CS 536 Fall 2021

Lab 2: Basic Socket Programming and Timed Client/Server Interaction [240 pts]

Due: 10/04/2021 (Mon), 11:59 PM

Objective

The objective of this lab is to practice basic network programming using datagram and stream sockets. We will utilize coarse timers and manage synchronous and asynchronous events to facilitate timed client/server interaction. 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 [120 pts]

Implement an application layer ping client/server using datagram sockets (SOCK_DGRAM) that allows a client, mypingcli, to check if the ping server, mypingsrv, is running on a remote host, as well as estimate RTT (round-trip time). A socket is a type of file descriptor (note that Linux/UNIX distinguishes 7 file types of which sockets is one) which is primarily used for sending data between processes running on different hosts. The hosts may run different operating systems (e.g., Linux, Windows, MacOS) on varied hardware platforms (e.g., x86 and ARM CPUs). Our lab machines run Linux on x86 PCs. Datagram sockets facilitate low overhead communication by not supporting reliability. That is, a message sent from a process running on one host to a process running on a remote host may not be received by the latter. It is up to the application to deal with resultant consequences.

1.1 General background

A socket can be of several types. In this problem, you will use datagram (SOCK_DGRAM) sockets that invoke the UDP (User Datagram Protocol) protocol implemented inside operating systems such as Linux. We will discuss the inner workings of UDP when we study transport protocols. In the same way that FIFOs were utilized as an abstract communication primitive to achieve inter-process communication in Problem 3, lab1, without understanding how FIFOs are actually implemented, we will utilize datagram sockets as an abstract communication primitive to facilitate communication between processes running on different hosts. As noted in class, to identify a process running on an end system (e.g., PC, server, router, smartphone) under the governance of an operating system we need to know how to reach the end system -- IP (Internet Protocol) address of one its network interfaces -- and which process we wish to talk to on the end system specified by a port number. A port number is a 16-bit non-negative integer that is bound to the PID (process ID) of a process to serve as its alias for network communication purposes.

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 feasible. 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 acts both as server and client -- are coded so that they do not depend on specific port numbers to facilitate portability and robustness. We will do the same when possible. 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 an end system. 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 in an internetwork speaking IP. In socket programming, we use IP addresses, not MAC addresses to facilitate communication between processes running on different hosts. IP addresses are translated to MAC addresses by operating systems before delivery over a wired/wireless link. We will study the inner working of IP when discussing network layer protocols. To facilitate human readability of IP addresses, a further layer of abstraction is implemented in the form of domain names. For example, www.cs.purdue.edu is mapped to IPv4 address 128.10.19.120 where the four decimal numbers specify the four byte values of a 32-bit address in human readable form. This translation operation is carried out with the help of a distributed database system called DNS (Domain Name System). Details of DNS, HTTP, SNMP, SSL, and other higher layer protocols are discussed later in the course.

1.2 Implementation details

User interface The ping server, mypingsrv, invokes system call socket() 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 to bind the server to an IP address and an unused port number on its host. Use the Linux command, ifconfig -a, to determine the IPv4 address following the dotted decimal notation assigned to Ethernet interface eth0 on our lab machines (e.g., 128.10.25.213 on pod3-3.cs.purdue.edu in LWSN B148) and provide it as command-line input to your server process along with a port number to use:

% mypingsrv 128.10.25.213 55555

If the specified port number is already being used, bind() will fail. Run mypingsrv again with a different port number.

The client, mypingcli, is executed on a different host with command line arguments that specify the server's coordinate. In addition, the client specifies an IP address of its network interface to use for network communication. For example, amber05.cs.purdue.edu in HAAS G050 is configured with Ethernet interface with IPv4 address 128.10.112.135. Hence running

% mypingcli 128.10.11.135 128.10.25.213 55555

on the client host specifies that the client should use 128.10.11.135 as its IP address and communicate with a ping server at 128.10.25.213:55555. Instead of specifying what port number to use to bind the client process, specify 0 as the port number which delegates the task of finding an unused port number to the operating system.

Operation: client The client, mypingcli, after creating a SOCK_DGRAM socket and binding to an IP address and port number, creates a UDP packet containing a 5 byte payload which it sends to the server using the sendto() system call. The first 4 bytes contain an integer -- message identifier (MID) or sequence number -- that identifies the client request. The value of the fifth byte is a control message that commands the server what to do. If its value is 0 it means that the server should respond immediately by sending a UDP packet to the client with a 4-byte payload whose content is the MID value contained in the client request. If its value is a positive integer D between 1 and 5, it means that the server should delay sending a response by D seconds. If the value of the fifth byte equals 99, then the server should terminate. Before sending a ping request, the client calls gettimeofday() to record the time stamp just before the packet is sent. Upon receiving a response from the server, the client calls gettimeofday() to take a second time stamp. Using the MID value in the response packet, the client calculates RTT by taking the difference of the corresponding send and receive time stamps which is output to stdout in unit of milliseconds (msec).

Upon starting up, the client reads from a configuration file, pingparam.dat, four integer values: N, T, D, S. N is an integer between 1 and 7 that specifies how many packets the client should send to the receiver. If N > 1, then T lying between 1 and 5 specifies how many seconds the client should wait before sending the next request. D is the 1-byte command to be sent to the server, and S is the sequence number (i.e., MID value) of the first packet. The 4-byte MID value of any additional ping packets are incremented by 1. For the last request packet, the client sets an alarm to expire after 10 seconds using alarm(). If a respond does not arrive and the alarm expires, the client ceases waiting and terminates.

Operation: server After binding to an IP address and port number, the server, mypingsrv, calls recvfrom() and blocks on client requests. When a request arrives, the server reads the 5-byte payload and inspects the fifth byte to determine what action is requested. If the command is 0, the server creates a packet containing the 4-byte MID payload and sends it to the client. That is, the server process behaves as an iterative server and performs the task itself. After responding to the client, the server goes back to blocking on recvfrom(). If the command is 99 then it terminates by calling exit(1). If the command is invalid, then the client request is ignored and the server calls recvfrom() to wait for the next request. If the command is an integer value between 1 and 5, then the server process delegates sending a response to the client to a worker process by forking a child. The child process uses sleep() to delay sending a response to the client by 1-5 seconds.

Implement your ping app in a modular fashion and provide a Makefile in v1/ to compile and generate mypingsrv and mypingcli. Create README under v1/ that specifies the files and functions of your code, and a brief description of their roles. When sending and receiving 5-byte messages, note that the x86 Linux PCs in our labs use little endian byte ordering whereas Ethernet uses big endian byte ordering. Test your ping app to verify correctness.

Problem 2 [120 pts]

Modify Problem 3 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 G050 -- and use stream sockets SOCK_STREAM to communicate in place of FIFOs.

2.1 General background

Stream sockets uses TCP (Transmission Control Protocol) which implements sliding window to achieve reliable data communication. Sockets of type SOCK_STREAM export a byte stream abstraction to app programmers where a sequence (or stream) of bytes sent by a sender using write() is received as a sequence of bytes in the same order and without "holes" by the receiver when calling read(). Thus the operating system shields the app from having to deal with the consequences of unreliable communication networks which applies to most real-world networks today. In contrast to SOCK_DGRAM sockets implementing UDP where payload carried by one packet is not part of a stream and data transport is unreliable, SOCK_STREAM sockets maintain a notion of persistent state referred to as a connection between sender and receiver which is needed to implement sliding window. Other aspects of TCP such as congestion control that require a connection between sender and receiver will be covered under transport protocols.

SOCK_STREAM is inherently more overhead prone than SOCK_DGRAM which is reflected in how a connection between sender and receiver is set up before communication can commence using write() and read() system calls. SOCK_STREAM sockets are well-suited for implementing concurrent client/server apps including file and web servers. A server calls socket() to allocate a SOCK_STREAM socket descriptor followed by bind() analogous to SOCK_DGRAM in Problem 1. After bind(), the server calls listen() to mark the socket descriptor as passive, meaning that a server waits on connection requests from clients. The second argument of listen() specifies how many connection requests are allowed pending. For our purposes, 5 will do. Following 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 (the first argument of socket()) is left intact so that it can be re-used to accept further connection requests. The new socket descriptor returned by accept() is called a full association which is a 5-tuple

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

that specifies the protocol type (SOCK_STREAM or TCP), server and client coordinates. The original socket descriptor is called a half association since the client IP and port number remain unspecified so that the descriptor can be re-used to establish a new connection upon calling accept().

On the client side, instead of calling bind() a client calls connect() with the server's IP address and port number. The operating system will fill in the client's IP address and port number with an an unused ephemeral number. If the client is multi-homed and wants to use a specific network interface to send/receive data, or wants to use a specific port number, then bind() can be used to do so. By default, bind() is not needed on the client side due to the actions of connect(). For typical clients, connect() is used which obviates the need to call bind(). When connect() returns, a client can send its request using write() analogous to Problem 3, lab1.

2.2 Implementation details

On the server side, a difference from Problem 3, lab1, is that client requests do not share a common FIFO queue but are transmitted through separate full association SOCK_STREAM sockets. That is, there exists a pairwise connection between a specific client and the shared server. An additional functional feature to add to the server is that upon receiving a client request, it tosses a coin, and if it comes up heads, decides to ignore the request. That is, without forking a child to delegate the task of executing the requested command, the server goes back to blocking on accept() to await the next client request. If the coin toss comes up tails, the server will first inspect the last three bytes of the client's IPv4 address to check that it matches either 128.10.25.* or 128.10.112.* which are addresses of our lab machines in LWSN B148 and HAAS G050. That is, the fourth byte's decimal value should be 128, the third byte 10, and the second byte either 25 or 112. If the match fails, the client request is ignored. Problem 2 implements a remote command server which presents a security vulnerability that must be guarded against. The above check restricts client requests to originate from our lab machines. However, an attacker may forge its source IP address, called spoofing. For additional protection, your server will only allow the legacy apps "date" or "/bin/date" without command-line arguments to be requested for execution. This is prevents a classmate who is unhappy about having lost a tennis match from sending "rm" with suitable arguments to your server. The fact that the source IP address is spoofed is immaterial since damage will have been done. Make sure not call execvp() until you have verified that the protection measures are functioning correctly.

On the client side, since the server is known to discard requests at the application layer despite the data transport protocol implementing sliding window in the operating system, a client sets a timer using alarm() before transmitting a request. Set its value to 2 seconds. If a response arrives before the 2 second timer expires, the SIGALRM signal is cancelled. Otherwise, a signal handler registered for SIGALRM is asynchronously executed. The signal handler closes the connection, establishes a new connection to the server, and sets a fresh 2 second timer before retransmitting the request. This is repeated at most 3 times. If the third attempt fails, the client gives up and outputs a suitable message to stdout.

Implement your client/server app in a modular fashion and provide a Makefile in v2/ to compile and generate rcommandserver.bin and rcommandclient.bin. Execute the client with command-line arguments

% rcommandclient.bin server-IP server-port

that specify the server's IP address and port number. Create README under v2/ that specifies the files and functions of your code, and a brief description of their roles. Test your client/server app with multiple client processes and verify correctness. Even with the protection measures in place, except when testing do not keep the server running.

Bonus problem [20 pts]

Compare the RTT estimates from your application layer ping in Problem 1 against those of /bin/ping. Perform tests between machines in the same lab (e.g., between pod machines in LWSN B148), and across

labs (pod machine in LWSN B148 and amber machine in HAAS G050). What results do you find? Give your interpretation of the findings in lab2.pdf and place it under lab2/.

The Bonus Problem is completely optional. It serves to provide additional exercises to understand material. Bonus problems help more readily reach the 45% contributed by 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 individual effort. Please note the assignment submission policy specified on the course home page.

Back to the CS 536 web page