CS 582 Programming Assignment 1
Distributed Application for Patent Data Searching

Due: 11:59 pm, September 30, 2014

Introduction

For this assignment, you have been provided with the U.S. patent data consisting detail information of almost 3 million U.S. patents granted between January 1963 and December 1999. In addition this data also includes the over 16 million citations made to these patents between 1975 and 1999. This data is collected from the National Bureau of Economic Research (NBER), http://www.nber.org/patents/ and includes five files *Patents.txt, Inventors.txt, Citations.txt*, *Assignee_Names.txt*, and *CUSIP_Match.txt*. The schema of the records stored in these files is given below. For this assignment, we will consider only the following two files: *Inventors.txt, Citations.txt*.

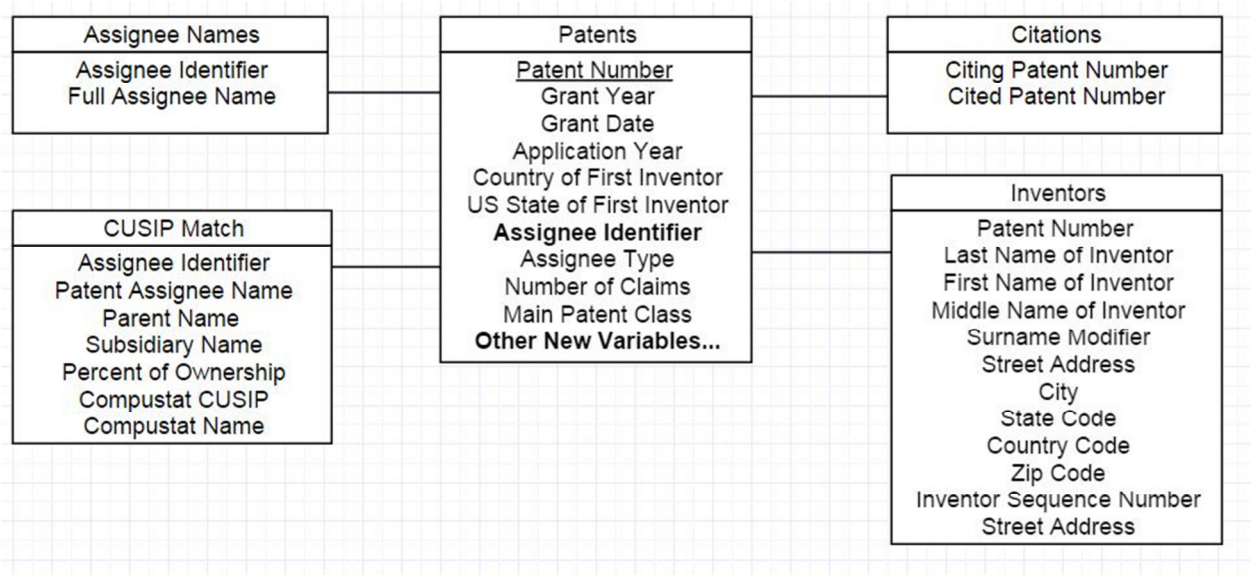


Figure 1: Schema for provided US patents data

You are required to build a distributed application to query the patents data files. Specifically, you need to implement three types of queries on data.

I. Find all the citations of a given patent. You will query *citation.txt* file and return list of all the cited patents against the given patent id.
   For example, find all the citations of the patent having patent id 3858280.

II. Find all the patents of a given inventor. You will query *inventor.txt* file and return list of all the patent ids against the given inventor name as <Last Name, [First Name, Middle Name]> i.e. Last Name is Mandatory; and First Name and Middle Name is optional.
   For example, find all the patent of the inventor name “Stevens,Peter,S.”.

III. Find all the citation of a given inventor. You will query both *inventor.txt* and *citation.txt* files and return list of all the cited patents against the given inventor name as <Last Name, [First Name, Middle Name]> i.e. Last Name is Mandatory; and First Name and Middle Name is optional.
   For example, find all the citations of the inventor name “Torrico,Jorge,L.”

The goal of this assignment is to create a distributed application that can run across different machines and query the U.S. patent data files provided to you. You will use the concepts of concurrency, threading, and client/server communication protocols to solve this task.

Design and Implementation

The system consists of three programs, the **worker_client**, the **request_client** and the **server**. A *request_client* sends a query to the *server*, and the *server* is responsible for dividing the job into parts and allocating those parts to *worker_clients*. The clients and server communicate with each other using the sockets API to send UDP packets. The protocol they must follow is detailed in the following sections. The server must be robust to communication failures (clients never receive a message) and to client failures by eventually assigning incomplete jobs to other clients.

Server

The server should recognize the two different clients i.e. request_client and worker_client. It should keep track of the clients and which jobs they're currently working on (worker_clients) or what job they've requested to server (request_client). It should handle timeouts. You can use multi-threading or *select* to do so. Using select provides an efficient and clean way to handle timeouts. The server may allocate jobs as it sees fit. It is your job to ensure that the jobs are of reasonable size and that the load is balanced across the worker_clients. The server should do this by having the number of workers assigned to each request be as equal as possible, without pre-empting workers. For example, there are 10 workers, which are all working on one request, and a new request comes in. As the workers finish their jobs, they are assigned to work on the new request, until each request is being handled by 5 workers.

Your code should produce an executable file named "*server*" and should use the following command line arguments:
./server <port>
Example: ./server 10440

For ensuring that the port number of the server process is unique for each student, please use the following formula to compute the port number of the server process.
*Port = 10000 + (last four digits of your registration number) *20*
For example if you registration number is 13030022, the port number that you should try first is:
*Port = 10000 + (0022)*20=10000+440=10440.*
In case this port number is already used, try the next port number (10441), until you found an unused port.

Worker_client

The worker_client should be multi-threaded to facilitate communication with the server. One thread should be used to do the actual work of the client (performing search operation on the data) and a second thread should be used to listen for messages from the server. The client should be independent and should be able to find out everything it needs to know through the protocol defined below. The client should not remember jobs it's worked on in the past.

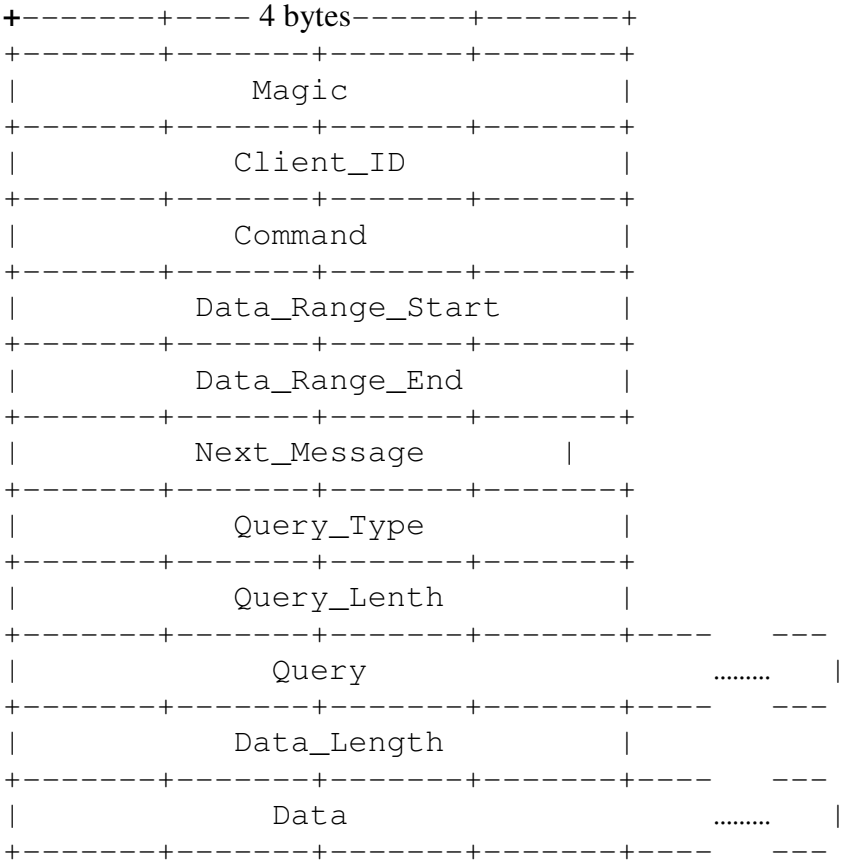
Your code should produce an executable file named "worker_client" and should use the following command line arguments:

./worker_client <server hostname> <port>
Example: ./worker_client compute-0-2 10440

Request_client
The request_client does not need to be multi-threaded, but it certainly can be if you like. Its job is to send the query to the server and display the results back to the user. It is also required to ping the server every 5 seconds to remind the server of its existence. Since we do not know how long it will take the server to find the answer, this approach is more feasible than defining an arbitrary timeout value.

Your code should produce an executable file named "request_client" and should use the following command line arguments:
./request_client <server hostname> <port> <query_type> <query>
Example:
./request_client compute-0-3 10440 SEARCH_PATENTS_BY_INVENTOR <Last Name, [First Name, Middle Name]>
./request_client compute-0-3 10440 SEARCH_CITATIONS_BY_PATENT_ID 3858280
./request_client compute-0-3 10440 SEARCH_CITATIONS_BY_INVENTOR Wilson,Pryce

Protocol
For this assignment, you will implement a binary format protocol described below.



Data Types:

Magic: unsigned int (4 bytes)
MAGIC refers to a magic number, so that we are able to identify whether a message is relevant to our application. Our MAGIC number is 14582.

Client_ID: unsigned int (4 bytes)
The Client_ID of worker_clients is the ID of the client randomly generated by the server when the worker sends a REQUEST_TO_JOIN. When the worker sends a REQUEST_TO_JOIN, it sets the client id field to 0. When the server responds (with a JOB), the server sets the client id field to a randomly generated number. The worker_client should use that number from that point on.

The client ID of request_clients is the ID of the client randomly generated by the server when the request_client sends a QUERY. When a request_client sends a QUERY, it sets the Client_ID field to 0. When the server responds with an ACK_JOB, the server sets the client ID field to a randomly generated number.

Command: unsigned int (4 bytes).
The command is the command that is being sent. All commands represented as number are listed under "commands" section below and these commands explains what to do in every possible situation.

Data_Range_Start: unsigned int (4 bytes)
Starting line number in data file to search the data.

Data_Range_End: unsigned int (4 bytes)
Ending line number in data file to search the data.

Next_Message: unsigned int (4 bytes)
This variable indicates that if data, to be sent, is more than the maximum message length i.e. 2048 bytes and remaining data will sent in next message and so on. If there is another message after this message, this field will be set to '1' else it will contain '0'. You can use this field whenever you encounter more than 2048 bytes of data and you need to send this data in multiple messages

Query_Type: unsigned int (4 bytes)
Type of the query you are sending to the server. For this assignment, there will be three types of queries represented as number i.e. 1. SEARCH_PATENTS_BY_INVENTOR, 2. SEARCH_CITATIONS_BY_PATENT_ID, 3. SEARCH_CITATIONS_BY_INVENTOR. These queries will be performed on *inventors.txt* and *citations.txt* files as mentioned in introduction section.

Query_Length: unsigned int (4 bytes)
Length of the data in the "Query" field of the protocol. The length field will be used to extract the actual query on the receiver side.

Query: Char (*Query_Length* bytes)
Query field will contain the actual query requested by the request_client. For this assignment, this field can contain the patent_id or inventor name represented as <Last Name, [First Name, Middle Name]> where Last Name is mandatory, First Name and Middle Name is optional.

Data_Length: unsigned int (4 bytes)
Length of the data in the "Data" field of the protocol. The length field will be used to extract the data on the receiver side.

Data: Char (*Data_Length* bytes)
This field contain result of the query after performing the query operation on the data files. This will contain the list of patent ids or citations for this assignment.

Commands:

REQUEST_TO_JOIN: The worker_client starts the relationship with the server by sending this command. There is no other info for this message.
Command Code: 0

QUERY*:* When a request_client wishes to make a request, it sends this command to the server with parameters: "QTYPE QLEN QUERY" Where QTYPE is the type of the query, QLEN is the length of the query and QUERY is the actual query sent to the server.
QUERY SEARCH_PATENTS_BY_INVENTOR 12 Wilson,Pryce
Command Code: 1

JOB: When the server wants to assign a job to a worker_client, it sends this command, along with the range of data, query type and query. For example: "X Y QTYPE QLEN QUERY" where the X represents the value of Data_Range_Start and Y is the value of Data_Range_End, and the client needs to search all the records in the range [X, Y] including the end points in the data file. The QTYPE is the type of query, QLEN is the length of query and QUERY is the actual query sent in the command.
For example:
JOB 10000 12000 SEARCH_PATENTS_BY_INVENTOR 12 Wilson,Pryce
Command Code: 2

ACK_JOB: When the worker_client receives a JOB, it should acknowledge it with an ACK_JOB with these contents: "X Y QTYPE QLEN QUERY" where the X, Y, QTYPE, QLEN and QUERY are the same as discussed above. The server should also use this command to communicate to the request_client that it received the request. The other info field does not need to contain anything, but the client id field should be filled in (by the server).
Command Code: 3

PING: Used for request_clients to ping the server, and for the server to ping worker_clients.
Command Code: 4

NOT_DONE: When a worker_client has received a PING, but has not finished processing the job, it sends this command to the server with parameters: "X Z NEXT_MSG DLEN DATA", where X is the start of the range the server assigned and Z is the latest record the client has tried to match with the query. This means that the client has so far checked the range [X, Z]. If DLEN is zero it means that server does not found any matching record so far and if it's greater than zero, it means that server has found some data against the query so far. If the message size becomes greater than 2048 bytes than you have to send data in multiple messages that will be sent successively. The next message field will be set to '1' in all such messages except in the last message. The reason for providing the latest value Z is that if the client fails and the job is assigned to a new client by the server, the new client does not start the job from the beginning of the range, rather it tarts from the last Z value received by the server from the old client before its failure.
NOT_DONE 10000 11274 0 (Processed lines from 10000 to 11274 and does not found any data)
Command Code: 5

DONE_NO_DATA_FOUND: When the worker_client completes its job and has not found any data matching to the query, it sends a DONE_NO_DATA_FOUND with parameters: "X Y". It means that worker_client searched the range [X, Y] and does not found any matching data. If the server is not able to find any record as result of query, it sends the request_client a DONE_NO_DATA_FOUND with: "QTYPE QLEN QUERY", Query Type, Query Length and Query is included in the message. It means that server has processed the query sent by the request client and does not found any matching data.
Command Code: 6

DONE_DATA_FOUND: When the worker_client completes its job and has found matching records or the server wants to send the result of query back to the request_client, it sends a DONE_DATA_FOUND "NEXT_MESSAGE QTYPE QLEN QUERY DLEN DATA" where QTYPE is the type of the query requested, QLEN is length of Query, QUERY is the actual QUERY, DLEN is the length of data matched and DATA is the records fetched as the result of query. NEXT_MESSAGE indicates weather there is another message to send the data or the message size with data is less than 2048 bytes.
Command Code: 7

***CANCEL_JOB*:** When request client wanted to cancel the request it sends CANCEL_JOB command to the server along with its Client_ID. The server forwards the CANCEL_JOB command to all the worker clients working on the job associated with the given Request client. Upon receiving the CANCEL_JOB command each worker client cancels the current job and wait for the new request.
Command Code: 8

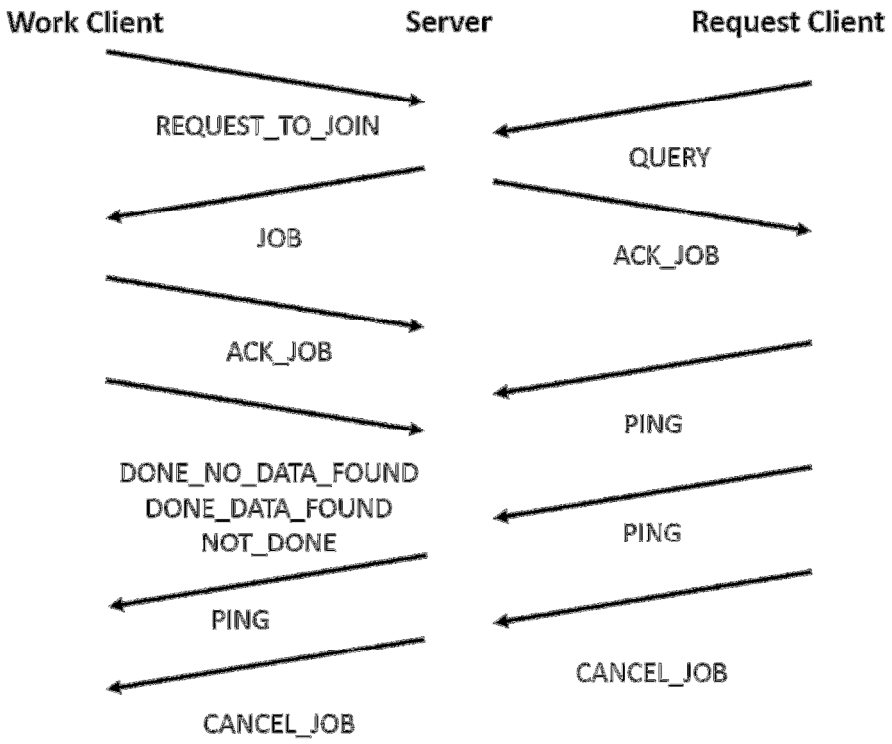


Figure 2 Protocol Working

Worker_client response to server's ping with: (a) NOT_DONE when the worker_client has not completed processing and if server does not found any matching record so far, DLEN will be zero and it finds any match data, it sends the data in NOT_DONE message with DLEN as the length of fetched data so far; (b) DONE_DATA_FOUND when the worker_client has completed processing and has found the data; (c) DONE_NO_DATA_FOUND when the worker_client has completed processing and has not found the any data.

You will notice in the figure above that the protocol is centered around the server. We assume that the server will not fail. That's why the server does not need to respond to the pings from clients.

Failure

If a worker_client fails to acknowledge the receipt of a job (times out), the server should attempt to send the job twice more. Once it has tried 3 times and the client has not acknowledged the job, the server should remember that job and assign it to the next available client and forget all about the previous client.

If the worker_client fails to send a message once it's completed a job (times out), the server should send a PING to the client. The worker should respond with its previous DONE message (whichever variant it might have been). If the worker is currently processing a job, it should respond with NOT_DONE. If a server receives the NOT_DONE response, it should reset the number of pings it sent to the worker, and wait for timeout again.
If the PING times out, the server should try twice more and then reassign the job to another worker_client and forget about this previous client.

The request_client is responsible for sending a PING message to the server every 5 seconds. The server should monitor these pings and if it does not receive a PING from a request_client in 15 seconds, the server should assume that the request_client died, and remove its request from the queue.

The timeout for all worker_client/server messages is 3 seconds. This means that the server should assign a job, and wait up to 3 seconds for the ACK_JOB. If it does not receive an ACK_JOB within that time period, the client has timed out and the server should try to send the job again. Once the server receives the ACK_JOB, it should wait 3 seconds before sending a PING.

Other Notes

I. Correct Implementation of Protocol

A significant percentage of the grade for this assignment depends on the correct implementation of the protocol. For testing your assignment we will design our own server, request_client and worker_client so make sure you follow the protocol defined in assignment. For evaluating packet loss and node un-availability we will induce errors in our workers to test your servers and vice versa.

II. Programming Language

You must write your server and clients in C. Remember that any numeric data that you send that are longer than 1 byte must be in network byte order.

III. Cluster Configuration

You can use the following compute nodes in the cluster: i) compute-0-5; ii) compute-0-6; iii) compute-0-7. Run all your client or server processes on compute nodes. Do not run your processes on rustam.lums.edu.pk.

IV. Data File Location

Data files are stored in the directory /home/basit/Assign1_Data/. Your program should read these files from this directory. Do not copy theses files to your local directory on Rustam as we do not have much space on the cluster.

V. Submission

When you are ready to submit your programming assignment, put all the files (.c, .h, Makefile, etc.) to be submitted in a folder/directory "PrgAssign1_[your login]." For example, if your login is "13030022" put the file in the folder/directory "PrgAssign1_13030022". Compress the folder into a zip file, and upload the file on the submission page on LMS.

VI. Policy against use of unfair means

We have zero tolerance policy against use of unfair means including plagiarism and cheating. Any such case will be reported to the disciplinary committee. For plagiarism detection, we will run the MOSS tool against all the submissions received as well as the programming assignment submissions of previous years.