GEORGE MASON UNIVERSITY

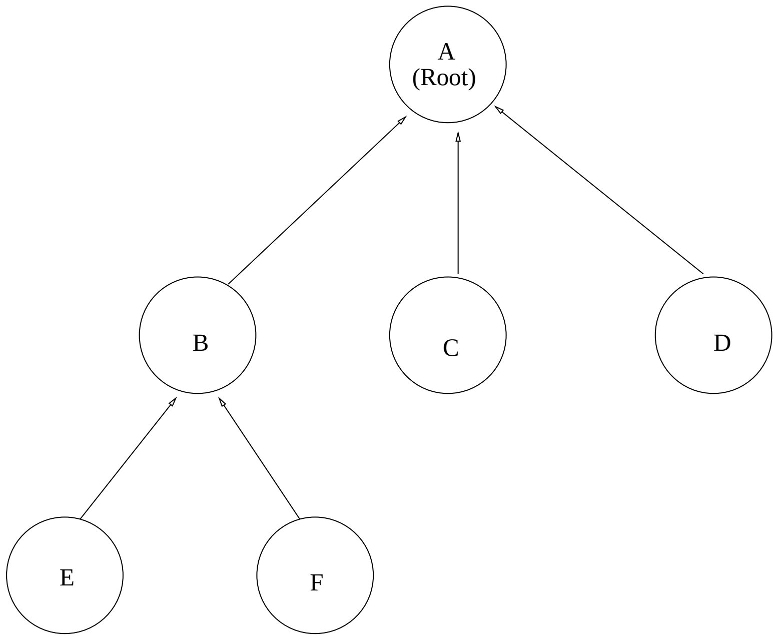
Department of Computer Science

CS 571 - Operating Systems, Fall 2013 Assignment #2
Due: Tuesday, October 22, 11:59 PM

DISTRIBUTED PHONEBOOK SYSTEM

The objective of this assignment is to provide some hands-on experience on **client-server programming using sockets and TCP/IP protocol suite in a distributed environment**. We consider a vertically-distributed phonebook system, implemented on multiple server processes (nodes). The client programs will contact the server processes for phonebook queries. A phonebook server will return matching entries, either directly from its local phonebook file, or through another ("parent") server. **In two different versions**, you will implement the server and client programs using first TCP, and then using UDP as the transport layer protocol.

Phonebook Servers: Each phonebook server will be represented by a separate process. The server processes will be organized in hierarchical fashion to form a *directed tree* (See figure for an example). In the example configuration, Server B has 2 children (Server E and Server F); and B's parent is Server A (which happens to be the root node). Each node (except the root) will have exactly one parent. A non-leaf node may have up to three child nodes.



A user will contact a specific server and send his/her query by running the *client* program. Upon the receipt of query, the phonebook server will first check its own phonebook file and return the matching entries. *If there is no match*, the server will forward the query to its parent node, and pass the reply from the parent to the user. The parent node, upon the receipt of its child's request, will follow the same steps: it will search its own phonebook and contact its own parent if there is no match.

This is a vertically-distributed system and a query may need to be forwarded to the root node if none of the intermediate nodes finds a match. In that case, the reply of the root (either the list of matching entries or "No Match" message) will be sent back to the user over the entire reverse path. The eventual reply of a server must be propagated back over the same path that was followed by the original request.

Each phonebook server process will be invoked by four command-line arguments, through the following command:

<server executable name> <phonebook filename> <client port> <children port> <parent port>

Above, <phonebook filename> indicates the name of the local file that will serve as the phonebook of that specific server in the current directory. <client port> is the port number where the server will be waiting for user queries (which will be submitted through the client programs). <children port> represents the port number where the server will be receiving forwarded queries from its child nodes[1]. <parent port> indicates the port number of the parent to which queries will be forwarded, if there is a need. For the root node, <parent port> will be given as NULL. Similarly, for the leaf nodes, <children port> will be NULL.

In a phonebook file, consecutive records will be separated by one or more blank lines. Each record in a phonebook will consist of three fields separated by newline characters:

<Last Name>
<First Name>
<Phone Number>

A phonebook file will have at most 32 records. In each record, the last name and first name will be at most 16 characters each, and the phone number will have at most 10 digits.
Note: The "<" and ">" characters should not appear in the command-line invocation string, or in the phonebook files.

Client Programs: A client program will connect to only one server when it is invoked. The port number of the server to which the client is connecting will be supplied as a command-line parameter (note that this number must correspond to <client port> parameter provided when the corresponding server is initiated). Once invoked, the client program will wait for a user query, contact the server, display the reply and wait for the next query. It will exit upon the receipt of exit command from the user.

In each query, the last name and/or the first name will be supplied by the user. Note that the user may want to use the wildcard character '*' when providing the first name and/or last name. For example, 'Michael *' will match both 'Michael Smith' and 'Michael Jones'. A server should return *all* the matching entries if it receives a query that includes the wildcard character. When processing queries with wildcard characters, a server will contact its parent only if it cannot find any matching entry in its own database. For simplicity, assume that the wildcard character can appear only as the first name and/or last name (i.e., you do not have to provision for cases like 'Mich* Smi*'). Finally, a query '* *' indicates that the user wants to receive all the entries in the phonebook of the server to which (s)he is connected.

Provided that you comply with the rules above, the details of the interface between the client programs and the users (including the format of the query) are left unspecified: you can use a simple text-based interface, but it should be efficient and easy to use. Your name look-up service should not be case-sensitive ('Michael' will match both 'MICHAEL' and 'michael').

[1]Note that all the children of a given node X will connect to the same port of X.

Concurrent Execution: The server processes must be able to process multiple requests simultaneously. That is, multiple clients should be able to connect to the same server at the same time. All such clients will connect to the same port (<client port>) of the server. If a server forwards a query to its parent, it should not delay processing any new queries from other clients while waiting for the reply of its parent. This requires designing servers that can process multiple requests in concurrent fashion. For this purpose, you will need to create a new (child) process or a new thread whenever a new request is received by a specific server. In order to allow the concurrency test, each look-up operation on a phonebook file should be delayed by 4-5 seconds on the servers.

Programming Language/Environment: For this assignment, you can use C or Java as the programming language. Your project must be developed on (or ported to) a Unix/Linux system. Your program *must* compile and run on either mason.gmu.edu or zeus.vse.gmu.edu – it will be tested and graded on those systems. Berkeley Sockets and Java Sockets are also available on VSE lab machines. **Do not use socket multicast/broadcast**. The use of middleware solutions (e.g., RPC or RMI) is not allowed in this assignment.

Run-time Details: In this assignment, you will create a unique process to represent each server on the **same** computer. You should be careful when choosing port numbers for your servers. All port numbers below 1024 are reserved and the ports used by other services/students may not be available either. Make sure to 'close' every socket you use in your programs. If you abort a program, the socket may still hang around and the next time you bind a new socket to the same port you previously used (but never closed), you may get an error.

During testing, we will invoke each server and client program in a separate window. Whenever there is a query-related communication between a parent and child server, the parent and child must print on their consoles short but informative messages (e.g., "Query forwarded to parent", "Query received from child X", "Reply received from parent", "Reply sent to child X"), to allow run-time monitoring. While managing sockets, you can assume that servers/clients never crash during execution.

Submission: Submissions will have both soft and hard copy components.

The soft copy components will be submitted through *Blackboard*. You will need to submit all the softcopy components (listed below) as a single compressed *zip* or *tar* file. Submit your compressed file through the Blackboard by Tuesday, October 22, 11:59 PM. Before submitting your compressed file through the Blackboard, you should rename it to reflect the assignment number and your name. For example, if Mike Smith is submitting a tar file, his submission file should be named as: assignment2-m-smith.tar. On Blackboard, select the Assignments link, then the Assignment 2 folder. In addition to the assignment specification file, you will see a link to submit your compressed tar or zip file. *Please do not use Windows-based compression programs such as WinZip or WinRar* – use tools such as zip, gzip, tar or compress (on Unix).

The soft copy components that need to be submitted in a single *tar* or *zip* file are as follows:

- a README file with:
  - information on whether you prefer that your programs be tested on *mason* or *zeus*
  - clear instructions on how to compile and run your server and client programs
  - a statement summarizing the known problems in your implementation (if any)
- The source codes of your programs (the client and server programs for both UDP and TCP –that makes a total of four programs). Each source file should include in the first commented lines your name and G number.
  IMPORTANT: Each of the four programs should have its separate source code.
- a write-up describing/explaining high-level design of your programs.

You should also submit the hard copy of all the components above to the instructor at the beginning of the class meeting on Wednesday, October 23, 4:30 PM. Failure to follow the submission guidelines may result in penalties.

Resources: You are strongly encouraged to refer to Socket/Network Programming links available at the CS 571 Blackboard site (follow the Useful Links after choosing the Resources option on the left-menu). Please direct your programming and system questions to CS 571 TA Changwei Liu (*cliu6@gmu.edu*), using the keyword "CS 571 Assignment 2" on the subject line. Make sure to check thorougly the web tutorials and resources provided at the address above, before contacting the TA. The CS 571 TA prepared also a web page with sample socket programs that compile and run on GMU systems; this can page can be accessed at `http://mason.gmu.edu/~cliu6/cs571/cs571.html`. The CS 571 TA holds office hours on Monday 3:30-5:30 PM and Thursday 4:00-6:00 PM in Engineering Building, Room 4456. Conceptual questions about the project can be directed to the instructor (*aydin@cs.gmu.edu*).

Suggestions: First of all, allow sufficient time for your project unless you have prior experience in socket programming. As the first step, you will have to study the basics of socket communication by reading Web tutorials and examining/writing simple (non-concurrent) client-server codes. Then you can incrementally develop your system, by adding multiple servers, providing concurrent execution capability on the server side, designing the user interface module for the clients and so on. The TCP and UDP components will have equal weight in grading.

Late and Incomplete Submission Penalties: Both the soft and hard copy components should be submitted by the specified deadlines to avoid the late penalty. Late penalty for Assignment 2 will be 15% for each day. You can re-submit your programs as many times as you want; we will grade the very last submission with the corresponding late penalty (if applicable). Each re-submission must contain all the required components as indicated above. In case that you re-submit the soft copy after the deadline, make sure to re-submit the hard copy components as well. Submissions that are late by five days or more will not be accepted.