

Homework 8 (UDP Client Programming)

Assigned: November 12

Due: Midnight November 22

Overview

In this assignment, you will again write a client that contacts our server. This client and server communicate using the `SOCK_DGRAM` service rather than the `SOCK_STREAM` service. The `SOCK_DGRAM` service is implemented using UDP, the User Datagram Protocol, rather than TCP. UDP is an unreliable datagram service; your client will be responsible for some reliability. The client and server communicate by exchanging structured binary information, instead of lines of ASCII text.

The server in this assignment is a database server, that maps fictitious social security numbers to fictitious Georgia Tech PO Box numbers. The client sends a request containing a single social security number; the server returns a response containing the social security number and the corresponding PO Box number. The database of SSNs and PO Box number pairs will be provided in a file so that you can generate valid requests and check responses.

The objectives of this assignment are:

- To acquaint you with implementation techniques for protocols that use structured messages and protocols that attach a header to user data.
- To help you understand the use of datagram sockets.
- To help you understand the basic ideas of *presentation encoding*, including network versus host byte ordering.

Setup and Preparation

The skeleton code for this assignment can be found in the CoC file system at:

```
~ewz/pub/cs3251/fa99/
```

The skeleton (and other files described below) will also be published on the course web page. Your job is to fill in the skeleton to implement the protocol described in detail below. You will need to be familiar with the routines `sendto()` and `recvfrom()`.

In addition to skeleton client code, that directory also contains a header file `hw8.h` that contains the definition of the message structure and other definitions useful to the client and server. The directory also contains the file `Data`, which contains a list of 50 SSNs and PO Box numbers that constitutes the server's database. The list is a sequence of lines, each of the form:

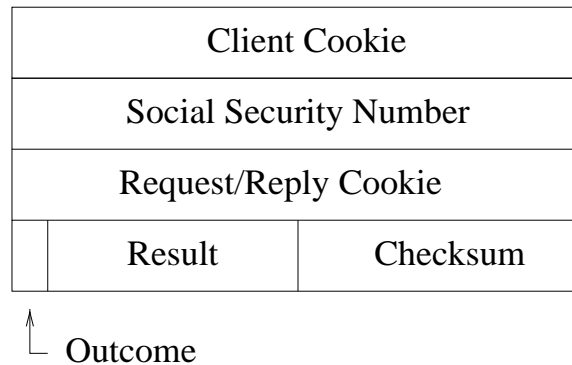
```
⟨SSN⟩ ⟨WS⟩ ⟨P.O. Box⟩
```

where `⟨SSN⟩` is a sequence of nine digits, `⟨WS⟩` is white space and `⟨P.O. Box⟩` is a sequence of four digits. Each line is terminated by a newline character.

The server will run on `flora.cc.gatech.edu` on port 23251.

The Protocol

The client and server communicate by sending *request* and *response* messages. The request and response datagrams have the same format, shown below.



The message fields have the following size and semantics:

- **Client Cookie.** This field is 32 bits in length and contains an arbitrary value chosen by the client and placed in the request message. The server copies it into the corresponding response message, thus it can be used by the client to associate a response with a particular request (e.g., in case more than one request is outstanding at a time).
- **Social Security Number.** This field is 32 bits in length. On a request, it contains the SSN for the request. The server copies the value into the response message.
- **Request/Reply Cookie.** This field is 32 bits in length. It has a value of 0 in a request message. In a reply message, it has a non-zero cookie value selected by the server. This allows us to match up the replies you receive with what the server logs as its activity.
- **Outcome-Result.** This field is 16 bits in length. It is unused in a request, and thus may have any value. In a response, the most significant bit is the **Outcome** flag. If the outcome is 0 then the other 15 bits contain the PO Box for the request. If the outcome is 1 then the other 15 bits indicate the type of error. The type of error is encoded as follows:
 - 1: Checksum error. The checksum calculation for the request packet detected an error. The most likely cause of this error is an incorrect computation of the checksum at the client.
 - 2: Syntax error. The value in the **Request/Reply Cookie** field is non-zero in the request message.
 - 3: Unknown social security number. The value in the **Social Security Number** field does not match any of the social security numbers in the database.
 - 5: Other. This is a catch-all category for any other problem at the server.
- **Checksum.** The last field is 16 bits in length and contains the Checksum, used to detect transmission errors and (more likely) improperly formed messages. For this assignment, we use a simple checksum which is the bitwise exclusive-or over all the 16-bit words of the message. Before sending a message, the sender does the following:
 1. Sets the value of the **Checksum** field to 0.

2. Viewing the message as a sequence of eight 16-bit words, computes the bitwise exclusive-or of these eight words.
3. Writes the result in the **Checksum** field of the message.

Upon receiving a message, the receiver computes the bitwise exclusive-or of the eight words of the message. If no errors have occurred, the result will be 0.

Reliability

Because this service uses UDP, it is best-effort. That is, there is no guarantee that any message sent by the client or server will be received correctly (or at all). The protocol as describes has some reliability, in the form of the protection of the checksum. However, the protocol does not protect itself against messages lost in transit. Lost messages are relatively infrequent within a campus network, however you should realize that losses may occur. (If a loss does occur, you will need to recover from it at the “layer” above this protocol. For example, by using ctrl-C to halt your program, and then re-trying.)

Client Operation

The steps followed by the client are:

1. Create a socket of type `SOCK_DGRAM` and family `AF_INET` using `socket()`.
2. Fill in a `sockaddr_in` structure with the IP address and port number of the server.
3. Generate a random value to use as a cookie. Fill in the cookie field of the message structure and save the value for comparison upon receiving a response.
4. Prompt the user for a Social Security Number from the terminal, and put it in the message structure.
5. Fill in the remaining parts of the message, as required.
6. **Convert all multi-byte fields to network byte order using `htons()` and `htonl()`.**
7. Compute the checksum and fill in the checksum field (without converting the order).
8. Call `sendto()` to send the message. Among other parameters, you will need to provide a pointer to the filled-in message structure and a pointer to the server address structure.
9. Call `recvfrom()`, giving it a pointer to an empty message structure and a pointer to the same address structure.
10. When the response is received, recompute the checksum to be sure that no errors occurred in transit. If there are no errors, then the checksum value will be 0. Then, **convert all multi-byte fields to host byte order using `ntohs()` and `ntohl()`**, and check the message for the proper cookie and SSN. If the message checks out, output the response PO Box number and Reply Cookie to the user. Otherwise, report the error contained in the message.

What to Turn In

The deadline for contacting the server is midnight, Monday November 22. You will turn in your assignment in two ways. (1) By midnight, send email to `cs3251@cc` with a publicly accessible location where your code can be found. (2) In class on Tuesday, November 23, turn in a printout of your well-documented code and its output (i.e., all lines returned by the server).

You will be graded on the correctness of the code and its readability and structure. The server keeps a log of all connections, which will be used to verify contact with your client. Include in your writeup a publicly accessible location where the TA can get an on-line copy of your code. November 23.