CS 536 Fall 2019

Lab 5: Network Tunneling and Overlay Networks [300 pts]

Due: 11/13/2019 (Wed), 11:59 PM

Objective

In this lab, we will investigate network tunneling, a technique that underlies VPNs and overlay networks through which limited user control over packet forwarding and routing can be exercised. This, in turn, facilitates creation of virtual network structures over physical networks and security.

Reading

Read chapters 5 and 6 from Peterson & Davie.

Problem 1 (150 pts)

1.1 Motivation

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 client or source IP-based filtering 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, making it seem to B that the request came 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, network address translation and port mapping also use this technique among other applications.

1.2 Establishing a tunnel

Your tunneling server supergopher

% supergopher vpn-port

takes command-line argument vpn-port that specifies on which port the UDP-based supergopher expects new VPN client requests. The tunneling client, minigopher, sends a UDP packet to supergopher at a well-known IP address vpn-IP and port number vpn-port

% minigopher vpn-IP vpn-port server-IP server-port-number

The third and fourth command-line arguments, server-IP and server-port-number, specify the coordinates of the real server that the client is aiming to contact. The message format (i.e., UDP payload) is server-IP followed by server-port-number (6 bytes total). supergopher returns an UDP ACK packet whose first byte, ANS, if of value

3, indicates that the request has been accepted. The second and third bytes of the UDP payload contain a port number, transit-port, that the client should use to send its UDP packets to supergopher at vpn-IP at port transit-port which the tunneling server will then forward to the real server at server-IP:server-port-number. If ANS is any other value, the request is assumed to have been rejected and an appropriate message is output to stdout. Output transit-port to stdout so that the user can use its value in 1.3.

1.3 Using an established tunnel

In the case of the talk application in Problem 2, lab4, after terve binds to a local port (cli-port) at a local IP address (cli-IP) a user would issue command

#ready: vpn-IP transit-port

as if the user were to establish a talk session with vpn-IP:transit-port (i.e., not server-IP:server-port-number). As a consequence of the interaction between supergopher and minigopher that established a tunnel (i.e., virtual channel), supergopher at IP address vpn-IP forwards UDP packets received on port transit-port from its tunneling client cli-IP:cli-port to IP address server-IP at port server-port-number. supergopher uses an unused port number, transit-port-2, as its source port when doing so. Conversely, when supergopher receives a UDP packet from server-IP:server-port-number on port transit-port-2, it forwards the payload to the tunneling client at cli-IP:cli-port.

In a production VPN tunneling system, the VPN client minigopher would be installed with kernel support so that a legacy client app can be run exactly as before to achieve backward compatibility and user transparency. That is, in the case of the chat application terve the command issued by the user would be

#ready: server-IP server-port-number

Whether in Linux, UNIX or Windows, minigopher's kernel component would intercept all UDP packets generated by user apps, encapsulate the UDP packet as payload of another UDP packet destined to supergopher (called UDP-over-UDP) so that forwarding is performed transparent to user and application. That is, user and app are both oblivious to the fact that traffic is being tunneled through supergopher. The same goes for TCP. Since we do not have kernel access to the shared lab machines, we will make do with the not-fully-transparent-to-the-user-and-app version which is coded entirely in user space.

1.4 Tunneling server forwarding operation

supergopher, upon accepting a request from minigopher for a virtual channel, inserts in its forwarding table a 7-tuple entry for socket-index, transit-port, transit-port-2, cli-IP, cli-port, server-IP, server-port. socket-index is an index to a 1-D array of socket descriptors to which each distinct transit-port is bound. Use MAXSOCKIND as an upper bound on the number of sockets supported by supergopher. For our purpose, define MAXSOCKIND as 10. cli-port is filled with a dummy value until the first data packet from cli-IP on transit-port is received at which time it is resolved. When a UDP packet on transit-port arrives (recvfrom()), it verifies that the source IP address matches cli-IP and forwards (sendto()) the payload to server-IP:server-port with source address vpn-IP:transit-port-2. If the source address does not match, the packet is discarded. Conversely, if a UDP packet from server-IP:server-port arrives on port transit-port-2, supergopher forwards the payload to address cli-IP:cli-port with source address vpn-IP:transit-port. To do so, a second entry for the same 7-tuple albeit with a different socket-index is inserted in the forwarding table so that packets arriving on transit-port-2 can be processed. Use even socket indices to represent packet flow from client to server, and odd indices for packets from server to client. Implement the forwarding operation using select() to facilitate monitoring of multiple sockets.

1.5 Testing

Perform basic testing (part (a) of 2.4 in lab4) of the terve app and verify that the app works correctly when its traffic is mediated by tunneling. One machine runs supergopher, a second machine minigopher and terve, and a third machine terve. Define a debug parameter, #define TABLEUPDATE, in header file supergopher.h so that when supergopher updates its look-up table, the update values are output to facilitate debugging and tracing of application behavior. Extend basic testing to concurrent tunneling sessions (i.e., multiple chat sessions) serviced by the same tunneling server supergopher running on a lab machine. Submit your code and Makefile in v1/. Make sure to annotate your code adequately.

Problem 2 (150 pts)

2.1 Motivation

In this problem, we will consider routing in overlay networks that allows users on IP internetworks to influence the paths packets may take. In general, users (i.e., end systems) cannot affect network routing which determine the path a packet takes. If, for example, a company is a customer of an ISP X and wishes to send data from an IP device A in X to an IP device B in ISP Y (e.g., B may be part of a branch office), then network routing determines the path a packet takes from X and Y. The path may include routers/switches within X and Y, and a third ISP Z that provides connectivity to ISPs X and Y. If Z's routers are located in an "unfriendly" region and the company is concerned about security, in general, the company's only recourse is to encrypt its flow. Using overlay network routing, the company may be able to circumvent ISP Z by employing one or more IP devices located in a fourth ISP T. A's traffic is sent to an IP device C located in T which then forwards the traffic to B located in Y. C is called an overlay router. Assuming the paths from A to C, and C to B, do not entail traversing through routers belonging to Z, the company has succeeded in by-passing Z. In general, to ensure that ISP Z is not traversed, it may be necessary to recruit more than one forwarding IP device across multiple ISPs.

Another example of overlay network routing is anonymizing sources. A client that does not wish to reveal its location may employ multiple forwarding nodes to hide its identity. Although real-time tracking while a flow is on-going is feasible, multi-hop forwarding increases the effort expended to backtrack the source (i.e., traceback). Overlay networks allow arbitrary logical network topologies to be embedded in physical network topologies, a form of virtualization that exports customer tailored views of a shared network which can be useful when structuring distributed computing services.

2.2 Overlay network establishment

Overlay message propagation To achieve overlay network routing, we will build on the tunneling implementation of Problem 1. As in the tunneling server, the overlay network router, overlaygopher, takes a command-line argument

% overlaygopher overlay-port

where overlay-port is a well-known port number on which it waits for client requests. A client request is transmitted by running createoverlay

% createoverlay router1-IP router1-port router2-IP router2-port ... routerk-IP routerk-port dst-IP dst-port

where the command-line arguments specify the IP addresses and port numbers of k overlay routers packets should traverse before reaching the final destination dst-IP:dst-port. Before executing createoverlay at a source, overlaygopher must be run at k hosts in our labs (pod and escher machines) given by the k router IP addresses and bound to the k ports. createoverlay, when executed at a source with IP address src-IP and port number src-port, sends a UDP packet to router1-IP at port router1-port whose payload has the format

k#router1-IP#router1-port#router2-IP#router2-port# ... #routerk-IP#routerk-port#dst-IP#dst-port

where k (a 1-byte unsigned value) specifies the number overlay routers, character '#' is a separator symbol, the IP addresses are 4-byte IPv4 addresses, and port numbers occupy 2 bytes. overlaygopher running at router1-IP:router1-port returns a UDP ACK following the same format as the ACK of supergopher in Problem 1. The transit-port number returned by router1-IP:router1-port is used by the source src-IP:src-port to send UDP app packets to router1-IP:transit-port. createoverlay outputs transit-port to stdout so that a user can use it when running its app (e.g., terve).

Overlay router router1-IP:transit-port strips itself from the received message and transmits the modified overlay establishment packet

k-1#router2-IP#router2-port# ... #routerk-IP#routerk-port#dst-IP#dst-port

to router2-IP:router2-port. router2-IP:router2-port, in turn, returns a UDP ACK with transit-port specifying the port number that router1-IP:transit-port should send forwarded packets from src-IP:src-port to. That is, router2-IP:transit-port. This iterative process is repeated until the k'th overlay router is reached. routerk-IP:routerk-port, upon receiving message

1#routerk-IP#routerk-port#dst-IP#dst-port

sends back a UDP ACK with transit-port that the preceding overlay router should forward UDP app packets to.

Forwarding table update An overlay router inserts in its forwarding table a 5-tuple

socket-index pre-IP pre-port post-IP post-port

where socket-index is an index into a socket descriptor array whose descriptor is associated with pre-port which represents the previous router's (or client's) port number to transmit packets to in the reverse direction (i.e., from server to client). pre-IP is the IP address of the preceding overlay router (or source src-IP in the case of the first overlay router). post-IP is the next overlay router (or dst-IP in the case of the last overlay router) and post-port the transit-port returned by post-IP. We need not inscribe the router's transit-port in the forwarding table since a socket descriptor specific to transit-port has been allocated and bound. As in Problem 1, use even socket indices to represent packet flow from client to server, and odd indices for packets from server to client. Implement the forwarding operation using select() to facilitate monitoring of multiple sockets.

2.3 Overlay forwarding operation

In the case of our chat application, after terve binds to a local port (src-port) at a local IP address (src-IP) a user would issue command

#ready: router1-IP transit-port

as if the user were to establish a talk session with router1-IP:transit-port where transit-port is the port number returned by router1-IP to createoverlay at src-IP. overlaygopher at router1-IP, upon look-up of its forwarding table, knows to forward the packet to post-IP:post-port based on matching with pre-IP:pre-port. If the source IP address of the incoming packet does not match pre-IP, the packet is dropped. The same operation is carried out at subsequent overlay routers until the destination dst-IP:dst-port is reached. UDP packets traveling in the opposite direction from dst-IP are forwarded analogously, albeit with the roles of pre-IP:pre-port and post-IP:post-port reversed. Use select() to facilitate monitoring of multiple sockets.

2.4 Testing

Perform basic testing of the terve app as in Problem 1 with one and five overlay routers spanning both pod and escher machines acting as intermediaries. Verify that the chat app works correctly. Extend basic testing to

concurrent overlay sessions (i.e., multiple chat sessions) serviced by overlapping overlay routers. For example, one of the overlay routers in the 5-overlay router configuration may be shared by two separate overlays. As in Problem 1, define a debug parameter, #define TABLEUPDATE, in header file overlaygopher.h so that when overlaygopher updates its look-up table, the update values are output to facilitate tracking of application behavior. Submit your code and Makefile in v2/. Make sure to annotate your code adequately.

Bonus Problem (20 pts)

One issue that is present in Problem 2, but not in Problem 1, is that an ACK returned by router1-IP with transit-port does not imply that a overlay path from src-IP:src-port to dst-IP:dst-port has been successfully established. That is, any one of the k overlay routers may be unreachable due to network or end system issues in which case the table entries of preceding overlay routers should be removed. Hence until the last overlay router successfully updates its table, all table updates in preceding overlay routers are to be considered tentative. These soft router states must be confirmed by a control message propagating back from the last overlay router to the first overlay router and the host executing createoverlay. Similarly, when an established overlay network is not needed anymore, its soft state must be updated so that look-up tables are accurate and compact. Extend the protocol of Problem 2 so that the last overlay router confirms successful overlay establishment to the preceding overlay router which propagates updates in the reverse direction until the first overlay router and the source are reached. Describe in Lab5Answers.pdf your design, implement and evaluate its correctness. For example, by not running overlaygopher on an overlay router specified in the path of createoverlay, we are assured that overlay network establishment will fail. Output the relevant reverse updates so that correct operation can be confirmed. Submit your code and Lab5Answers.pdf in v3/.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab5/ where you deposited the submissions and type the command

turnin -v -c cs536 -p lab5 lab5

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