CS 422 Fall 2014

Lab 3: Concurrent TCP Server Design and UDP Tunneling

Due: 10/19/2014 (Sun), 11:59 PM

Objective

The objective of this lab is to implement a TCP-based concurrent server and investigate the use of tunneling in network services.

Reading

Read chapters 13-16 from Comer (textbook).

Problems [280 pts]

Problem 1 (60 pts)

Perform ping to the following web sites from one of our lab machines: www.ucsd.edu, www.indiana.edu, www.mit.edu, www.brandeis.edu, www.ufl.edu, www.tsinghua.edu.cn. Using google (or any other method that you prefer) estimate the ballpark distance from West Lafayette, IN, to these sites. Divide distance by SOL (speed-of-light) to get an estimate of the latency or delay needed to send a single bit. Compare the latter with the results obtained from ping. What do you find, and how would you explain the difference? Brandeis and MIT are both located in the Boston area. Why might their ping values be significantly different? To help explain, use the traceroute utility to uncover the routers traversed when packets are sent to www.brandeis.edu and www.mit.edu, respectively.

By default, ping sends out a packet with payload size 56 bytes (your myping app used a default size of 1024). With the -s option, payload size can be specified as a command-line argument. Using www.indiana.edu as the target destination, perform ping with payload sizes 500, 1000, 1500, 2000, 2500, 3000, 3500. What do you find, and how would you explain the results? Perform a ping with payload size 1500 from a laptop in LWSN to www.indiana.edu and compare with the 1500 payload result obtained from the lab machine. How do you explain the difference? If you need access to a laptop, check with our TA during the PSOs.

Record your traces/results when running ping and traceroute using the script utility. Submit your results and answers in lab3writeup1.pdf under mylab3/p1.

Problem 2 (100 pts)

A variant of Problem 3, Lab 2, rewrite the UDP-based concurrent ping client/server app so that is uses TCP (SOCK_STREAM) instead of UDP (SOCK_DGRAM) when creating a socket description using socket(). As noted in class, a key difference of TCP is that is provides reliable communication using a form of retransmission (i.e., ARQ). Thus an application programmer need not be concerned about checking whether a message gets delivered or not, as long as the other party is connected and reachable via an internetwork running TCP/IP. In Problem 2 of Lab 2, because we used the lightweight UDP protocol, the app itself needed to deal with potential missing or corrupted reminder messages.

Two other important features to note about TCP as a service exported by operating systems: one, the programmer is provided with an abstraction of data communication where the unit of communication is not individual packets (as in UDP) but a contiguous stream of bytes (hence the term "stream"). For example, as in a regular file that is part of a disk-based file system where the fact that disks are block devices accessed in units of blocks by a kernel is hidden from read()/write() system calls, the same holds for communicating bytes using TCP sockets. Hence read()/write() is used in place of recvfrom()/sendto(). Two, the reliability feature and stream of bytes abstraction comes at the cost of significant overhead and protocol statefulness, which are TCP's main drawbacks. This is reflected in TCP-based servers programmed using the following sequence of system calls: socket(), bind(), listen(), accept(), read()/write(). socket() and bind() perform the same functionality as in UDP, and read()/write() replace recvfrom()/sendto().

listen() specifies to a kernel that the app is a server that passively awaits client requests and the second argument of listen() (typically set to 5) specifies how many client connection requests are buffered while a server is handling the current request. accept() is a blocking call that waits for a client request. It is important to note that when accept() returns, it returns the socket descriptor of a newly created socket (i.e., not the one returned by socket()) whose 5-tuple values TCP, server IP, server port, client IP, client port have been filled in. That is, as discussed in class, a full association has been established. The original socket descriptor returned by socket() remains as is, a half association that specifies TCP and server IP/server port through bind(), so that it can be used to listen to new client connection requests. In the concurrent ping client/server app of Problem 3, Lab 2, forking a worker process is performed after accept() returns.

At the client side, statefulness is reflected in the client process calling connect() after socket(). bind() is optional in the sense that leaving it out (as in the UDP case) asks the kernel to select a default IP enabled network interface and ephemeral port number on behalf of the calling process. connect() establishes a stateful reliable communication channel with a server (if successful), i.e., a full association with a specific server only. The client cannot use the same socket descriptor returned by socket() to send data to other servers (as is possible in UDP). Thus whereas at the server side accept() returns a new socket descriptor each time a full association is established so that the original socket descriptor can be used to listen to new connect() requests, at the client side the original socket descriptor returned by socket() is committed to a single server (and resultant full association) by connect().

Rewrite the ping client/server app of Problem 3, Lab 2, using TCP and verify that it works correctly.

Use the same payload size as in the UDP case. Perform side-by-side comparisons with your UDP-based ping app and note their performance with respect to the response times reported by the ping client. Repeat the ping experiments 10 times to get an average value. Report the results in lab3writeup2.pdf and discuss your findings. Submit the code and write-up in mylab3/p2.

Problem 3 (120 pts)

Tunneling is a popular technique used in network services including virtual private networks (VPNs) where the aim is to affect logical or virtual services that do not necessarily correspond to how the actual network system is architected. For example, some network service sites check client IP addresses, and if they originate from regions/countries deemed potentially risky, the requests are ignored by a server or filtered at routers connecting the server (e.g., firewalls). A common way to by-pass such cient or source IP-based filtering (by friend or foe) is to use tunneling where a client A that aims to interact with server B sends its packets through an intermediary server C. The intermediary C who has an IP address that is not filtered by server B then forwards A's request to B (after a bit of processing), making it seem to B that the request comes from C. The underlying technique is called tunneling as the client request from A to B is "tunneled" through C. The response from server B to C is then tunneled back to A who is the actual (and hidden) client of B. Servers that provide a measure of anonymity to clients also use this technique, among a number of other applications. In the mobile reminder client/server app of Problem 1, Lab 2, tunneling may be used to hide from the server that the client is mobile (i.e., changing its IP address and port number over time). This may help simplify the mobilereminderd server design by reducing the need to maintain mobility information.

Your tunneling server, vpntunneld, will work as follows. It takes two command-line arguments

% vpntunneld vpn-port-number

where vpn-port-number specifies which port the UDP-based vpntunneld expects new VPN client requests. The client, vpntunnel, sends a UDP packet to vpntunneld at a well-known IP address vpn-IP and port number vpn-port,

% vpntunnel vpn-IP vpn-port server-IP server-port-number secret-key

and the actual server IP and port number that a client app wants to interact with. Note that vpntunneld is completely open in that it does not perform an access control check. vpntunneld returns an UDP ACK packet that contains a second vpn-port-number the client should use to send UDP payload packets directed at the real server (e.g., mobilereminderd). Thus the first vpn-port-number is used to establish a VPN client/server contract (i.e., control signaling through management packets) and it is the second vpn-port-number that is used for tunneling of actual messages to the real server (e.g., mobilereminderd).

In a production VPN tunneling system, the VPN client vpntunnel would be installed with root privilege and kernel access so that the legacy client app, say, mobilereminder is run

% mobilereminder server-IP-address server-port-number secret-key

exactly as before. That is, the command-line arguments server-IP-address and server-port-number are that of its true server, mobilereminderd. Hence tunneling preserves backward compatibility for both the server and client sides of a legacy client/server app. Whether in Linux/UNIX or Windows, kernel modules can be installed that inspect/intercept all packets generated by mobilereminder and the server-IP-address and server-port-number replaced with vpn-IP and vpn-port so that the packets are sent to vpntunneld. In order for vpntunneld to know where to tunnel the received packets, server-IP-address and server-port-number are inserted into the payload of the UDP packet by the kernel module at the client machine running mobilereminder. vpntunneld removes the extraneous information from the UDP payload and forwards the packets to server-IP-address/server-port-number (i.e., mobilereminderd). Since the packets reaching mobilereminderd have vpntunneld's IP address and port number as their source, it will accept them and send a response to vpntunneld which forwards the packet to mobilereminder. vpntunneld remembers the IP address/port number of mobilereminder by noting during the initial registration by vpntunnel.

Since we do not have root privilege/kernel access to the lab machines in LWSN B158 (they are shared with other courses), we will have to be satisfied with almost-client-side-backward-compatibility in the sense that the mobilereminder's binary remains as is but its command-line arguments must specify vpn-ip and vpn-port in place of server-IP-address and server-port-number. The vpn-port number to be used is the one returned in the ACK packet by vpntunneld. Hence the client, vpntunnel, prints the received port number on stdout so that a human user can use it as the second argument to mobilereminder. When mobilereminder changes its IP address/port number (i.e., you move from one lab machine to another), vpntunnel needs to be rerun which results in a new port number returned to vpntunneld to use by mobilereminder. Thus we achieve complete transparency with respect to server mobilereminderd and binary backward compatibility with respect to mobilereminder, but vpntunneld and vpntunnel must contiually coordinate when a client's IP address/port number changes.

Write the VPN tunneling vpntunnel/vpntunneld client/server app and submit their code in mylab3/p3. During testing, move between lab machines as before and if the tunneling system is functioning correctly, the same reminder results as in the tests of Problem 1, Lab 2, should result. As always, make sure to provide adequate comments in your code.

Bonus Problem (40 pts)

Look into how you might be able to implement the packet inspection/intercept functionality as part of the vpntunnel client with kernel support in Linux and Windows. Note that both kernels are layered and allow installing code modules within their protocol stack. Sketch solutions for Linux and Windows and submit as lab3writeup4.pdf in mylab3/p4.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Create a directory named mylab3 under which the subdirectories p1, p2, p3 (and p4) and code submissions are organized. Go to the parent

directory of mylab3 and type the command

turnin -c cs422 -p lab3 mylab3

Please note the assignment submission policy specified in the course home page.

Back to the CS 422 web page