CS 356 **S. S. Lam**

Lab Assignment 1

Overview

This Lab assignment is similar to the previous one, in that you will use Python to implement a simple client-server protocol. There are several differences, however. This time you will use the `SOCK_DGRAM` service instead of `SOCK_STREAM`; the `SOCK_DGRAM` service is implemented using UDP, the User Datagram Protocol, instead of TCP. In addition, the client and server in this assignment communicate by exchanging structured binary information, instead of lines of ASCII text.

You will be constructing a server as well as a client. The server in this assignment is a "database" server, which maps (fictitious) social security numbers to (fictitious) P.O. box numbers. The client sends a request containing a single social security number; the server returns a response containing that social security number and the Post Office box number of the student to whom it belongs. The "database" of SSN–P.O. box number pairs will be provided in a file that you can use to check responses and to initialize your own server.

The objectives of this exercise are:

- To acquaint you with some of the implementation techniques for protocols that use structured messages and attach a header to user data, such as IP, TCP, and UDP.
- To help you understand the use of datagram sockets.
- To help you understand the basic ideas of *presentation encoding*, including network vs. host byte ordering.
- To acquaint you with the implementation of *timeouts* for recovery from message losses.

You should use a Linux machine that is connected to the department's network (more specific instructions on TA web page). More information for this assignment can be found on the TA web page.

Exercise 0: Implementing a Client

The client and server in this exercise communicate by sending *request* and *response* messages over an *unreliable datagram* service. Because the service is best-effort (there is no guarantee that a message sent will arrive at all), implementing reliability is up to the communicating parties. For Lab 1, reliability is entirely up to the client, which uses *timeouts* to detect losses.

The client loss-detection mechanism works like this: the client sends a request to the server, then waits to receive the response (each request and each response is a single message). If a response is not received within, say, 5 seconds, the client retransmits its request to the server. It keeps trying until it either receives a response or reaches a predefined maximum number of tries, after which the client gives up and reports failure to its user.

In addition to detecting losses, the client and server both add a *checksum* to each message, to detect corruption of messages in transit. Each computes the checksum value and places it in the message before transmission; each performs a checksum calculation on a received message to verify that it is the same as the message sent.

For this lab assignment, the CS356 server runs on three cores of `paris.cs.utexas.edu` (IP address `128.83.144.56`) waiting to receive messages from ports 35604, 35605, and 35606. Your program can contact the server at any one of these three ports.

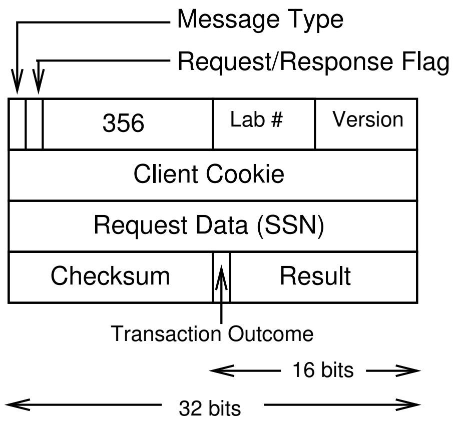
When a server receives a *request datagram* containing a 32-bit integer representing a social security number, it looks up the number in its database to find the corresponding P.O. box number, if any; places the result in a *response datagram*; and sends the response back to the originating host and port. The server always gives the same response for the same request. A list of 50 SSNs and P.O. box numbers that the server knows is contained in a file available from the TA web page. The file contains a sequence of lines, each containing two tokens separated by whitespace:

⟨SSN⟩ ⟨P.O. Box⟩

An "SSN" is a sequence of nine digits, and a "P.O. Box" is a sequence of five digits. Each line is terminated by a single newline character '`\n`'.

The Protocol

The request and response datagrams both have the same format, which is shown below.



Each message consists of exactly 16 bytes, organized as a series of one, two, and four-byte *fields*.

> **Note:** All multi-byte integer-valued fields are transmitted in BIG-ENDIAN order, i.e. most significant byte first. This is standard "network byte order."

The first field of each message is 16 bits long, and contains the following information:

- The low-order 14 bits contain the value 356 in binary, i.e. 00000101100100.
- The most significant bit of the first 16-bit field is the **Message Type** bit, which for this exercise is 0. (The message format for a Type 1 message is described in the next exercise.)
- The next-most significant bit of the first 16-bit field is the **Request/Response** bit; it is set to 0 in a request, and 1 in a response. Every message the client sends has this bit set to 0.

The next field is eight bits long, and contains the **lab number** of this exercise, i.e. the value one, or 0x01 in hex.

The next field is also one byte long, and contains the **version number** of the lab, which is seven, or 0x07 hex. Thus, the first 32 bits of a request in this exercise will contain the value:

```
0000 0001 0110 0100 0000 0001 0000 0111
```

The next field of the message is 32 bits in length and contains the **Client Cookie**. This is an arbitrary value chosen by the client and placed in the request; it is always included by the server in its response to a client request, and may be used by the client to associate a received response with the corresponding request (e.g., in case the client has more than one request outstanding at any time).

The next field is also 32 bits long, and contains the **Request Data**. In a Type 0 message, this is a social security number, represented as a (big endian) 32-bit integer.

The next field of the message is 16 bits in length and contains the **Checksum**, which is used to detect transmission errors and (more likely) improperly-formed messages. For this assignment, we use a simple version of the Internet checksum. Specifically, before sending a message (either a request or a response), the sender constructs the value to be placed in the checksum field as follows:

1. Initialize the **Checksum** field to zero.
2. Viewing the message as a sequence of eight 16-bit integers, add them using ones complement arithmetic and take ones complement of the sum.
3. Write the result in **Checksum** field of the message, i.e., overwriting the initial zero value.

Upon receiving a message, the Receiver computes the ones complement sum of the message viewed as a sequence of eight 16-bit integers; if no error detected, the result is a bit string of 16 ones.

The last field of a Type 0 message is also 16 bits. It is unused in a Request (and its value is ignored by the server). In the Type 0 Response message, it is the **Result** field. The most significant bit of this field is the **Transaction Outcome** bit, which determines the meaning of the low-order 15 bits. A Transaction Outcome of 0 signifies success; in this case the low-order 15 bits contain the Post Office Box number, encoded as an integer. A Transaction Outcome of 1 indicates that an error occurred in processing somewhere; in this case the following four integer values of the low-order 15 bits encode four types of errors:

1: Checksum Error. The server's checksum computation on the received request yielded a result other than a bit string of 16 ones. (This almost always means that the client has computed the checksum incorrectly, since actual corruption of data is very rare in this environment.)

2: Syntax Error. The checksum on the request verified correctly, but some aspect of the message format is incorrect (e.g. the value of the low-order 14 bits of the first two bytes was not 356).

4: Unknown SSN. The request data supplied in the message does not correspond to a Social Security Number contained in the database.

5: Server Error. The server aborted processing of this message after detecting an internal error.

The entire transaction consists of a single message in each direction: the client sends a request, and the server sends its response. In case of loss, retransmission is the client's responsibility.

Client Operation

To implement the above protocol, the client does the following:

(i) Create a socket of type `SOCK_DGRAM` and address family `AF_INET`.

(ii) Fill in a message structure with the required message type and version information, and a client "cookie" value (and remember it for later use).

(iii) Prompt the user for a Social Security Number from the terminal, and put it in the appropriate field.

(iv) Compute the checksum and fill in the checksum field.

(v) Start a timer (to detect lost messages) using `settimeout()`.

(vi) Send the request message to server (identified by its IP address and port) using `sendto()`.

Do not forget to convert each multi-byte field in the message to network byte order.

(vii) Receive the response message using `recvfrom()`.

(viii) If the timer expires, send the request message again, up to some maximum number of retransmissions (five should be plenty).

(ix) When the response message is received, check it for the proper version, cookie, etc., and recompute the checksum to ensure that no error has occurred in transit. If it is okay, output the P.O. Box number in a response to the user.

Writeup

Turn in your well-commented code along with a record of the information your client received from the CS356 server. (See TA web page for detailed submission instructions.)

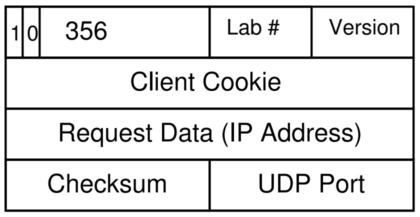
Exercise 1: Writing and Testing a Server

In this exercise you will write a *server* that implements the protocol above. It is suggested that you first test your server with the client you wrote in Exercise 0 by running them in different UTCS Linux machines. For your own tests, the server address in your client program will need to be changed to that of your own server.

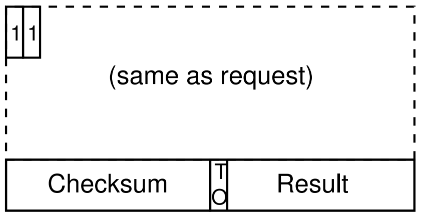
Subsequently, when your client and server interact with the CS356 server, your client and server will run in the same UTCS Linux machine (they will have the same IP address.)

Then you will *modify your client* to send a Type 1 Request to the CS356 server. A Type 1 Request message asks the CS356 server to act as a client and "probe" another server. The Type 1 Request contains the IP address (a 32-bit integer) and port number (a 16-bit integer) of your server to be probed. These integers **must** be represented in network byte order, i.e. Most Significant Byte is transmitted first.

The Response to a Type 1 Request contains information indicating whether the test was successful, i.e. whether your server-under-test (SUT) successfully handled the CS356 server's Type 0 requests.



Request Format



Response Format

The Protocol (Type 1 messages)

The format of a Type 1 message is shown above. It is identical to the Type 0 message except for the **Message Type** field (which is 1) and the interpretation of bytes 8–15.

The fifth field of the message (i.e. bytes 8–11) contains, instead of an SSN, the IP address of your server to be tested, in *network byte order*, i.e. most significant byte first. The next field is the **Checksum** as before. The final 16-bit field in the Type 1 Request contains the (big-endian) **UDP Port Number** of your server to be tested. In the Response from the CS356 server, the most significant bit of the final field is the **Transaction Outcome** (TO) bit, and the low-order 15 bits contain an integer **Result** code. A zero Transaction Outcome bit indicates success; in this case the Result field contains **zero**. A Transaction Outcome of one indicates failure, with the Result field containing error codes.

All of the low-order 15 bits are zero if no error detected. If errors were detected by the CS 356 server, the lowest six bits are used to denote six different error conditions.

bit 0 is one: No response from SUT at all

bit 1 is one: No response from SUT to some requests

bit 2 is one: Bad message from SUT (syntax error)

bit 3 is one: Probable bad message from SUT (bad checksum)

bit 4 is one: Wrong result (P.O. Box Number) returned by SUT for a SSN

bit 5 is one: Incorrect error handling by SUT (in response to a bad message sent intentionally by the CS 356 server)

The protocol for a Type 1 transaction is a three-party protocol: the client, the CS356 server, and the server-under-test (SUT). It proceeds as follows:

1. Client sends a Type 1 Request to the CS356 server, containing the IP address and port number of the SUT.
2. The CS356 server, having received the Request, formats and sends a Type 0 Request with a SSN to the SUT at the specified address.
3. The SUT receives the Type 0 Request, looks up the P.O. Box number for the SSN and sends a Type 0 Response to the CS356 server.
4. The CS356 server receives the Response and checks it for well-formedness. It will send several different Type 0 Requests to the SUT, and check every response from the SUT.
5. When it has finished interacting with the SUT, the CS356 server sends back a Type 1 Response containing an indication of whether the SUT passed the test.

Server Operation

Note: the server you write *only* has to implement the server side of the protocol described in Exercise 0, i.e. Type 0 transactions.

The steps followed by the server are the following:

1. Load the database information from the file. (Alternatively, it's okay if you just "wire in" the data in the server code.)

2. Create a socket of type `SOCK_DGRAM` and family `AF_INET`.

3. Bind the socket to some port using `bind()`.

4. Loop forever, doing the following:

   (a) Receive a datagram with `recvfrom()`. Check it for correct version, format and checksum, etc. If any of these is incorrect, return the appropriate result error code specified in Exercise 0.

   (b) Otherwise, look up in the database the P.O. Box number for the SSN supplied in the request; if the SSN is present, fill in the **Result** field of the response, re-compute the checksum, and send the response datagram back whence it came using `sendto()`. Note that you can use the address returned in the `recvfrom()` call as the argument to `sendto()`; you don't even need to examine the address itself.

You would want to have your server record each transaction in a log file, or output a summary as it executes.

Writeup

Turn in your well-commented code and a record of your client's interactions with the CS356 server. (See TA web page for detailed submission instructions.)

Notes

Modifications to the client in Exercise 0 to make it send a Type 1 request are straightforward.

You can run both the client and your server (SUT) of Exercise 1 in different Linux machines. However, we ask you to run them both in the same UTCS Linux machine to facilitate the work of the TA.

Lastly, the interaction between the CS356 server and your server may involve several transactions, and may take a long while, particularly if any messages are lost in transit. Therefore the client should use a much longer retransmission timeout interval for Type 1 Requests than for Type 0.