CS 332 Computer Networks **Spring 2018**

Project 3 — “Who” Directory Service

Assigned: February 26 *Due: Friday, March 23, 5:00 p.m.*

1 Introduction

In this project, you will write a “who” directory service. The `who` command in UNIX shows the users logged in at a machine. Figure 1 shows a sample output from the CS server `mathcs01`.

The `-H` flag was used to generate the headings above the columns. The `who` command shows the set of users who are logged in, on what UNIX terminal, when they logged in, and if not on the local machine, from where they logged in. See `who(1)` for details if you like, but you won’t need them for this assignment. Note that the who command only shows who is logged on this machine (there is also a `rwho` command for checking who is logged in to a remote machine).

In this assignment, we will devise a protocol to list the set of users logged in not only on a single machine but on a pre-defined set of machines. You will write a server that maintains a simple database of users logged in to various machines.

It is obvious that for this to be of any practical use, the database needs to be updated periodically when users log on and log off hosts. We will not worry about the database updates in this project. Instead, the set of users will be kept in a static file, called “database.dat” that your server can use. (database.dat is provided for you, so you can keep a copy in the directory where you code your server.)

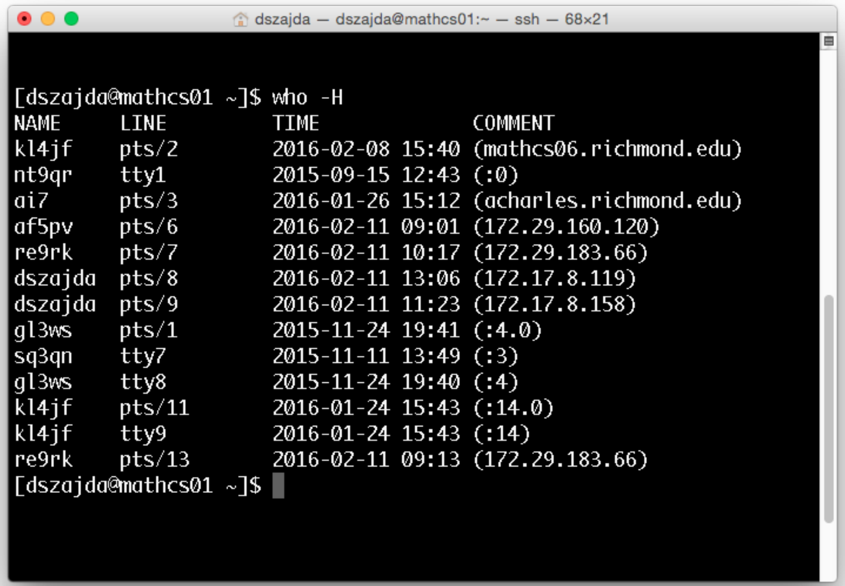
In a nutshell, the protocol works as follows: the client queries the server with a hostname as the parameter. When the server gets the query, it looks in its database for the list of users who are logged onto that machine. It then generates a packet which contains the list of users and their login times and sends it back to the client.

In this assignment, the server and the client will communicate using the User Datagram Protocol (UDP). UDP is an unreliable protocol, and unlike TCP, does not guarantee that the all of the data you send will show up at the other end.

You will need to implement two new mechanisms:

- **Re-transmissions from the client:** Since data transfer in UDP is not reliable, you will need to create a re-transmission mechanism in the client for queries that do not lead to response from the server.

- **Checksum computation:** To ensure that the data received at either end has not been corrupted in the



```
[dszajda@mathcs01 ~]$ who -H
NAME     LINE         TIME             COMMENT
kl4jf    pts/2        2016-02-08 15:40 (mathcs06.richmond.edu)
nt9qr    tty1         2015-09-15 12:43 (:0)
ai7      pts/3        2016-01-26 15:12 (acharles.richmond.edu)
af5pv    pts/6        2016-02-11 09:01 (172.29.160.120)
re9rk    pts/7        2016-02-11 10:17 (172.29.183.66)
dszajda  pts/8        2016-02-11 13:06 (172.17.8.119)
dszajda  pts/9        2016-02-11 11:23 (172.17.8.158)
gl3ws    pts/1        2015-11-24 19:41 (:4.0)
sq3qn    tty7         2015-11-11 13:49 (:3)
gl3ws    tty8         2015-11-24 19:40 (:4)
kl4jf    pts/11       2016-01-24 15:43 (:14.0)
kl4jf    tty9         2016-01-24 15:43 (:14)
re9rk    pts/13       2016-02-11 09:13 (172.29.183.66)
[dszajda@mathcs01 ~]$
```

Figure 1: Screenshot showing output of `who` command on `mathcs01`

network, on receiving a packet from the network, you should first compute the checksum to see if the data is correct. The server and client will silently ignore all corrupted packets.

2 Protocol Specification

The client will construct queries, containing a single hostname, and send them to the server. For each client query, the server will search its internal database for that hostname, and send back a (possibly empty) list of users logged on to that machine. The client is responsible for re-transmitting any query that gets lost or corrupted in the network.

2.1 Packet Header

Both query and response packets have the same header structure shown below.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |Version|Type |X|    Length       |           Query-Id            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |          Checksum               |           Data                |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                               Data                             |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- `Version (4bits)` The version field is set to binary 0110 (0x6 if you prefer) for this project.
- `Type (3 bits)` The contents of the type field are:
  - For Query packets: the type field should be 000 (binary)
  - For Response packets: the type field should be 100 (binary)
- `X (1 bit)` This bit is set to zero in the `query` packets (i.e. packets from the clients to the server) and set to one in the `response` packets *if and only if* the hostname is found in the server database. If the hostname is not found in the database, then the X bit in the reply packet from the server is set to 0. Note that it is possible for the hostname to be found, but there are no users logged on. In this case, the X bit should still be set to 1, since the hostname was found in the database! This is necessary because we want the client to be able to distinguish between the case of a bad hostname and the case of a good hostname with no users.
- `length` This is an unsigned char value that represents the number of data items contained in a packet. For a request packet, the length should be 1, since there is a single data item–the single hostname. For response packets, the length field gives the number of users found on the requested host.
- `Query-Id` This is a randomly generated 16 bit number that the client uses to map server responses to outstanding requests. The query-ID need not change on retransmitted packets.
- `Checksum` This is a ones-complement checksum of the entire packet, including header and data. The checksum computation is also explained below.
- `Data` The format of the data field changes depending on the type of the packet.
  - For the `query` packet the data just consists of the hostname as a string terminated by a null character. We will not worry about the domain name of the hosts. Thus, the data part of a query could be “mathcs02\0”.
  - For the `response` packet, the data consists of a variable length list of {username,logintime} pairs. The username is exactly eight characters long and the login time is an unsigned integer of size 4 bytes. If the actual username is not eight characters, there should be an appropriate number of NULL characters appended to the username to pad the username field so that it is 8

bytes long. There is no field separator between the username and login time—it's not necessary since each field is a fixed number of bytes.

The server will ignore any packet whose type it does not recognize. It will also ignore any packet that contains a bad checksum. In general, both the server and the client will gracefully ignore ill-formed packets. (I.e. an ill formed packet should not crash either the client or server.) Upon receiving a valid query (and only in this case), the server will query its internal database for the information requested and will return what it finds in the database.

Note: the server will (re-)construct a new packet with its answer. For queries for which the hostname is listed in the database, but for which there are no users, the data part of the response packet is 0 bytes and the X-bit is set to 1. For queries for which the hostname is not listed in the database, the X-bit of the response packet is set to 0 (and of course the data part is 0 bytes). Of course, all of this was explained in the `X` bit description, but I feel it needs reiterating. The client is responsible for mapping the answer to the original query using the `Query-Id`.

2.2 Retransmission and Checksums

If the server does not reply within a pre-defined time period, the client will re-try its query. The server is completely stateless, meaning that it does not *remember* which queries it has served, to whom it has served them, and when it served them. If it receives the same query from the same client, the server will answer as appropriate in response to each query.

Note: Clients *must* be able to disambiguate stale responses from the server. Thus, if a client gets a response which does not match the query-id, it should ignore this response and *not* reset the retransmission timer. In general, if the client receives a response that is bad in any way (i.e. bad checksum, packet is from an address other than the server, the type field is incorrect, etc) the client drops the packet silently and does not reset the retransmission timer. For example, if the retransmission interval is 4 seconds, and the client receives a bad response 1.5 seconds later, it should act as if it never received the bad packet, and wait another 2.5 seconds to retransmit (unless it receives a good packet in the interim).

The checksum contained in both the query and the response is a ones-complement sum of all the contents in the packet, computed just as the checksum is computed in an IP packet (see RFC 1071, or our class lecture notes). Compute the checksum in the following manner:

- While sending a packet:
  - Initialize the checksum field to 0x0
  - Consider the entire packet as an array of 16 bit unsigned integers and compute the ones-complement sum of the array (which is equivalent to an XOR). If there is a carryover (from bit 16), add one.

The result is put in the checksum field. Note that if the packet is an odd number of bytes long, then a pad byte consisting of all zeros should be appended to the packet.

- When you receive a packet
  - Run the algorithm above on the entire packet, including the checksum
  - Accept the packet *if and only if* the resulting checksum is zero.

3 Deliverables

You will have to write both the server and the client for this assignment. I am providing you with binaries of both a working server and a working client so that you can see the project in action and so that you have programs with which you can debug both your client and server. Of course you can also use the StdoutEchoServer for testing of individual messages. I am also providing you with the files Database.c (providing database access functions – see below), database.dat (the database file), and WhoHeader.h, which you may modify as you see fit. You will also want to include NetworkHeader.h, though I will not provide it since you should already have one. Of course, you are expected to provide a README file, a Makefile, a testing report, all of which should be included, with all the code, etc., in a single tar file called XXXProject3.tar, where XXX is replaced by the first three letters, in uppercase, of your last name. **Remember, that I should be able to extract the contents of your tar file into an empty directory, build the executable, and run without having to add any additional files. Thus you should include the database sources and other code and headers provided in your tar file!**

The command line arguments for the server and client are as follows:

- Server:
  `Project3Server -p ⟨port⟩ -d ⟨database-file-name⟩`
  - port : Port number at which server will run.
  - database-file-name : Name of the database file name to use for host $\leftrightarrow$ $\{$ username, logintime $\}$ map.
- Client:
  `Project3Client [-h ⟨serverIP⟩] [-p ⟨port⟩] -t ⟨timeout⟩ -i ⟨max-retries⟩ -d ⟨hostname⟩`
  - serverIP: IP address of the host on which the server is running.
  - port: Port number at which server is running.

  - timeout: Indicates in seconds, how long to wait before regenerating an unanswered query.
  - max-retries: Indicates the number of times the client will re-generate the query, before it quits, if no response is received from the server.
  - hostname: This is a string that will form the data portion of the query. It should not be longer than 64 bytes. It represents the hostname of the machine for which you would like to obtain user information.

In the client, if the hostname and the port number of the server is not specified, the defaults from NetworkHeader.h should be used.

4 Database Interface Functions

The server will query entries in the database using the following interface routines :

- `void open_database (char *file_name);`
  The server should log an error and terminate if a database with the given name was not found.
- `char **lookup_user_names (char *host_name, int * no_of_entries );`
  The hostname is passed as an argument. This function returns an array of null terminated strings and the number of users logged onto the machine is returned via `no_of_entries`. `NULL` is returned if and only if the requested hostname is not found in the database. The strings are of the form "username:logintime". The login time represents a 32-bit integer that you should convert to an `unsigned int` before passing back to the client.
- `void close_database (void);`
  Closes the database.

Note, in `lookup_user_names` the function will allocate space for these strings. Your program need not `free` the space `malloc`-ed here, as it is done through the call to `close_database()`.

5 One Final Comment

This is the first assignment requiring concurrent programming. Specifically, to implement the timeout mechanism, some kind of alarm should be used, with the SIGALRM signal being used to indicate the event that the retransmission timer expires. Since the SIGALRM signal is generated asynchronously, your client code must be prepared to catch and properly handle the signal at any point during it's execution, including times when the client may be processing a packet, but before the packet has been determined to be "good". See the TCP/IP Sockets in C text sections on asynchronous I/O (Sections 6.2.2 and 6.2.3), and particularly on the implementation of the timeout mechanism for a UDP echo client.

As with all of your projects, you are free to use any of the code in the TCP/IP Sockets in C text. Remember, though, that *you must be able to explain all part of your code to me*. For example, if you use non-blocking I/O, I expect that you realize that you are using non-blocking I/O, and that you can explain to me both what it is and how your I/O code works!

6 Submission

Project must be submitted using the guidelines on the class web page. The name of the client and server executables that your makefile builds should be `Project3Client` and `Project3Server` respectively. Also, the name of your tar file should be XXXProject3.tar, where XXX represents, in uppercase, the first three letters of your last name. **IF YOU SUBMIT AN IMPROPERLY NAMED FILE, OR IMPROPERLY NAMED SOURCE FILES, I WILL TREAT IT AS IF YOU HAVE NOT SUBMITTED ANYTHING!!!!** By now you know there is a text file, containing submission email addresses, in the content section of the course Blackboard page. As always, you should submit your project by attaching the tar file to an email sent to the appropriate address. This has the effect of dropping the tar file into one of my Box folders.