CS 536 Fall 2021

Lab 4: Enhancements to Transport Layer Protocols [240 pts]

Due: 11/1/2021 (Mon.), 11:59 PM

Objective

The first objective of this lab is to implement a performance oriented but lightweight reliable file transport protocol. The second objective is to facilitate improved client authentication in the framework of public-key and symmetric key cryptosystems by utilizing basic crytographic primitives.

Reading

Read chapters 4 and 5 from Peterson & Davie (textbook).

Problem 1 [180 pts]

1.1 Basic operation

We will implement a custom reliable file transfer protocol, rrunner (roadrunner), aimed at improving performance compared to TCP based and tftp file transfer apps under low loss network conditions. tftp is popular in LAN environments for transporting system files. It uses simple stop-and-wait to achieve reliability. Our objective is to increase throughput over tftp by employing a form of sliding window to transmit multiple packets at a time, however, without incurring the overhead in complexity and ACK traffic of full-fledged sliding window protocols. The key element of rrunner is a fixed window size W in unit of UDP packets. The sender, i.e., file server rrunners, transmits W data packets -- unless end of file is reached -- where each packet is marked by its sequence number 0, 1, ..., W-1. The actual sequence number space is twice as large and discussed in Section 1.3. The receiver, i.e., client rrunnerc, sends a UDP ACK packet only when all W packets have been received. The server then sends the next W data packets. Before transmitting the data packets, the server sets a timer. If it expires before an ACK is received, all W data packets are retransmitted.

1.2 App interface

We will adopt the app interface specification, system parameters, and file I/O component of the TCP based file transfer application myftpc/myftps of Problem 2, lab3. The client is executed with command-line arguments

```
% rrunnerc server-IP server-port filename secret-key blocksize windowsize
```

where windowsize specifies W. The other arguments are the same as for myftpc. The file server is executed with command-line arguments

```
% rrunners server-IP server-port secret-key blocksize windowsize timeout
```

where the windowsize must be the same value as in the client. Parameter blocksize determines the unit of read()/write() when performing file I/O. Since roadrunner uses UDP, blocksize specifies the number of bytes sent/received by sendto()/recvfrom() system calls. The last argument, timeout, specifies in unit of microseconds how long the server will wait before W data packets are retransmitted. Use the setitimer() system call of type ITIMER_REAL to set a high resolution timer to timeout in a one-shot (i.e., not periodic) fashion. Use ping or myping to estimate RTT based on which a suitable timeout value is set. We do not want timeout to be too close to RTT since statistical variability may trigger premature timeout causing unnecessary retransmission. On the other hand, we do not want timeout to exceed RTT significantly since it leads to wasted time and resultant decrease of throughput. System parameters are typically specified in configuration files or as program constants. We will continue to use command-line arguments for clarity and configurability during testing and performance evaluation. We will adopt the same filename and secret-key convention as Problem 2, lab3. However, we will omit the server throwing a coin to determine if a request should be ignored. That is, all received file transfer requests are honored. The client will still need to set a timer (500 msec) to retransmit a request in case UDP fails to deliver the client's request.

1.3 Implementation details

We will limit windowsize, i.e., W, to 0-63. With blocksize 1024 (bytes) and RTT 2 msec in an extended LAN environment, the maximum throughput achievable by roadrunner is at least 1024 * 64 * 8 * 500 = 262 Mbps. This ignores 28 bytes contributed by UDP and IPv4 headers, and at least 18 bytes contributed by Ethernet from source/destination address and CRC fields. In the opposite direction, the numbers ignore operating system, file I/O, and other software overhead. During testing we must keep windowsize well below 64 since with RTT smaller than 2 msec the file transfer app is easily capable to exceeding 1 Gbps. Since our lab machines are a shared resource and not a dedicated testbed, we will test W values to compare against the performance of myftpc/myftps, not the maximum throughput achievable by roadrunner.

The first byte of every data packet transmitted by sendto() will specify the packet's sequence number 0, ..., W-1. By the relationship between sequence number space and maximum window size in sliding window protocols, we know that sequence number space must be at least twice as large as maximum window size to correctly handle retransmitted packets. Hence the actual sequence number space is 0, ..., 2*W - 1. Hence, sequence number W represents the first data packet of the next window of W packets. For example, in the case of three consecutive windows without retransmission, the sequence numbers of the 3*W data packets will be: 0, ..., W-1; W, ..., 2*W-1; 0, ..., W-1. When the client receives all packets of a window, it sends a UDP ACK packet containing a 1-byte payload with value W-1 or 2*W-1 depending on the sequence number of the last packet received. Upon receiving an ACK, if the timer has not expired then the timer is cancelled and the next W data packets are transmitted after setting a fresh timeout. If the timer has expired, then any on-going retransmission is ceased and the newly set timer prior to retransmission is cancelled. Due to the 1-byte sequence number, a data packet transmitted by sendto() is blocksize + 1 bytes long. We will limit blocksize to less than or equal 1471 bytes so that the total payload of an Ethernet frame carrying IPv4 and UDP packets does not exceed 1500 bytes. When comparing roadrunner performance against that of myftpc/myftps, use blocksize+1 as the latter's blocksize value.

The server, rrunners, needs to convey to its client that end of file has been reached. This is done in one of two ways. First, if the size of the file being transferred is not a multiple of blocksize bytes, the last data packet having payload size less than blocksize + 1 implies that it is the last data packet. In the case where file size is a multiple of blocksize, the server sends a packet of size blocksize + 2 bytes where the last byte serves as a dummy padding byte. If the recvfrom() at the client returns blocksize + 2 bytes, it strips off the last byte and knows that file transmission has ended. The client then transmits 8 duplicate ACK packets in the hope that one of them will reach the server so that it can terminate. This is obviously a hack whose theoretical roots we discussed in class. The server, upon (hopefully) receiving an ACK of the last data packet, terminates. In general, the server rrunners should follow a concurrent server structure where a child is forked to handle the transfer of data. For this problem, you may use an iterative server structure where a single client is handled.

1.4 Testing and performance evaluation

For roadrunner blocksize values 512, 1024, 1471 bytes, test that windowsize = 1 (i.e., plain stop-and-wait similar to tftp) works correctly. Compare throughput and completion time of large and small files benchmarked in Problem 2, lab2, between roadrunner and myftpc/myftps. For blocksize 1024, increase windowsize gradually to verify correctness of roadrunner and evaluate performance compared to W = 1. Stop increasing windowsize if completion time exceeds 4 seconds. As noted in class, for large files typical completion times should be in the 2-3 second range. Discuss your results in lab4.pdf. Implement roadrunner in a modular fashion and provide a Makefile in v1/. Include README under v1/ that specifies the files and functions of your code, and a brief description of their roles. Remove large files created for testing after completion.

Problem 2 [60 pts]

2.1 General background

In Problem 1 we have used an insecure way -- sending a secret key in the clear -- to authenticate a client. Such methods were the norm well into the 1990s before the adoption of cryptographic primitives in everyday network protocols such as supplanting telnet with ssh. Authentication is facilitated by basic cryptographic primitives which can also be used for implementing confidentiality (i.e., encryption) and integrity (i.e., message has not been modified by an attacker). The underlying mechanisms are one and the same. Cryptographic primitives used in network protocols rely on a message encoding function E, a decoding function D, and two distinct keys, e and d, called public and private keys, respectively. These systems are referred to as asymmetric or public key cryptographic systems. Given a message m (a bit string), called plaintext, that Alice wishes to send Bob, confidentiality that protects m from eavesdroppers works as follows:

Confidentiality:
(a) Alice computes encrypted message, s = E(m,e), using m and Bob's public key e. B's public key, as the name indicates, is assumed known to all parties who wish to send secret messages to B.
(b) Bob receives s (e.g., payload of Ethernet frame, UDP or TCP packet), then computes m = D(s,d) which yields the original unencrypted message m.
(c) It is assumed that without knowing B's private key d, computing m from s is computationally difficult.

D can be viewed as an inverse of E, and D is a "one-way function" in the sense that encrypting -- going in one direction -- is computationally easy but decrypting (i.e., inverting E) without knowing B's private key (which we assume he safeguards) is computationally hard. That is, unless P = NP. For authentication, the same primitives are employed but in reverse. That is, Bob wishes to send a message (called certificate) s to Alice whereby Alice, upon receiving s, can determine that Bob is the originator of s.

Authentication:
(a) Bob computes certificate, s = D(m,d), where m is a plaintext message that says "I am Bob, born around 1905, my favorite color is blue, gettimeofday() is ... and I want you to send me the requested file" using his private key d.
(b) Alice, upon receiving s, computes m = E(s,e), which yields the original plaintext message which follows a strict format (e.g., name, birthday, favorite color, time stamp, etc.).
(c) It is assumed that only B can generate a certificate s using his private key d, that when encrypted via E using B's public key e results in a meaningful plaintext message that follows a specific format.

Hence the two functions E and D commute in the sense that either order of function composition yields the original (plaintext) message as they act as inverses of each other. The popular RSA public key cryptosystem that relies on the assumption that factoring large primes is computationally hard yields such E and D.

2.2 Client authentication: public key infrastructure

A central issue of using a public key cryptosystem for authentication and confidentiality is bootstrapping, since Alice knowing Bob's public key is easier said than done. For example, if an impostor C somehow convinces A that a key that C created, say e*, is B's public key, all bets are off. So how do we bootstrap a distributed system where the true public keys of all relevant parties are accessible? Unfortunately, there is no solution to the bootstrapping problem but for brute-force system initialization where designated "trusted" parties, called certificate authority (CA), serve as arbiters of verifying public keys. Operating systems are distributed with hardcoded public keys of CAs so that a user of the operating system can make use of their services. For example, if a secure communication session over UDP or TCP (e.g., user data sent as UDP or TCP payload is encrypted before transmission) to a server is desired, the server's address -- the symbolic domain name of an IP address, say, www.cs.purdue.edu in place of 128.10.19.120 -- is transmitted to the CA (encrypted with the CA's public key). The CA then responds with the public key of the server as a signed certificate (i.e., step (a) of Authentication) which the client can verify for authenticity by applying E with the CA's public key (step (b) of Authentication). To reduce overhead, servers may obtain pre-signed CA certificates which are communicated to clients directly.

When engaging in lengthy data exchanges (e.g., file server responding to client requests), a public key infrastructure (PKI) is only used to establish mutual authentication and share a secret private key, after which symmetric encryption using the shared private key is used to achieve data confidentiality. This is due to symmetric encryption being more efficient. A popular symmetric encryption method is one-time pad which XORs data bits with a random sequence (obtained from private key). To date, one-time pad is the only provably secure encryption method assuming that the random sequence is indeed random and the private key is known to the communicating parties only. In the real world, pseudo-random sequences take the place of random sequences, hence one-time pads are only as secure as the randomness of pseudo-random sequences generated from private keys. As John von Neumann noted: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

2.3 Client authentication: implementation

When building network applications using socket programming, TLS (Transport Layer Security) and its precursor SSL (Secure Sockets Layer) using the OpenSSL API is the default way to implement crytographic protocols. Unlike TCP/UDP/IP socket programming which is narrow in scope and supported by select system calls of an operating system, SSL is more complex due to its generality stemming from the diversity and richness of context dependent security protocols. SSL programming is the subject of a course in network security and outside the scope of our course. We will, however, implement a limited form of cryptographic security by replacing the simple secret-key/password method of authenticating the client in Problem 1 with a more secure method following the cryptographic framework of 2.2.

We will assume that our file server maintains an access control list (ACL) of IP addresses (in dotted decimal form) and associated public keys. A client sends a request in the form of a certificate signed using function D and the client's private key. The server, upon receiving a request, applies E with the client's public key assuming its IP address is contained in the ACL. If the result matches a certain format, the requested file is transmitted. Otherwise the request is dropped. From a network programming perspective, the internals of E and D are not relevant since they can be treated as black boxes. In place of emulating RSA (or other cryptographic algorithms), we will define simplified black boxes E and D as follows.

Decoding function D:
The decoding function, unsigned int bbdecode(unsigned int x, unsigned int prikey), takes unsigned int x and private key prikey as input and returns an unsigned int value which will be used by the client to authenticate itself to the server. bbdecode() works by performing bit-wise XOR of the 32 bits of x and prikey. The value x will be the client IPv4 address viewed as a 32-bit unsigned integer. The 4-byte unsigned integer returned by bbdecode() will be sent in place of the 2-byte secret-key in the request packet to the server. Hence the length of the request packet increases by 2 bytes.

Encoding function E:
The encoding function, unsigned int bbencode(unsigned int y, unsigned int pubkey), takes unsigned int y and public key pubkey as input. pubkey is the public key associated with the IPv4 address of a client that sent a request. The server looks up the public key pubkey associated with the client's IPv4 address. The first four bytes of the request packet when viewed as unsigned int will be used as y to compute bbencode() which takes the bit-wise XOR of y and pubkey. If the value returned by bbencode() is the client's IPv4 address viewed as an unsigned int, the server will consider the client authenticated. Note that little endian/big endian conversion must be implemented to ensure that bbencode() and bbdecode() computation yield correct results. Since XOR is its own inverse, E and D will commute and satisfy our needs.

Using the certificate computed by D, y, is subject to replay attack by an adversary who may record y through traffic sniffing, then reuse it to authenticate itself as a valid client to download a file. This can be mitigated by extending the certificate to include sequence numbers and timestamps that do not remain static. We will omit this part in this exercise. Implement the modified roadrunner and place the code in v2/. Create a file, acl.dat, that contains entries comprised of IP address (in dotted decimal form) and public key associated with the IP address (to be read in as unsigned integer). Use three entries of lab machine IPv4 addresses for testing. Continue to use secret-key in rrunnerc's command-line argument as its private key value. Remove the secret-key in rrunners's command-line argument since it is not needed. Test and verify that roadrunner with improved authentication support works correctly. Check if there is any degradation of throughput due to performing encryption/decryption

---

Bonus problem [20 pts]

Encrypt the file content sent to the client by applying E at the server byte-by-byte using the client's public key. The client decrypts the received content by applying D with its private key byte-byte-byte. Since E and D are defined to operate on unsigned integers, promote each byte of the requested file from char to unsigned int before computing E and D. Use the least significant byte of the computed unsigned int as the output of E and D for encryption/decryption purposes. Place your code in v3/. Test and verify that the modified roadrunner app of Problem 2 works correctly. Check if there is degradation of throughput from performing encryption/decryption operations to facilitate confidentiality.

Turn-in Instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Place lab4.pdf under lab4/. Go to the parent directory of the directory lab4/ and type the command

```
turnin -v -c cs536 -p lab4 lab4
```

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