Overview

This Lab assignment is similar to the previous one, in that you will be implementing 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. (This document is written as if you are going to write your programs in C, even though you are allowed to write your programs in C++ or Java or Python.)

You will be constructing a simple 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. Setup and Preparation.

Setup and Preparation

You should use a UNIX system that is connected to the department's network. The skeleton code and other information for this assignment can be found on the class web page.

Among the library routines and system calls you will need to understand for this assignment are: `sendto()`, `recvfrom()`.

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.

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.

The CSE3300 server for this exercise waits to receive messages on UDP port 3300 of tao.ite.uconn.edu (IP address 137.99.11.9).[1] It iteratively receives a request datagram containing a 32-bit integer representing a social security number; 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

[1] To increase availability and spread the load, an identical server runs on 3301. Your client may access any one of the servers

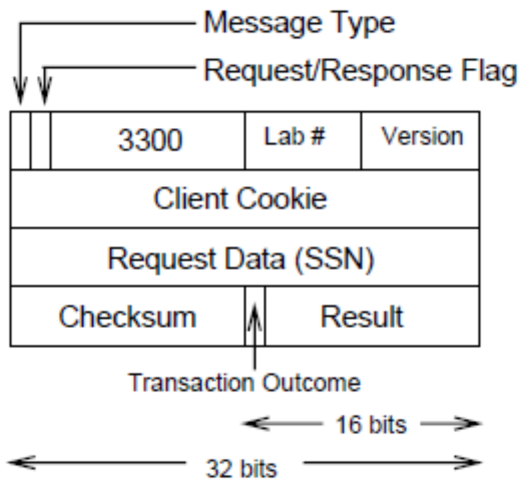
originating host and port. The server always gives the same response for the same request. A list of 52 SSNs and P.O. box numbers that the server knows is contained in a file available from web page. The file contains a sequence of lines, each containing two tokens separated by whitespace:

```
<SSN> <P.O. Box>
```

An “SSN” is a sequence of nine digits, and a “P.O. Box” is a sequence of four 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”, so you can use the routines `htonl()`, `htons()`, etc. to convert to it.

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

- The low-order 14 bits contain the value 3300 in binary, i.e. 00110011100100.
- The most significant bit of the first 16-bit field is the **Message Type** flag, 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 flag**; 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 four, or 0x04 in hex.

The next field is also one byte long, and contains the **version number** of the lab, which is eight, or 0x08 hex. Thus, the first 32 bits of a request in this exercise will contain the value:

```
0000 1100 1110 0100 0000 0100 0000 1000
```

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 3300).
3. Unknown SSN. The request data supplied in the message does not correspond to a Social Security Number contained in the database.
4. 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.

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.

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`) using `socket()`.

(ii) Fill in a message structure with the required version information, message type 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.

Do not forget to convert this and other multi-byte fields to network byte order.

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

(v) Fill in a `sockaddr_in` structure with the IP address and port number of the server.

(vi) Call `sendto()`, giving it a pointer to the filled-in message structure, and a pointer to the destination address structure.

(vii) Call `recvfrom()`, giving it a pointer to an un-filled-in message structure and a pointer to the same address structure. (Note: When the call returns, the system will have filled in the address of the sender of the received datagram. It is extremely unlikely that it would be from anywhere other than the server, but it *is* possible.)

(viii) When the response 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 the response to the user.

The C skeleton code for the client program is available from the class web page. Your job is to fill in the missing portions of the code, so that the client follows the steps given above to implement the protocol. Also, the file lab4.h contains some useful definitions, including a declaration of the message structure.

Extra credit:

The packet can be lost due to the unreliable transmission. Thus a client loss-detection mechanism can be implemented 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.

Writeup

Turn in your well-commented code along with a record of the information your client received from the CSE3300 server.

Exercise 1: Writing and Testing a Server

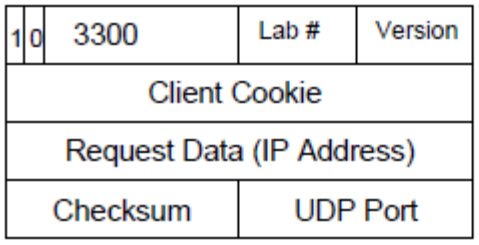
In this exercise you will write a Server that implements the protocol above. It is suggested that you first test your server with the client you wrote in Exercise 0. Then you will modify your client to send a Type 1 Request to the CSE3300 server. A Type 1 Request message asks the CSE3300 server to act 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 CSE3300 server’s Type 0 requests.

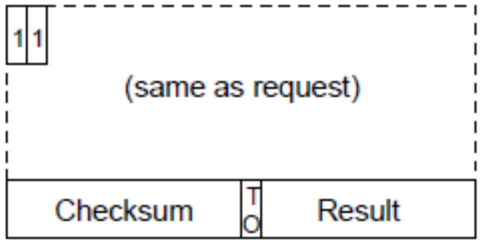
The Protocol (Type 1 messages)

The format of a Type 1 message is shown below. It is identical to the Type 0 message except for the

Message Type field (which is 1) and the interpretation of bytes 8–15.



Request Format



Response Format

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, 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 CSE3300 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 3300 server)

The protocol for a Type 1 transaction is a three-party protocol: the client, the CSE3300 server, and the server-under-test (SUT). It proceeds as follows:

1. Client sends a Type 1 Request to the CSE3300 server, containing the IP address and port number of the SUT.
2. The CSE3300 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 CSE3300 server.
4. The CSE3300 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 is finished interacting with the SUT, the CSE3300 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` using `socket()`.
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. 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 CSE3300 server.