Programming Assignment 1 (30 pts)
A Client Dynamically Accessing Servers Using the Net Oracle

Due: 11:59 p.m. Monday, November 8, 2004

This assignment introduces client-server programming using the TCP/IP protocols. In this assignment, you will write both a client and a server. Your client and server communicate using TCP, and your server may implement any service you choose. The ground rules are: the server reads and writes data to and from a ***TCP connection***. The server may prompt the client for input, or simply print a random message. Once you have debugged the server, it executes in the background (even after you log out) waiting for service requests from a client.

To access a service, the client opens a TCP connection to the server, sends and receives data, and then closes the connection. Conceptually, the client acts as a pipe between your terminal and the server, copying data sent by the server to standard output and sending data read from standard input to the server. The client terminates the connection with the server when it receives an end-of-file from either the server or standard input.

This scenario omits a key aspect of client-server programming: How does the client find out *where* the server is? That is, what transport-level address (Internet address and port number) should the client use to connect to the server?

One solution is to use a **name server** that dynamically maps service names into their transport-level addresses. You have access to such a server, called ***oracle***, allowing you to register the service you provide and advertise it to other students in the class. Conceptually, the *oracle* is like the white pages in your phone book. A server registers the name and transport address of its service in the phone book, and clients use the phone book to map service names to transport addresses.

When your server starts, the operating system will assign it an unused port number (e.g., **p**) on which it can wait for incoming connection requests. The server then registers the availability of its service by sending a short message to the *oracle* containing the name of the service (e.g., ``finger'') together with the transport address (host number and port **p**). The oracle server records the name-to-address mapping in its local database.

A client wishing to connect to a server first sends a message containing the desired service name (e.g., ``finger'') to the *oracle*, and the *oracle* returns a message with the appropriate transport address. The client then opens a TCP connection to that service. Exact details for communicating with the *oracle* are described below.

Client Overview

The **client** continually responds to single line commands from standard input. The client **must handle correctly** the following three commands from standard input:

list : Send a message to the *oracle* requesting a listing of all the currently available services. The client relays the list of current services back to standard output.

connect service [uid]: Open a connection to the server providing **service**. *Service* is the user-friendly service name registered with the *oracle*. An optional argument *uid* is used to distinguish between services provided by different students having the same name. That is, multiple users may register services having the same name.

quit : terminates the client program.

When the client wishes to connect to a service, it takes the following steps:

1. Contact the *oracle* to locate the transport-level address (host name and port number) of the server requested.

2. Open a TCP connection to the requested server.

3. Copy standard input to the server and copy all data sent by the server to standard output.

4. After receiving an end-of-file from *either* standard input or the TCP connection, close the TCP connection. Note: if the client receives an end-of-file from standard input, it terminates the connection to the server, but the client continues reading additional commands from standard input because the user may want to connect to another service (see *clearerr(3)*).

Server Overview

The **server** takes the following steps when making a service available:

1. Create a TCP socket (similar to a UNIX file descriptor --- see *socket(2)* .)

2. Bind the socket to a *sockaddr_in* structure with family AF_INET, port number 0, and address INADDR_ANY. This directs the Unix kernel to accept TCP connection requests from any machine (INADDR_ANY) in the Internet, and specifying a port number of 0 (indicating ``don't care'') directs the kernel to allocate an unused port number (see *bind(2)*).

3. Extract the port number allocated in the previous step (see *getsockname(2*)), and fetch the Internet address of the host on which your server resides (see *gethostname(2)* and *gethostbyname(3).* Fill in the appropriate fields in the om structure (described below) and register the service with *oracle*.

4. Specify the backlog of incoming connection requests you are willing to tolerate (see *listen(2)*).

5. Finally, wait for a connection request and service it. When you have serviced the request, repeat the process by waiting for the next connection attempt (see *accept(2),* and *close(2*)).

Interacting With the *Oracle*

The *oracle* resides on machine ***ccc3*** at well-known UDP port ***netoracle***. All communication with *oracle* is through UDP messages containing a structure called an **om** (for ``oracle message", pronounced ``ohhmm"), whose definition can be found in the file *oracle.h* in */cs/cs4514/pub/lib* . The file is reproduced below:

```
# define luid 16
# define cchMaxServ 10
# define cchMaxDesc 40b
# define verCur 'C'

enum cmd {
  cmdErr,                       /* An error occurred. See sbDesc for details */
  cmdGet,                       /* Get the address of a service */
  cmdAckGet,                    /* ACK for cmdGet message */
  cmdEnd,                       /* Last response to a cmdGet message */
  cmdPut,                       /* Register a new service  */
  cmdAckPut,                    /* ACK for cmdPut message */
  cmdClr,                       /* Unregister a service */
  cmdAckClr                     /* ACK for cmdClr message */
};

struct om {                     /* oracle message */
  char ver;                     /* version number of this structure */
  enum cmd cmd;                 /* command/reply code */
  char sbDesc[cchMaxDesc];      /* description of service (or error reply) */
  char uid[luid];               /* user id (login id) of requester/provider */
  char sbServ[cchMaxServ];      /* name of service requested/provided */
  struct sockaddr_in sa;        /* socket addr where service is available */
  unsigned long ti;             /* time of registration */
};
# define lom (sizeof (struct om))
```

Locating a Service

To find a service, your client program fills in the fields of the *om* structure as follows:

ver: Set to *verCur* in all messages.
cmd: *cmdGet* .
uid: The Unix user id of the user offering the service. If a specific uid is not needed to identify the server, the client sets this field to the NULL string (NULL character in first byte).
sbServ: The name of the service of interest. To get a listing of all available services, set ***sbServ*** to the NULL string (NULL character in first byte).

In response to a *cmdGet* message, the *oracle* returns two or more messages. Response messages have a cmd type of *cmdAckGet*, and the last of the *cmdAck* responses is signaled by a *cmdEnd* message. *CmdEnd* messages do not contain the name of a service; they simply signal the end of the last response. If only one service matches the client's request, the server will

return two messages: a *cmdAckGet*, followed by a *cmdEnd*.

Each *cmdAck* message contains the following fields:

sbDesc: A sentence describing the service. Your server fills this field when registering a service, and the *oracle* returns it in response to *cmdGet* queries. When locating a service, the field should contain all zeros.

uid: The user id providing the service (e.g, ``rek'')
sbServ: The name of the service (e.g., ``finger'').
sa: The transport address at which the service resides.
ti: The time at which the service was registered.

Registering a Service with *Oracle*

When a server registers its service with the *oracle*, it sends an *om* message with the following fields:

cmd: *cmdPut*, to register a service.
uid: The login id of the user registering the service (see *getuid(2)* and *getpwent(3)*).
sbServ: One word name of the service (e.g., ``finger'').
sbDesc: A brief description of the service.
sa: The transport level address at which the server can be reached.

In response to a *cmdPut* message, the *oracle* returns a message of type *cmdAckPut* if the registration succeeds. In the case of errors, the *oracle* returns a message of type *cmdErr*, and sets the field *sbDesc* to contain a short explanation of the error.

Exchanging Datagram messages with the *Oracle*

Note: both the client and the server communicate with the *oracle* using UDP.

1. Create a UDP socket (see *socket(2)*).

2. Get the Internet address and port number of the oracle (see *gethostbyname(3)* and *getservbyname(3)*).

3. Open a UDP connection to the *oracle* (see *connect(2)*).

4. Send an *om* message of the appropriate type to the oracle (see *send(2)*).

5. Wait for an *om* reply from the oracle (see *recv(2)*).

Assignment Objective

The basic objective of the assignment is to build a client that can obtain a list of services from the oracle server and connect to a simple service (one that just returns output such as *daytime*). A basic server must register with the *oracle* server and return information when a client connects to it.

Additional Work

Completion of the basic objective is worth 20 of the 30 points for the assignment. For the additional points of the project, your client will need to be able to handle connecting to **multiple** servers within a session (in a serial manner, not in parallel). An advanced client works well both with services that require interaction (both input and output) as well as simple services just producing output. To obtain additional credit for your server, it **must** be interactive in that it both requires input and produces output.

General Advice

Nearly all system calls and library routines return some form of error code if the operation was not successful. **You need to check the return value from every routine for an error code.**

The following steps are recommended:

1. Start small. Write a procedure that sends a datagram to the *oracle*, and receives a datagram response from the *oracle*. If you make a mistake, the *oracle* will return an *om* with *cmd* set to *cmdErr*, and *sbDesc* will contain a brief description of the problem.

2. Write a client that connects to a trusted server registered with the *oracle*. We will register at least one simple server (e.g., ``finger") that can be used to test your client.

3. Once you have tested the client, begin work on your **unique** server.

You have about two weeks to complete assignment 1. There are three parts to the project and you should have each part done in less than one week to stay on schedule. In other words your client routine should be able to obtain a listing of all services no later than ***November 2, 2004***.

As an aid to debugging, as well as to trace use of your server, you should print trace output when connections are made. For example, you might print the host name and port number of every user that connects to your server, along with the time of the connection.

Requests to the oracle are actually a bit more general than has been described so far. Regular expressions can be used as service names or user ids to effect a kind of ``wild-card search" for services. For example, specifying a service name of ``.*" matches all services. Specifying a user name of ``..." matches all services provided by users with three-character ids. The format of regular expressions is the same as that of *ed(1)* .

Use your imagination and creativity in designing your server. ``Neat'' servers are popular for testing by clients written by your peers.

The following library procedures and system calls will be helpful*: clearerr(3) , gethostname(2) , listen(2) , close(2) , getpwent(3) , getuid(2) , send(2) , recv(2) , getservbyname(3), gethostbyname(3), gethostbyaddr(3), getsockname(2), bind(2), socket(2), connect(2), accept(2) , listen(2) and select(2)* .

For more information on how the Unix networking-related library routines and system calls, see the book *Unix Networking*, which explains how to use the Unix library routines.