CS 270 Systems Programming Spring 2021

Project 3: Client-Server Datagram Protocol

***Out**: Wed 24 March 2021* ***Due**: Monday 12 April 2021*

This is an **individual project**. You must do this assignment on your CS Department VM in order to get credit. Most of the information needed to do this assignment is provided below. Some hints may be provided via the Canvas page for this assignment. A **rubric** will also be provided via Canvas.

1 Overview

The main goal of this project is to acquaint you with some techniques for implementing client-server applications that use the *User Datagram Protocol*, or UDP.[1] Your textbook does not deal with UDP, though it is in many ways simpler to use than stream sockets, which the book covers. You will also make use of some of the bit-diddling skills you acquired in Project 0, and your knowledge of data structure layout in memory from Chapter 3 in the text.

You will be given a basic code framework for UDP-based client and server programs. Your job will be to add code to implement both sides (client and server) of the simple protocol described in this document. There will be a server, running on the VM `calvert.cs.uky.edu` (IP address 172.31.145.229, port 31416), against which you can test your protocol during development, and another server on the same host that will automatically grade your project when you are ready to submit.

In the protocol, the client sends a single datagram to the server containing your user ID and a request identifier (an arbitrary number that should be different for each request). The server then replies with some different information, derived (in an unguessable way) from the information in the client's message. The goal of the protocol is simply for the client to get this additional information, called a "cookie".

> **Note:** In system programming and networking, a *cookie* is a piece of data (a number or string) given to a client by a service, so that the client can later prove something to the service. In this case, the purpose is just to prove that the transaction was completed successfully (like getting a receipt for a purchase).

The protocol is called "UDP Cookie Protocol".

Your job will be to write code (you may use C or C++) to do the following:

- correctly format the protocol messages as described later in this document;
- send protocol message(s) and re-transmit them if they get lost on the way and no response is received within a certain time;
- receive message(s) and interpret them according to the protocol; and
- print out (or write to a file) the information received, including the information in both the request and response messages.

The project has two parts. First, you will implement the cookie protocol client, and use it to get a cookie from the *test server*, which is waiting at IP address 172.31.146.118, port 31416. Then you will implement a server, and have it interact with the test server, which will then turn around and act like a client to test *your* server. The grading will be automatic, except for a small part based on code quality. A grading rubric will be provided on the Canvas page for this course.

[1] *Not* "Unreliable Datagram Protocol", as your textbook claims—though UDP is indeed not reliable like TCP.

2 Turnin

You will turn in a tar file containing *only* the following (i.e., there should not be any subdirectory).

- The source files for your code.
- A Makefile that compiles your code into client and server executable files, called `UDPclient` and `UDPserver`, respectively.
- Text files containing the output from your client and server successfully interacting with each other. (Example files will be available on the Canvas page for this assignment.)
- The output from your client's successful Type 0 interaction with the test server, as recorded using `script`.
- The output from your client's successful Type 1 interaction with the test server, as recorded using `script`.

3 UDP Cookie Protocol Specification

As with any client-server protocol, the server waits to receive messages on a particular port. (For this protocol the server is on port **31416**.) The client sends a *request message* to the server and waits for a response. Upon receiving a *request message* from a client, the server processes the request and sends back a *response message* to the address and port from which the request message were received. Each client-server interaction consists of one packet in each direction. Both request and response messages are short enough to fit in a single IP packet—the *maximum* length is 1200 bytes, and packets are usually much shorter.

The client is responsible for recovering from messages that are lost or duplicated. It does this by setting a timeout (using the `alarm` system call), and retransmitting its request message if the alarm goes off before it receives a response.

Note that either the request message or the response message may be lost, but the client is responsible for retransmitting in any case. Therefore the client-server exchanges should *idempotent*—the server should always return the same cookie for the same request information.

3.1 Message Formats

The messages exchanged by client and server all have the same format. Each message begins with a 12-byte *header* containing binary information, followed by a variable-length field containing text information. Request messages (sent by the client) are distinguished from Response messages (sent by the server) by one bit in the message, which is set to 1 in a Request and to 0 in a Response. The layout of the header is shown in Figure 1, where each row in the figure represents four bytes (32 bits) of the message. All multi-byte fields are in big-endian (network) byte order. (You can use the

Magic number (270)		message length
transaction identifier		
version/flags	result	port (type 1 only)
user ID/cookie/error message (variable length)		

Figure 1: Message Format

`htons`, `ntohs` functions to convert byte order.) The fields have the following meanings:

Magic Number (2 bytes) This field always contains the value (decimal) 270, or hexadecimal `0x10e`, in **network (big-endian) byte order**. A Cookie Protocol client or server may safely ignore any message it receives that does *not* contain the correct value in this field. (However, the testing server will return an error message in response to a malformed message.)

message length (2 bytes) This field contains the **total** number of bytes in the message, in network (big-endian) byte order. The value in this field should be at least (decimal) 16 and at most 512.

transaction identifier (4 bytes) This is a value chosen by the client to identify the transaction. The server returns this value unmodified. This field *should* contain a different value for each new transaction Request. (Note: this field is constant and not interpreted, so byte order does not matter.)

version/flags (1 byte) The bits of this field contain the following information (high-order bit is on the left):

```
bit 7 6 5 4 3 2 1 0
    V V V V T Y R E
```

The four high-order bits (corresponding to the first digit if the field is printed in hexadecimal) contain the **version number** of the protocol. This document describes version 2, so these bits should have the values 0010. The low-order four bits have the following meanings:

- Bit 3 (`T` above, for **testing**) is set to 1 by the client to indicate that it does not wish the server to log its score. If this bit is 0, the server will log the result and return a score in the result field. The server does not modify this field.
- Bit 2 (`Y`) indicates the **type** of the request. In a Type 0 request (Part 1 of this assignment), the client is just asking for a cookie. A Type 1 Request (Part 2) asks the server to test the that is running on the same host from which the Request originated, on the port indicated in the **Port** field of this message.
- Bit 1 `R` is the request/response flag, which is always set to 1 by the client, and to 0 by the server.
- Bit 0 (`E`) is always set to 0 in a Request. If it is set to 1 in a Response, it means the Request was malformed or erroneous in some way (e.g., the userid in the Request did not match the name of the host from which it was received). In this case, the variable-length field contains a textual error message rather than a cookie.

Result (1 byte) This field is set to 0 in a Request. In the Response to a well-formed non-test message (`T` flag is 0), this field contains the results of the server's test of the client, which is a numerical score between 0 and 100. (While this number is *not* directly mappable to your score on that part, because part of your marks depend on your code quality, 0 and 100 do represent the ends of the scale with respect to the auto-gradable part.)

Port (2 bytes) This field is always set to 0 in a Type 0 message. In a Type 1 message, it contains the **port number** on which the server to be tested is (already!) waiting to receive messages. **Note well: the server MUST have a datagram socket bound to the indicated port when the Type 1 Request is sent.** Also, this port **MUST be different** from the test server's port 31416.

user ID/cookie/errormessage This is a variable-length field; its actual length can be calculated by subtracting the length of the fixed header (12) from the total message length.

In a Request, this field contains the requesting user's LinkBlue ID. This ID **MUST** match the hostname from which the request originates, or the server will not give a cookie.

In a Response, if there is no error in the request, (`E` bit in the flags field is zero), this field contains the cookie, which is a string of ASCII characters.

Note well: the strings in the variable-length part of the message are *not* null-terminated; you must determine their length from the overall length of the message, and append a null (0) byte if you want to print them using standard I/O functions (printf, etc).

If the `E` bit is set in the Response, this field contains a human-readable error message that describes the error the server encountered, something like:

> "Incorrect magic number: ⟨value⟩"
> "Unknown protocol version: ⟨value⟩"
> "Invalid user ID: ⟨value⟩"
> etc.

3.2 Example Message

As an example, a Request for a Type 0 transaction from a user with ID `abc123` could contain the following (decimal) values in the fields (the Request ID could be anything):

0x10E		0x0012	
0xdeadbeef			
0x21	0x00	0x0000	
0x61 ('a')	0x62 ('b')	0x63 ('c')	0x31 ('1')
0x32 ('2')	0x33 ('3')		

3.3 Type 0 Transactions

A Type 0 transaction is very simple, as shown in Figure 2: the client sends a Request to the server, and the server sends back a Response message. If the client message is well-formed, the `E` bit in the Response is not set, and the Response includes a cookie. If the `T` flag was not set in the Request, the Response also includes a score indicating how well the client did.

If the client does not receive a Response within a reasonable time (say, 3 seconds) it should retransmit the Request. It keeps retransmitting (the same message) until it receives a Response, or until a limit on retransmissions (say, 5) is reached. (Retransmitting does not cause problems, because identical Requests elicit identical Responses.)

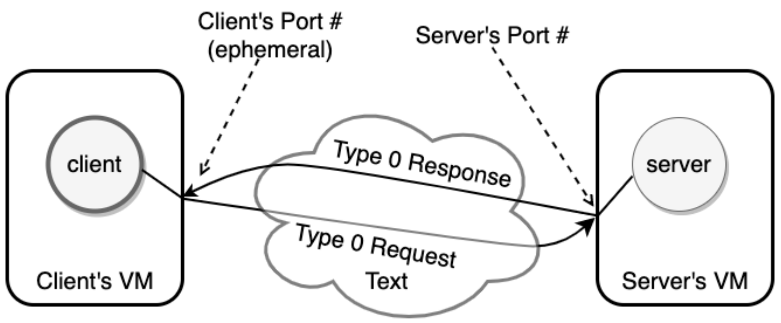

Figure 2: Message exchange in a Type 0 Transaction

3.4 Type 1 Transactions

Type 1 transactions are more complicated, as shown in Figure 3. They involve three programs: the

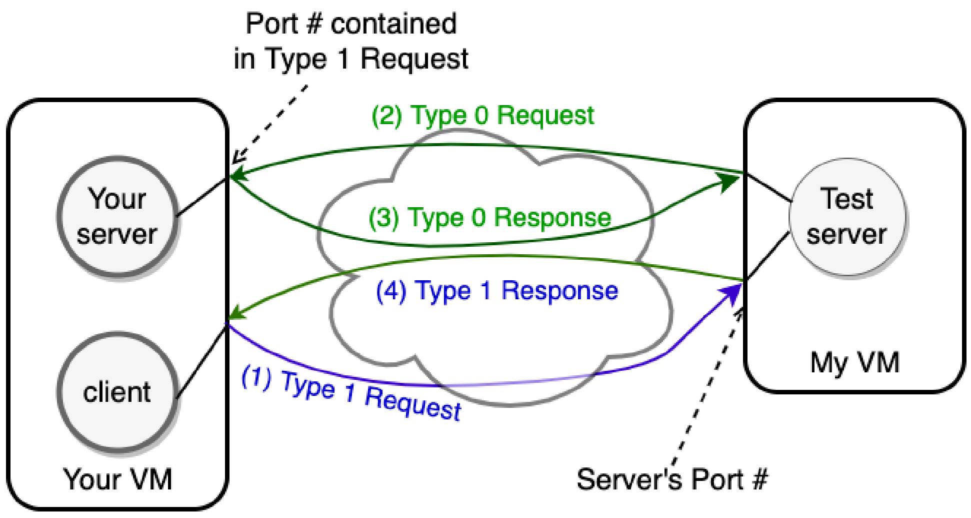

Figure 3: Message exchange in a Type 0 Transaction

student's client, which sends the original Request; the test server (listening at the same address and port as before), and the student server. The client sends a Type 1 Request to the test server, containing the port number on which the student server is ready to receive messages. The test server then sends a Type 0 Request to the student server at the given port on the same IP address from which it received the Type 1 Request. After receiving a Response from the student server, the test server may send additional Type 0 Requests to the student server. Finally, the test server returns a Response to the client, indicating the outcome of its interaction with the student server. If the server handled all the test server's Requests without error (and the Testing (`T`) flag was not set, the Response to the original message will include a cookie after the fixed header. If the student server is deficient in one or more tests, the Response will include an error message listing the problems detected by the instructor's server.

Among the things the test server looks for when interacting with the student server are:

- Correctly-formed response messages. For example, the message length field accurately reflects the contents of the message—that is, its value is 12 plus the length of the cookie string.
- Ability to handle malformed messages. The student server SHOULD indicate an error in the response to a malformed Request, but it MUST NOT crash.
- Always returning the same Response to the same Request from the same client address and port.
- Returning responses in a timely way, say, within three seconds.

Your student server may construct the cookies it returns in Response messages in any way you like, but they must be tasteful human-readable ASCII strings.

4 Your Assignment

To get started, download the **tarball** from the Canvas page for this assignment onto your VM, and unpack it into a directory where you will work on this project. The tar file includes:

`UDPClient.c` A skeleton file for the client. You will add code to construct the Request message, parse the Response message, and handle retransmission as necessary.

`UDPServer.c` A skeleton file for the server. You will add code to parse incoming Request messages, construct a response message (including a cookie) and send it back to the client.

`die.c` A couple of procedures that print error messages and call `exit`.

`UDPCookie.h` A header file that declares some useful constants and types.

You can compile your client with:

```
unix> gcc -Wall -o client UDPClient.c die.c
```

and similarly for the server:

```
unix> gcc -Wall -o server UDPServer.c die.c
```

4.1 Part 1: Your Client Works with Test Server

For this part, which is worth 20 points, you will add the necessary code to `UDPClient.c` to construct and send a Cookie protocol **Type 0 Request** message containing your UserID. You will also write code to set up timeouts, and to handle retransmission after a timeout.

While you are debugging your client, be sure to set the Test flag in the Request message. When you believe your client is working correctly, send a Request with the Test flag clear. If the Response message indicates success (contains a cookie), be sure to save it and turn it in with your code. The server will log your Request and the corresponding Response, and they will be used in grading.

4.2 Part 2: Your Server Works with Your Client

For this part your program does not have to interact with the test server. You will develop your server by adding code to `UDPServer.c` to parse and sanity-check incoming Type 0 Request messages, and to prepare and send appropriate **Type 0 Response** messages. Your server can create cookies however you want, as long as:

- different (valid) Requests elicit different cookies; and
- identical (valid) Requests elicit identical Responses (including cookies).

The server should display (or write to a file) information about each Request it processes and the corresponding Response.

This part is worth 10 points. You will turn the client's and server's output from a successful transaction along with your code.

4.3 Part 3: Your Client and Server Work with Test Server

For this part, which is worth 30 points, you will modify your client to send Cookie protocol **Type 1 Requests**. (An easy way to do this is to accept an optional third command-line argument that specifies the port number to put in the Request. If the third argument is present, a Type 1 transaction is always performed.)

As described in Section 3.4, upon receipt of a Type 1 Request, the test server immediately acts like a client and sends **Type 0 Requests** to your server at the port number specified in your Type 1 Request message (at the same IP address from which the Type 0 Request originated, i.e., that of your VM). The test server will eventually return a **Type 1 Response** indicating the result; if successful, the message will include a cookie.

As in Part 1, your client should set the `T` flag in your Requests until you believe you are finished debugging your server. The test server will not send cookies in response to Requests with the `T` flag set, but will set the `E` flag if errors are detected, and include text error descriptions after the fixed header of the message. As in Part 1, your client should display the Request and Response messages, so you can record them to turn in.