client port number)

with the client IP address and port number filled in, and is called a full association. In your concurrent server process, it is this new socket descriptor that is used by the child process to service the requested remote command execution and communicate the results back to the client. The technique is the same as in Problem 4, lab1, i.e., use dup2() to redirect stdout of legacy apps such as ls, date, ps, to the new socket descriptor. The parent process uses read() to read and parse the client request, delegates the execution of the actual task to a child process, and goes back to blocking on accept() for new client connections. As in Problem 4, lab1, upon receiving a client request, the server tosses a coin, and if it comes up heads, decides to ignore the request. When it decides to drop a request, it closes the connection to the client. The client, upon waiting for the response from a dropped request and timing out, closes the connection, sets a new timer and opens a new connection to the server. It does so up to three times. The server, remote-command-srv, is executed with a command-line argument specifying the IP address to bind to as in Problem 1.

2.3 Client side system calls and actions

Unlike Problem 4 of lab1, there is no need to set up a separate client-side FIFO to receive the server's response as SOCK_STREAM is bi-directional. The client, remote-command-cli, follows the same convention as in Problem 1 except that instead of calling bind() you call connect() with the server's IP address and port number. The kernel will fill in the client's IP address and port number (after allocating an unused ephemeral number) within connect(). If the client is multi-homed and wants to specific network interface to send/receive data, and/or wants to use a specific port number, then bind() can be used to do so. By default, bind() is omitted on the client side. Unlike in Problem 4, lab1, where a signal handler was used to respond to SIGALRM timeouts set to 2 seconds, use the select() system call without a SIGALRM handler to carry out stop-and-wait retransmission of dropped requests. Give up after 3 attempts. Follow the same instructions as in Problem 1 for testing and submitting your code.

Problem 3 [130 pts]

3.1 Stop-and-wait file transfer

TFTP (Trivial FTP) is a reliable file transfer protocol that uses UDP (i.e., SOCK_DGRAM socket) to achieve smaller resource footprint than FTP (File Transfer Protocol) which is TCP (i.e., SOCK_STREAM) based. As such it is used in network booting over LANs where computing systems undergo configuration and initialization with limited resources before their operating systems take control. Unlike TCP which uses sliding window ARQ to achieve reliable transport, TFTP uses stop-and-wait which significantly reduces its complexity and overhead. As discussed in class, this comes at the cost of speed in high-speed links due to the large delay-bandwidth product stemming from high bandwidth, especially in today's LANs. Two important parameters that influence TFTP's performance are block size -- how many bytes are sent in a single UDP packet (i.e., data size of sendto()) -- and timeout used by the server for retransmitting a packet. Timeout is set on the order of seconds without dynamically estimating round-trip time (RTT). In this problem, you will implement your own version of TFTP, call it MYFTP, aimed at evaluating the benefit of using RTT values that better reflect the conditions of a target LAN.

3.2 Server and client specification

The file server (i.e., sender), myftpd, is run with command-line arguments

```
% myftpd filesize blocksize timeout cli-ip cli-port
```

where filesize specifies the total number of bytes to be transferred (i.e., file size), blocksize is the data size in bytes of a single sendto() system call (see 3.5), timeout in unit of microseconds is the SIGALRM set to retransmit a packet if an ACK is not received in a timely manner, cli-ip is the dotted decimal IPv4 address of the client and cli-port its port number. Use the setitimer() system call of type ITIMER_REAL to set a high resolution timer to timeout microseconds in a one-shot (i.e., not periodic) fashion. When an ACK arrives in a timely manner, the current unexpired timer is cancelled and a new timer is set to the same timeout microseconds. Do not use other timer mechanisms nor the option provided by select() (used in Problem 2) to manage timeouts.

The client (i.e., receiver), myftp, is executed with command-line arguments

```
% myftp cli-ip dropwhen
```

where cli-ip is the IP address to be used by myftp (as in Problem 1 use ifconfig -a to determine IP address of eth0) and dropwhen is an integer that specifies how often the client should drop received packets. That is, not send a positive ACK to the server. For example, a value of 100 means that every 100th packet a positive ACK is not transmitted which will prompt the server to retransmit after its timeout expires. This emulates a packet loss rate of approximately 1%. Since the LANs connecting the Linux PCs in our labs are high-speed, unless significant load is introduced it is expected that packet loss will be low. Since lab resources are shared, we must abstain from negatively disrupting the system. The controlled losses allow us to do that, and, in addition, tune the losses when evaluating myftp's performance. dropwhen set to -1 means no server-side dropping. The client finds out the server's IP address and port number after receiving the first packet from the server by calling recvfrom(). After binding, myftp outputs its port number to stdout so that it can be used, along with its IP address, to run myftpd.

3.3 Estimating completion time and throughput

To measure completion time of file transfer, the receiver myftp calls gettimeofday() when it receives the first packet from sender myftpd using recvfrom(). It calls gettimeofday() again when it receives the last packet. By taking the difference, we can estimate approximate completion time. As a sanity check, make the receiver count the total data bytes received (excluding 1-byte bookkeeping overhead in 3.5) which must not count any duplicate packets received. Keep track of the number of duplicate packets received which is detected by the binary sequence number not having flipped. When the last data byte has been received, the client outputs to stdout total data bytes received, number of duplicate bytes (from duplicate packets) received, completion time (in milliseconds), and speed in unit of bps which is obtained by dividing total data bytes received by completion time. Note that this application layer bps speed underestimates the throughput bps achieved over Ethernet for the network system as a whole which includes bits from overhead introduced by application layer bookkeeping, UDP, IP, and Ethernet.

3.4 Eliminating file I/O influence

myftpd will not send the contents of an actual file of size filesize but ship data in units of blocksize until filesize bytes have been sent and reliably received. This implies that approximately filesize / blocksize number of SOCK_DGRAM sendto() system calls will be made. Of course, with losses introduced, retransmissions will results in additional sendto() calls being made. The reason we refrain from sending the contents of an actual file is to remove the impact that file system I/O will have on overall performance. That is, performing file I/O (at the server read() and at the client write()) is a costly operation which must be carefully managed to avoid becoming a bottleneck. Our focus is on gauging the impact of calibrating timeout to improve myftp performance. We will consider complications resulting from file I/O in lab3.

3.5 Payload format

The data given as input to sendto() obeys certain formatting rules. The first byte will contain an integer value of 0 or 1 which we will use as the sequence number of stop-and-wait. Note from our discussion in class that stop-and-wait requires a binary sequence number to indicate duplicate transmissions to achieve correct operation. We will limit the length of the actual pure data (i.e., part of the filesize bytes requested to be transferred by myftpd) given as input to a single sendto() call to 1471 bytes. The bound is derived from the our knowledge that the lab machines are connected by Ethernet switches and Ethernet packets (i.e., frames) have maximum payload size of 1500 bytes. Our application layer message is shipped via SOCK_DGRAM (i.e., UDP) which, in turn, is further packaged in a container (IPv4 in our case) before being handed to Ethernet. UDP adds 8 bytes of its own bookkeeping information, IPv4 adds (by default) 20 bytes of overhead. Since we are adding 1 byte of bookkeeping for binary sequence numbers, the total overhead is 29 bytes. Hence 1471 bytes remains for actual file data. Set each of the 1471 data bytes to be the ASCII character '3' (i.e., value 51). Unless filesize is a multiple of 1471, the last packet sent by myftpd will have a payload smaller than 1500 bytes. To let the receiver, myftp, know that the last packet has been transmitted, we will inscribe in the 1-byte application header the integer value 2 (instead of 0 or 1). As in Problem 4, lab1, without an explicit indication by the sender, the receiver has no way of knowing if the sender is done transmitting. In some cases (e.g., FIFO and SOCK_STREAM), tearing down a connection provides a means to indicate termination. However, doing so is disruptive, inefficient, and faulty, and not to be used unless necessary. Ethernet also has a minimum payload size which will not be a concern for us in Problem 3. As with all system calls, check the return value of sendto() to ensure that it worked as intended.

As to the client's ACK packet, in this version of stop-and-wait, the client will send an ACK with sequence number 0 if it received a data packet from the server with sequence number 0. If the data packet had sequence number 1, the ACK will contain 1. The ACK packet transmitted by the client using sendto() will have a single byte (sequence number) as application layer payload. When the last data packet containing sequence number 2 is received, the client will send an ACK with sequence number 2, close the connection, and terminate. The client will not worry about the possibility that its ACK may be lost and the server may retransmit the last data packet again. And again. And again, ad infinitum. The server, for the last packet only, will retransmit 3 times and give up if an ACK containing sequence number 2 is not received. The issue we are facing is an instance of what is called 2-party consensus (or agreement) problem whose networking implications we will discuss when investigating transport protocols.

3.6 Testing

Perform two types of tests where in type A both myftpd and myftp run on different pod machines in LWSN B148. In type B testing, myftpd runs on a pod machine and myftp runs on an escher machine in HAAS G056. Before running a type A or B test, use ping to measure the RTT between sender and receiver. Set the timeout value of myftpd to, K * avrg-RTT, where avrg-RTT is the approximate average observed from ping measurements. ping is a simple network protocol that sends a probe packet to a receiver which is echoed back which allows the sender to estimate RTT, in addition to checking if the receiver is alive.

(i) *Baseline*. For K = 2, blocksize = 1471, dropwhen = -1, find filesize using the stop-and-wait throughput formula discussed in class that will result in a completion time of 4 seconds (rough ballpark target number) for type B test. avrg-RTT (and hence timeout) for type A test will be significantly less. Use the same filesize parameter for type A and B tests, and compare their performance. Run each test 3 times to get a sense of variability of the measurement results. Discuss your findings in Lab2Answers.pdf.

(ii) *Drops*. For K = 2, blocksize = 1471, and the same filesize as (i), set dropwhen to 10. Repeat the runs of (i). Discuss your findings.

(iii) *Jumping the gun*. For blocksize = 1471, dropwhen = -1, and the same filesize as before, set K = 1.2 and repeat the experiments. Note that K = 1.2 means a 20% slack is added over average RTT which may, when RTT is variable, result in overly aggressive retransmission. Compare the type A and B runs, and discuss your findings.

3.7 Code and result submission

Create subdirectory v3/ and place your code and Lab2Answers.pdf in the directory. Make sure to provide adequate comments/annotation in your code.

Bonus Problem [20 pts]

Use a computing platform other than the Linux machines in our labs to port and carry out part (i) of Problem 3. For example, if you have a laptop running Linux, UNIX, MacOS, or Windows, port the client, myftp, to your system while keeping the server, myftpd, on our lab machines. Porting the client is straightforward whereas porting the server can be to varying degrees more complicated. Chiefly due to asynchronous event handling and timer related issues. The latter can vary even across different versions and flavors of UNIX/Linux. The CS Department provides Windows PCs in its instructional labs where socket programming can be done using Visual Studio. Cygwin is a popular porting option for Windows which provides a POSIX compliant interface and run-time environment. Purdue University operates computing facilities accessible to students which may also be utilized for testing networking code across campus.

If using MacOS, install Xcode which is Apple's IDE environment. Unless you rewrite Linux/UNIX timer code using Darwin's native API, high resolution timer support provided by setitimer() may break or produce unexpected results. For example, even if you set a timer event to be triggered 100 microseconds in the future, the event may be raised significantly later. This will impact timeliness of retransmissions and resultant application throughput under lossy network conditions. Be aware and vigilant when writing or porting network programming code. Please do not use Python, Perl, Java, or other interpreted platforms which can inject significant overhead. We are focused not on getting something to work but understanding how well it works. You may use C++ for the bonus problem if you so choose. Submit your ported code in v4/. Describe your client's computing environment (hardware and software) along with a discussion of the performance results in Lab2Answers.pdf.

The Bonus Problem is completely optional and serves to provide additional exercises to help understand material discussed in class. The bonus points count toward reaching the 45% contribution of the lab component to the course grade.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab2/ where you deposited the submissions and type the command

```
turnin -v -c cs536 -p lab2 lab2
```

This lab is an individual effort. Please note the assignment submission policy specified on the course home page.

Back to the CS 536 web page