Operating Systems (PG) CSE531
Assignment # 2 – Gnutella Hybrid P2P File Sharing
Deadline: 19th September 17:00 PM

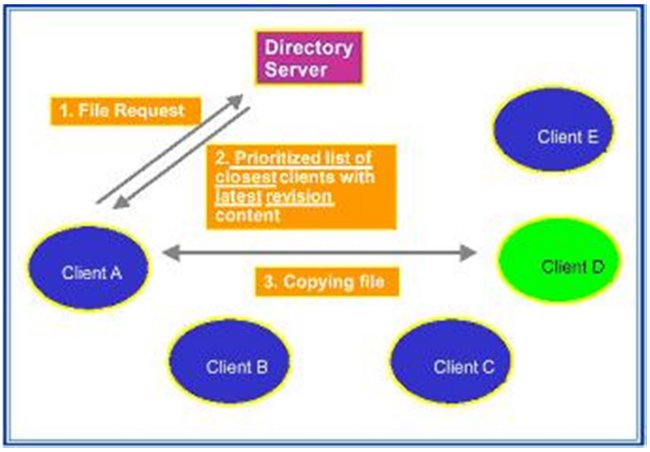
Objective :

Create a hybrid P2P file sharing system over the LAN using socket programming and implement a well-defined RPC mechanism to run and fetch results of unix shell commands executed on remote machines.

Prerequisites :

1. A fundamental grasp of networks (theory).
2. A firm understanding of the technicalities involved in linux socket programming for establishing connections across machines (TCP and UDP).
3. A basic understanding of multi-threaded and multi-process software architectures.

The broad architecture of the file sharing implementation is shown in the figure below.



The tasks of this assignment can be divided into three major parts.

PART 1

Building the Central Repository Server

The Central Repository Server (CRS) is a **file listing** server, i.e. it is responsible for storing and maintaining updated information of file locations over the network. It does NOT store the files themselves or act as a file download server. The objective is to design and build the CRS which would handle access to the central shared repository by distributed clients.
The central repository consists of two files:

1. The main repository file which maintains a mapping of files to machine IP addresses on which the files are actually stored and can be downloaded from.
2. A dynamically updated file that keeps track of all client computers (client aliases and IPs) connected to it currently.

The exact formats of the files will be provided during the tutorials. Following the provided format exactly is absolutely compulsory.

Specifications of the CRS

1. **Search for a file** : Clients must be able to query the CRS to obtain the location of a particular file. Multiple clients (mirrors) may host the same file and all such entries must be returned back.
    a. The search request itself from the client may have keywords / aliases associated with the file. Thus mapping the file to such keywords must also be performed so that any query for a particular keyword must point to the correct file.
2. **Share** : Any client must be able to add a new file entry in the central repository.
3. **Deregister** : Any client must be able to remove a file listing associated with it. If there are multiple mirrors associated with the particular file, ONLY the entry corresponding to the requesting client must be removed.
4. **Logging** : Maintain a separate log file which logs all the incoming and outgoing events at the server. Name the file "repo.log" and place it in the same directory as the executable. The logging format to be followed is listed below.
    a. Log all events on a new line.
    b. <timestamp>: <event -description>
    c. Example:
    <timestamp>: Search request from 10.1.1.2
    <timestamp>: Search response send to 10.1.1.2
    <timestamp>: Share request from 10.1.1.2
    <timestamp>: Share ack sent to 10.1.1.2

Note 1 : You are free to implement a WIRE protocol of your own design for communication over the network and participating entities. However, the external interface at the client side MUST follow the pre-specified format which will be shared with you during the tutorial session.
Note 2 : Error handling is a crucial aspect requiring considerable focus. Graceful handling of network errors, termination of program, etc. is desirable.

PART 2

Building a client

The clients communicate with CRS to obtain information about file locations so as to initiate the actual download of files. The clients themselves double up as servers, listening for incoming requests from other clients to download files located in the client machine. The objective is to

write the client program which will provide a menu driven interface (according to strict specifications which will be shared during the tutorial) to do the following:

1. **Search** : Initiate a search request to the CRS for a specified file. The search feature will allow the user to specify keywords / actual file names, which will be relayed to the CRS. The result returned would be a list of options (mirrors) from which the file can be downloaded. Once a mirror is selected, the file must be downloaded from the selected mirror.
2. **Share** : Add a file to the central repository by communicating with the CRS.
3. **Deregister** : Remove a file listing associated with the client at the central repository.
4. **Server** : The **SAME** client program acts as a file download server and should allow for other clients to download files parallely. This can be achieved in a multi-process or multi-threading environment.
5. Maintain a separate log file which logs all the incoming and outgoing events at the client (especially the download server component of the client). Name the file "client.log" and place it in the same directory as the executable. Log format is the same as for the CRS log.

Note : You are free to implement the WIRE protocol of your own design for communication between clients in their "server" and "client" roles respectively.

PART 3

Building the RPC functionality

Whereas the above two parts deal with files, this part deals with RPC communication between clients. Any client can communicate with another client to execute and fetch the results of unix commands. The objective here is to implement an RPC mechanism which allows for the execution of unix commands on a remote client via your client. The RPC mechanism must perform the following major steps.

1. Query the CRS for all the clients that are currently up and available, and select one of them to initiate RPC.
2. Implement a simple **stub** in the client program which performs the task of marshalling the data to be sent across the network (in this case the actual command to be executed). The stub is also responsible for establishing a connection with the server-side stub (skeleton) for passing the request and receiving the response.
3. At the server-end (remote client machine to which the request is being sent), the stub (skeleton) listens for incoming requests. The **skeleton** un-marshals the received message and passes the argument to the main client program. The client program validates and runs the received command on the machine and captures the output. The output is again marshalled and sent back to the requesting client by the skeleton.
4. The functionality of the client needs to be extended to send regular **heartbeat** messages to the CRS to mark itself as being available for communication over the network.

Note : Protocol for communication between the stub and skeleton is left up to you. Ensure that the flow sequence mentioned above is followed.

Pointers :

1. Learn how to program a simple socket connection in C++. Test by sending and receiving messages between client and server.
2. Read more about fork() call. Question yourself as to why is it required in the case of the client program. Can we do without it?
3. Write sample codes to append / edit data in files in C++. This will help in updating the repository at the CRS as well as maintaining log files.
4. The WIRE protocol mentioned above is just the way you would distinguish different incoming requests INTERNALLY. For e.g the CRS would listen for two kinds of requests namely 'Search' and 'Share'. One way to do this is to send messages as follows:

   search#@#music

   share#@#./test.txt

   Here, '#@#' is a delimiter. At the receiving end you can split the message using '#@#' as a delimiter and distinguish among different types of messages and perform tasks accordingly.

Languages allowed : C/C++ only