CS 645 Computer Networks **Fall 1999**

Assignment 1

Assigned: 21 Sept. 1999 *Due: 30 Sept. 1999*

Overview

This 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.

You will again be constructing a simple server as well as a client. The server in this assignment is a "telephone number" server, which maps (fictitious) identification numbers to (fictitious) 5-digit telephone numbers. The client sends a request containing a single social security number; the server returns a response containing that social security number and the telephone number of the person to whom it belongs. The "database" of ID-telephone 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 protocols that 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

You may use any UNIX system that is connected to the campus network.[1] The information required for this project is available via the course web page. Among the library routines and system calls you will need to understand for this assignment are: `sendto()`, `recvfrom()`, `signal()` (or `sigaction()`) and `alarm()`.

Exercise 0: Transaction-oriented Client

The client and server in this exercise communicate by exchanging *request* and *response* messages via UDP's *best-effort 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 loss-detection mechanism works like this: the client sends the request to the server, then

[1] See the special note in Exercise 1 if you are using the ResNet/@home cable network.

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 the user. Note that the client will also retransmit a request if the server's response gets lost, so it is important in such protocols for client requests to be *idempotent*, i.e. the same request results in the same response, no matter how many times it is processed.

In addition to detecting losses, the client and server both add a *checksum* to the 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.

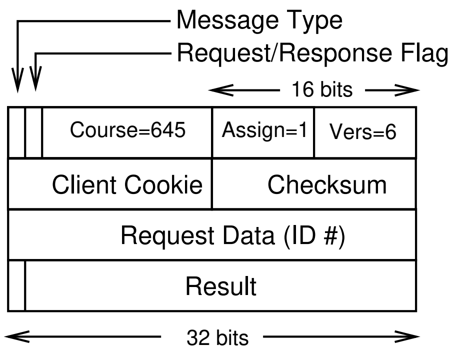
The server for this exercise waits to receive messages on UDP port 13579 on `gladstone` (IP address 204.198.76.142). It iteratively receives a *request datagram* containing a 32-bit integer representing an ID number; looks up the number in its database to find the corresponding telephone number, if any, and 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. The list of ID and telephone numbers that the server knows is available via the course web page. The file contains a sequence of lines, each containing two tokens separated by whitespace:

⟨ID Number⟩ ⟨Telephone Number⟩

In the file, an "ID Number" is a represented as a sequence of nine decimal digits, and a "Telephone Number" is a sequence of five decimal digits. Each line is terminated by a single newline character '`\n`'. (Note that in the request and response messages, SSNs and P.O. Box numbers are transmitted as *binary* numbers.)

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 Well:** 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 645 in binary, i.e. 00001010000101.
- The most significant bit of the first 16-bit field is the **Message Type** flag, which for this exercise is set to 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 **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 **number of this assignment**, 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 six, or `0x06`. Thus, the first 32 bits of a request in this exercise will contain the binary value:

```
0x02850106 = 0000 0010 1000 0101 0000 0001 0000 0110
```

The next field of the message is 16 bits in length and contains the **Client Cookie**. This is an opaque value[2] 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 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 checksum that computes the bitwise exclusive-or over all of the 16-bit words of the message. The process of computing the checksum is described below.

The next field is 32 bits long, and contains the **Request Data**. In a Type 0 message, this is an ID number, transmitted as a 32-bit integer in **network byte order**.

The last field of a Type 0 message is also 32 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, which tells whether the transaction was successful. The most significant (in network byte order) bit of the **Result** tells whether the transaction was successful, and determines the meaning of the other bits. A value of 0 in this bit signifies success; in this case the low-order 31 bits contain the 7-digit telephone number, encoded as an unsigned binary integer. If the most significant bit is 1, it indicates that an error occurred somewhere In this case the other 31 bits indicate the nature of the error. Errors that may be indicated (see `lab1.h`) include:

- Checksum Error. The sender's checksum computation on the received request yielded a result other than `CSUMRESULT`. (This almost always means you are computing the checksum incorrectly, since actual corruption of data is practically unheard-of in this environment.)
- 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 645), or the assignment or version numbers were not correct.
- Unknown ID Number. The request data supplied in the message does not correspond to one of the IDs in the database.

[2]"Opaque" means it is not interpreted by the receiver in any way, it is just a bit string. In particular, it is not treated as an integer and therefore is not byte-swapped at the receiver.

- 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 the 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 protocol family `PF_INET`) using `socket()`.

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

(iii) Prompt the user for an ID number from the terminal, and put it in the appropriate field.

(iv) Compute the checksum (see below).

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

(vi) Start a timer, to detect lost messages. (See "About Timeouts", below.)

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

(viii) Call `recvfrom()`, giving it a pointer to an un-filled-in message structure and a pointer to the same address structure. (Note: it would be OK here to give a pointer to an address structure containing `INADDR_ANY` and 0 for the port number, as long as you were sure nothing would show up from anywhere else other than the server. 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.)

(ix) If the timer expires, call `sendto()` to send the message again, up to some maximum number of retransmissions (five or six should be enough).

(x) When the response is received, validate the checksum to ensure that no errors occurred in transit. If OK, check the message for the proper version, cookie, etc. If everything checks out, output the response telephone number to the terminal in the format "DDD-DDDD"; otherwise, output information about what went wrong.

The file `lab1.h`, available via the course web page, contains some useful definitions, including a declaration of the message structure. Your job, as usual, is to fill in the missing portions of the code, so that the client follows the steps given above to implement the protocol.

Computing the Checksum

One slightly tricky aspect of sending binary-formatted data is computing the checksum. The problem is that the checksum value in general depends on the ordering of bytes, which may differ between sender and receiver. Therefore both ends must agree on (i) exactly what parts of the message are byteswapped, and (ii) when the checksum is computed relative to byteswapping.

The general principle is that the first thing the receiver does is compute the checksum on a received message; if this does not result in the correct value, the message is discarded. The result of computing the checksum must be an agreed-upon, recognizable value (in this case, the constant value CSUMRESULT, which is `#defined` in the file `lab1.h` to be `0xbabe`).

Because the receiver does the checksum computation *before* byteswapping, the sender must do it *after* byteswapping, i.e. with all fields of the message exactly as they will be transmitted. Just before sending a message (either a request or a response), the sender does the following:

1. Set the value of the **Checksum** field to `CSUMRESULT`.
2. Make sure all multi-byte integer fields (including **Checksum**) are in network byte order. (Note: this implies byteswapping the Checksum field *after* you have initialized it.)
3. Viewing the message as a sequence of eight 16-bit words, compute the bitwise-exclusive-or of these 8 words.
4. Write the result in the **Checksum** field of the message (i.e., *overwriting* the original value.

Upon receiving a message, the Receiver computes the bitwise exclusive-or over all the 16-bit words of the message (i.e. what the sender does in step 2 above); if no errors have occurred, the result will be the constant value `CSUMRESULT` (in network byte order).

Writeup

Turn in your well-commented code along with a record of the information your client received from the 645 server. Include a pointer to your client code, in case the TA wants to test it.

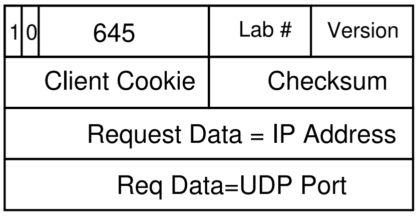
Exercise 1: Writing and Testing a Server

In this exercise you will write a *Server* that implements the protocol above. It is suggested that you first test your server with your own client, i.e. the one you wrote in Exercise 0. Then you will *modify your client* to send a Type 1 Request to the server on `gladstone` server. A Type 1 Request message asks the Server to "probe" another server, i.e. to act as a client (sending Type 0 messages only). The Type 1 Request contains the IP address (a 32-bit integer) and port number (a 16-bit integer) of the server to be probed. These integers **must** be represented in network byte order, i.e. Most Significant Byte is transmitted first.

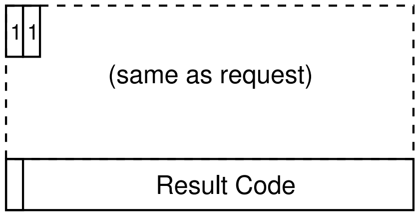
The Response to a Type 1 Request contains information indicating the results of the request, i.e. whether the server-under-test successfully handled the `gladstone` server's Type 0 request.

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 **Request Data** field of the message (i.e. bytes 8-11) contains, instead of an SSN, the IP address of the server to be tested, in *network byte order*, i.e. most significant byte first. The final field is again 32 bits in the Type 1 Request. It contains the UDP port number of your program, i.e. the server to be probed, as a 32 bit integer in network byte order. Note that the first two bytes of this number will always be zero.

The protocol assigns a special meaning to the value `INADDR_ANY` (i.e. the address 0.0.0.0) when it is sent in the Request Data field of a Type 1 Request message. In that case it means "use the IP address from which this request originated." So if the server receives a Type 1 Request from 128.163.17.21, and the value in the first part of the Request Data field is 0.0.0.0, and the UDP port number is 6000, the server will probe the remote server at address 128.163.17.21-6000. *This enables clients to run behind Network Address Translation devices, i.e. on the ResNet system.*

In the Response, the most significant bit of this field again indicates the overall outcome (0 for success, 1 for failure) and the low-order 31 bits encode any errors that were detected during the test. In addition to the errors defined for Type 0 requests, `lab1.h` defines the following additional error flags for Type 1:

- **Invalid.** Either the IP address or the port specified in the Type 1 request was an invalid value. Note that if either high-order byte of the "result" field is nonzero in the request you will get this error.

- **No Response.** No response was received from the specified server for one or more of the requests sent to it.

- **Checksum Error.** The server returned a message with an incorrect checksum. (Note: in theory this could be due to a corrupted message, but that is highly unlikely, especially if the error is repeatable.)

- **Size Error.** The server returned a message that was the wrong size.

- **Incorrect.** The value returned by the server under test was not consistent with the database, or the result flag was incorrect.

- **Cookie Problem.** The server returned a message containing an unknown client cookie value.

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

1. Client sends a Type 1 Request to the `gladstone` server, containing the IP address and port number of the Server-Under-Test (SUT).

2. The server on `gladstone`, upon receiving the Request, formats and sends one or more Type 0 Requests to the Server-Under-Test at the specified address.

3. The SUT receives each Type 0 Request, looks up the result, and sends a Type 0 Response to the `gladstone` server.

4. The `gladstone` server receives the Responses and checks each for correctness.

5. When it is finished interacting with the SUT, the `gladstone` server sends back a Type 1 Response containing the outcome of 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 to be followed by your server are really quite simple:

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

2. Create a socket of type `SOCK_DGRAM` 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.

   (b) Otherwise, look up the data supplied in the message in the database; if present, fill in the **Result** field of the message according to the result, re-compute the checksum, and send the 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 may want to have your Server record each transaction in a log file, or output a summary as it executes.

Writeup

Turn in your code and a record of your client's interactions with the `gladstone` server. Include a pointer to your client executable, in case the TA wants to test it.

Other Notes

The modifications to the client to have it send a Type 1 request are straightforward. You just need to provide some way for your client to know your server's address. You might do this by wiring it in as a defined constant, or you could type it in to the client.

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

About Timeouts

If a Request gets lost on the way to the server, a client will block forever on `recvfrom()`, waiting for a message that will never arrive. Therefore you need some way to "interrupt" the `recvfrom()` call if it has not returned within a few seconds. (It generally should not take more than a few seconds for the server to respond.)

For this purpose we can use the `alarm()` system call, and the *signal* facilities of Unix. The signal facilities provide a simple form of *asynchronous* communication. Signals are like interrupts, which may be delivered to a process at any time. When you type `^C` at the terminal, for example, to kill a running command, the system delivers a SIGINT signal to the program executing the command. There are a few dozen different signals that can be delivered to a program, to indicate various conditions.

Delivery of some signals —such as SIGALRM, the one we'll be using for this assignment— is controlled by the program, i.e. it can specify when it wants to receive the signal; others cannot be blocked. A program *can*, in most cases, specify what action it wants to take when a signal arrives, however. (Note that a signal may arrive at any point in the execution of the program, even in the middle of an assignment statement.) The possible actions are:

- Ignore the signal. Nothing happens; the process continues execution, oblivious to the arrival of the signal.
- "Catch" the signal. When the signal is delivered, a routine designated by the user called the *signal handler* is invoked. The signal handler may do (almost) anything a regular program can do. When it returns, the main program resumes execution where it was interrupted.
- Take the "Default" action, which is usually to terminate the program.

The program specifies what action it wants to take with the `sigaction()` system call. The `sigaction` routine takes a signal and a `struct sigaction` containing a pointer to the handler routine, if any, to be used. The `sigaction` structure also contains other information, including flags that describe various options. You want the options that indicate that (i) the handler should *not* be reset upon signal delivery, and (ii) blocking system calls (such as `recv()`) should *not* be restarted. Under POSIX (and Solaris) these options are the default, i.e. you can set the flags to 0 and be safe. This may or may not be true under linux.

The other fact that is important to understand about signals is that when a receiving system call (`recv()` or `recvfrom()`) is blocked waiting for something to arrive from the network, and a signal is delivered, the system call will return -1, with the error number set to `EINTR` (this is assuming you have the options set as described above).

This is how retransmission timeouts work: the program first calls `sigaction()` to arrange for `SIGALRM` to be "caught" by a simple routine, which might increment a global variable that counts total retransmissions. Then it calls `alarm(n)`, which arranges for the `SIGALARM` signal to be delivered in n seconds. It then calls `recvfrom()`. If no message is received within n seconds, the signal is delivered, the handler is invoked, and the `recvfrom()` call returns -1, with `errno` set to EINTR. Otherwise, if `recvfrom()` returns normally, the program immediately calls `alarm(0)`, which *cancels* the "wake-up call".