CSE310 Fall 2015

Course project: a distributed name service

You are to implement a distributed name service using Internet domain **stream** sockets. You shall implement both a name client program and a name server program. The client program allows users of the name service to update and query name records that are stored at the name servers. Multiple users/clients can access the name service at the same time.

There are two stages of development. In the first stage, you build a single name server that can be accessed by multiple clients at the same time. In the second stage, you build multiple name servers which collectively store the entire name record database, which in turn is accessed by service users. At the time of submission, state at the beginning of your documentation how much of the project you are able to complete.

You are to develop this project in groups of two students.

Stage 1: Building a single name server

There is only one name server that maintains the name record database. Each record in the database contains three fields: <*name*, *value*, *type*>. An example of such a record is the DNS resource record that we discussed in Chapter 2, where each record has a type. For a type A record, *name* is a domain name, and *value* is the domain name's corresponding IP address. For a type NS record, *name* is a domain name, and *value* is the host name of the authoritative name server that is in charge of the domain name. Our name server database is initially empty; it is then updated by our name service clients.

The server is started first. It creates a listening socket and binds to port 0. This would let the system assign an available (so-called ephemeral) port to the server socket. Your server program should print the port number assigned by the system on the standard output. This port number is used when starting the client program.

The client program takes two inputs via command line arguments: (i) the host name of the machine on which the server program is running, (ii) the port number that the server is listening at. Once the client program is started, say, by you, the following commands should be supported:

- help: this command takes no argument. It prints a list of supported commands, which are the ones in this list. For each command, the output includes a brief description of its function and the syntax of usage.
- put: this command is used to add name records to the name service database. It takes three arguments: name, value, and type, in that order; they constitute the three fields of a name record. After receiving this name record information, the client sends this name record to the server, which enters the record to its name service database. If a record of the same name and type already exists, update the record with the new value.
- get: this command is used to send a query to the server database. It takes two arguments: name and type, in that order. Upon receiving a query, if a record with the name and type is found, the server returns the record value to the client. Otherwise, it returns a "not found" error message.
- del: this command is used to remove a name record from the database. It takes two arguments: name and type, in that order. Upon receiving a query, if a record with the name and type is

found, the server removes the record and sends a positive feedback. Otherwise, it returns a "not found" error message.

- browse: with no argument. This command is used to retrieve all current name records in the database. Upon receiving a browse request, the server returns the name and type fields of all records, the value field is not included. If the database is empty, the server returns a "database is empty" error message.
- exit: with no argument. Upon receiving this command, the client terminates the current TCP connection with the server, and the client program exits.

Each command above invokes message exchanges between the client and the server except for the first and last commands. The protocol for the communication between the client and the server, including the types of messages, the syntax and semantics of each type of message, and the actions taken when each type of message is sent and received at each party, needs to be designed and specified by you. You can refer to the HTTP protocol (RFC2616) and use formats similar to the HTTP request / response messages (Slides 2-26 to 2-31) for the types of messages that you define for this project. For example, you can use methods such as "PUT", "GET", and "DEL", etc. in the method field of the message that is sent to the server. The command arguments can be put in subsequent fields of the same message.

Each message sent to the server should be properly acknowledged by way of response messages. Status codes and status phrases similar to that in the HTTP response messages should be used. Some examples are 200 OK and 404 Not Found, among others. The protocol specification that you define needs to be clearly stated in the written documentation described below.

Upon receiving server responses, the client program should print properly formatted messages on the standard output to inform the name service user.

As previously mentioned, the name server and its name record database can be accessed by multiple clients/users at the same time. When with such concurrent accesses, conflicts may occur. E.g., while serving the browse request of user 1, user 2 may be adding or removing records from the database. You need to design and implement reasonable semantics to deal with such conflicts. Describe your scheme in your documentation.

Stage 2: Building multiple name servers

There are multiple name servers to provide a distributed name service. Each server is in charge of one particular type of name records. E.g., in the DNS example above, there would be one name server that maintains all type A records, and another name server that maintains all type NS records. Your project program should support at least two different types of name records, i.e., at least two servers.

To access these distributed name servers, you shall implement a third program, called manager. The manager program first reads an input file called `manager.in` in the current directory. The file contains all name record types in your name service, with one type per line. E.g., the first line is "A", the second line is "NS", and so on. Assume there are *n* record types in your service, after reading all the types, the manager starts *n* name servers, using the system calls **fork** and **exec** to run the *n* servers in *n* different processes. The manager records the record type and port number of each name server.

The manager runs on a well-known TCP port that is unique to your group. When a client program starts, it takes the host name and port number of the manager as input. The client program informs

the user that s/he needs to provide a record type that s/he is interested first. Once the user enters the type, client sends the type to the manager. Manager returns the address of the name server who is in charge of that record type to the client. Client then establishes a TCP connection to that name server and starts to update and query the database on that name server using the same set of commands as in Stage 1 except that all occurrences of the *type* field in commands can be dropped, because the client is now communicating with a known record-type server.

Thus all commands and their functions are essentially the same as before. Two new commands that a client can issue additionally are:

- type: with one argument: record type. This command allows the user to enter a record type that s/he is interested. Client can then contact the manager to obtain the address of the server serving the type. An error condition for this command could be "type not found".
- done: with no argument. This command indicates that the user is done with the current record type. As the result, the client program terminates the current TCP connection with the type server. The program then prompts the user for further commands

There are anomalous scenarios that your implementation should address. For example, a wrong port number or a non-existent hostname is supplied by the program user when starting the manager. Include such scenarios in your documentation.

Again, the protocol (message types, syntax of each, etc.) that is used for message exchanges between the client and the manager should be designed and specified by you. Each message needs to be acknowledged, with responses that contain status code and status phrase. The protocol needs to be documented in the written documentation. The server should be able to handle concurrent accesses and potential race conditions as in Stage 1. You can start with just two name servers. Indicate in your documentation how many your code can support.

Development

You may implement the project using Java or Python. You should implement a command line interface similar to Linux command line for the client program. Your manager and server programs should print adequate messages on the standard output, to convince the TA that they contain required functionalities. You program should run on the Linux machines that we used for the socket programming in Labs 3 and 4.

Python stream socket programming was covered in Lab 3 as well as the textbook. For Java socket programming, a simple example is provided in the folder. You are also free to read online tutorials.

For information on the Linux machines to use, how to access them via ssh, as well as instructions on how to choose the well-known port number for your manager program, refer to Lab 3 specification.

You are responsible to clean up any processes of yours that might be left around running as you develop, debug, and test your programs. Orphan processes consume extensive resources, slowing down both others' work and yours. Under Linux, to kill a process, type "`ps aux | grep your_id`", where `your_id` is your login id, to find the process id(s) of the orphan processes (in the second column of the `ps` command output), and type "`kill -9 pid`" to kill the process with process id `pid`.

What to submit

Submit the following two materials separately before the due date:

(a) Well-documented source code files, together with a README file.

- Each program should have proper program documentation in it. Program documentation is the comments that appear in the source code to aid in the understanding of the program. They tell what effect the code will have or to elucidate a complex statement. Do not reiterate in English what is obvious from the code such as "variable x is incremented".
- You can **only** submit the source code. No executable or object file is accepted. This means before you submit, you must make a clean submit directory that has only the required files in it. Name all Python program files with .py suffix and all Java program files with .java suffix.
- Submit a single .tar or .zip file instead of multiple files. Include in your README file instructions on how to obtain the source files from the archive or compressed file. This must be able to be performed in the lab environment as well.

(b) Written project documentation in a single file that includes the following sections:

- Overview: indicate how much of the project you have completed. A brief description of the major functionality in each part that you have implemented.
- User documentation: it contains everything that the user of your program (and the grader) needs to know to properly use the program to obtain desired results. This section does not describe how the program works, but only what it does and how to use it. You should describe the syntax and parameters used to run your programs, the program output (including possible error messages), and program limitations (for example, "at most 16 name servers can be up at the same time")
- System documentation: system documentation is for people who need to know what is going on inside the program so they can perhaps change it or add new features. It describes the client server communication protocol that you defined for this application; it describes how concurrent operations are handled. It describes major data structures and important algorithms if there are any, where to find things, error conditions, what causes them and what happens, how to compile and generate the executables. Try to be complete but concise.
- Testing documentation: include a list of testing scenarios to convince the grader that the program works. Explain why you chose the test cases, what are expected output and what are actual output.
- The written documentation should be type-set. Only pdf is accepted.

How to submit

Both the source code archive file and the documentation file are to be submitted via Blackboard Assignment. Blackboard Assignment submission instructions are at: http://it.stonybrook.edu/help/kb/creating-and-managing-assignments-in-blackboard. You must read the submission instructions very carefully, and check to make sure that your project has been submitted correctly before the deadline. You can only submit once. Only click on "Submit" after you have checked and are certain that all requirements are followed.

Due date

The project due date is 23:59pm, Wednesday, December 2. No late submissions will be accepted.

Tentative marking scheme

- 30% of marks will be allocated to the documentation, within which 20% will be on logically structured, well documented, and easy to understand code, 80% for a clean written documentation.
- 70% of marks will be allocated to the correct implementation of the specified functionality. Stage 1 carries more weight than Stage 2.