Operating Systems I

Search engine simulation

Ariel University

Department of Computer Science

Dr. Elad Aigner-Horev

Contents

Abstract. In this assignment you will be simulating the functionality of a search engine. You will implement both the client and server side. The assignment puts an emphasis on multithreaded programming and the synchronisation of these which is the main theme of our course. The assignment involves the use of *sockets*.

§1. Introduction

In what follows a general description of a software system simulating the functionality of a search engine is provided. The description intentionally avoids delving into concrete implementation details. It is here that your ingenuity comes into play as you spend the entire semester planning the design of your system and later on implement it in a manner that conforms to the description given here.

The assignment will be graded via an oral examination to be conducted with one of the TAs in the course. In preparation for the examination please have the following in mind while developing your system.

1. *Design:* You will have to explain and defend the design of your system. We shall take interest in the modularity and robustness of your design. Design the classes and their relationship carefully. Make sure that the software entities you define are granted access only to data that they need to perform their task.

   - We are not looking for a program that works. We are looking for a well-designed program that works.
   - Please erase the phrase *"but it works"* from your lexicon as it will not be respected.

2. *Efficiency:* We leave the decision as to which data structure to use up to you. However, your choices must provide efficient solutions to the various algorithmic problems you encounter throughout the assignment.

3. *Proper coding:* You may only use Java to code up your assignment. Your code must conform to common coding conventions and paradigms of the Java community.

§1.1 Submission. Please consult the course website for instructions on how to submit your assignment and when.

§2. Preliminary description

We distinguish two entities. A *client* program and a *server* program. Each operates in a separate process. Throughout there is only *one* server process and there is no bound on the number of client processes (which is the situation that say Google has to face). The client processes do not communicate with one another. All communication is conducted between the server and its clients. The communication is done through (network) *sockets*. If you are unfamiliar with the Java Socket API then it is your responsibility to study the relevant API on your own. A good place to start would be [1].

Roughly speaking, a client process sends search requests or queries to the server through a predefined socket in an infinite loop. Each client request is processed by the server which returns a reply. In our small simulation a search request will simply be a number chosen at random from some range. The response will also be a number. Below we make this precise. Let us do mention already here that we will not draw the numbers uniformly at random. Instead we will have prescribed probabilities for each number in the range.

The server keeps a database of all requests and their replies. To simulate this database we shall use ordinary files. (This is surely not the way this is done in the real world but the student body is yet to be properly exposed to databases at this stage of the program). That is, the server maintains a set of files in which it saves triples of numbers (x, y, z) where x is the request, y is its reply, and z is the number of times that x was requested so far. Let us explain the need for z. The server keeps track of z in order to maintain a *cache* of most frequently asked requests and replies. The cache is kept in the main memory and not using files making queries to it much faster. If the cache is maintained in such a way that frequently requested numbers are kept in the cache then querying the cache makes it possible to avoid searching the database files repeatedly. Indeed, repeated queries to the files will harm the performance of the system. This will be made precise below.

During examination we shall first run the server process and then we will create an arbitrary number of client processes. The server listens on a single socket port for client queries/search requests. All clients send their search

queries/requests through this single port throughout the application. The server may listen on other ports for other purposes, say for various logistical purposes; however, for search queries there is only one port.

For each incoming search request the server assigns a separate thread to handle the request and return the reply. We refer to the latter as a *search thread* (S-thread, hereafter). Naturally, we cannot allow the server to create a large number of S-threads. Hence, we shall use a thread pool manager to organise and handle these server threads. You are not allowed to use the custom pool manager API supplied by Java. You must write your own thread pool manager. The number of S-thread the server is allowed to create is S, where S is a parameter passed to the server process through the command line upon its creation.

§2.1 The life cycle of a search request. Here is a general outline of the course that a search request takes in our system.

1. Upon creation a client process is passed two integer numbers namely R_1 and R_2 such that $R_1 < R_2$, and in addition to that the name of a file. The numbers R_1 and R_2 indicate the range $[R_1, R_2]$ from which the client can take numbers and send those as queries to the server. The file contains a sequence of numbers $(p_x : x \in [R_1, R_2])$ where p_x is the probability that x will be chosen by the client as a query. You may assume that $\sum_x p_x = 1$ and that these are arranged in the order $p_1, p_2, \ldots, p_R$. Also, each p_x will be in decimal format of up to 3 points of precision. (by the way, this is a very common job interview problem).

   Different clients can be passed different R-values and different probability file.

2. A client runs in an infinite loop. It draws a number $x \in [R_1, R_2]$ independently at random with probability p_x. Once the number x is drawn the client sends it through the socket to the server and awaits for the reply. It does not continue until a reply is sent back to it. Prior to sending the request x it prints

   Client <name>: sending x.

   When it receives a reply y for x it prints

   Client <name>: got reply y for query x.

3. Up until now we focused on the client side. Let us track the query x on the server side. When the search request x reaches the server dedicates a S-thread for the handling of x. As mentioned above, this thread is managed by a thread pool manager. (You cannot use Java's built in pool manager API, you must construct this pool manager on your own). If there are no S-threads available in the pool the socket will queue x. This is a functionality that you do not have to write up; it is provided to you implicitly through the socket.

4. Suppose that a S-thread $\mathcal{T}$ was available to pick up x. The thread $\mathcal{T}$ first searches for a reply to x in the *cache*.

   (a) In addition to its database the server also maintains a cache of frequently asked requests and their replies. The cache can hold up to at most C such associations. Each search request in the cache must have been requested at least M times by all the clients so far. The parameters C and M are passed to the server through the command line upon its creation.

   Amongst the replies that were searched each at least M times the cache only keeps track after the top C requested search requests.

   The decision as to which data structure to use for the cache is yours. Also, the decision as to which additional metadata you require in your cache is also yours. Below in Section 3 we go into more details as to how $\mathcal{T}$ actually gets access to the cache.

   (b) If the replay for x is found in the cache $\mathcal{T}$ writes it back trough the socket to the client that asked for x. Then $\mathcal{T}$ returns to its pool manager ready to process a new request.

5. If x cannot be found in the cache $\mathcal{T}$ proceeds to search for a reply to x in the database, i.e., the files that the server keeps. Below, in Section 4.1 we provide the details as to how $\mathcal{T}$ can read from the database.

   (a) If a reply is found in the database $\mathcal{T}$ writes it back to the client asking for x through the socket. In that case $\mathcal{T}$ becomes free to process a new request.

   (b) If a reply is not found in the database then $\mathcal{T}$ draws a number y uniformly at random from $[1, L]$, where L is a parameter passed to the server through the command line upon its creation, and

writes y to the database. Then $\mathcal{T}$ sends y through the socket and becomes free again. In Section 4.2 we detail the manner in which $\mathcal{T}$ can write to the database.

This has been a general description of the life cycle of a query x inside the server. There are two main issues which we need to clarify. These regard the interaction between S-threads with the cache and the database.

§3. The cache

The cache of the server is maintained by a separate thread $\mathcal{C}$. It is the sole thread in the server that is allowed to search the cache. Let $\mathcal{T}$ be a S-thread that picked up a search request x. In order to search x in the cache $\mathcal{T}$ adds x to a data structure that $\mathcal{C}$ exposes. $\mathcal{T}$ then waits until $\mathcal{C}$ searches the cache and lets $\mathcal{T}$ know whether a reply to x was found in the cache or not.

On the part of $\mathcal{C}$, while the data structure is empty it waits until some request is added. Here you have the freedom of adding helper threads to $\mathcal{C}$ and treat $\mathcal{C}$ as a group of threads instead of just one thread. These can than help $\mathcal{C}$ track the data structure.

Above we mentioned that the cache maintains the top C search queries requested at least M times. In fact, our cache here will only strive to do so but will not fulfil this requirement at all times. Having the cache accurate means repetitive searching in the database files. This we must avoid.

To keep the cache relevant without halting the system and frequently searching the database files we will have the database consist of triples (x, y, z) where x is a search request, y is its reply, and z the number of times x was requested. When a S-thread probes the database in search of a reply it also searches for possible updates to the cache at the same time. If it finds any it "informs" $\mathcal{C}$ that updates are possible and $\mathcal{C}$ has to update the cache accordingly. Or may choose to ignore depending on your design.

To accomplish this you may endow $\mathcal{C}$ with its own helper threads that will wait for incoming update "messages" and consequently update the cache. However, updating the cache means it has to shut down. Hence, updating the cache too frequently is not considered a good idea.

We leave the details to you. You have to come up with an updating algorithm for the cache that will make it as accurate as possible while at

the same time not harming the performance of the system. You will have to defend your design during examination.

§4. The database

A S-thread $\mathcal{T}$ searching for a reply to x is not allowed to access the database files directly for reading or for writing. In what follows we explain how access to the database is granted to $\mathcal{T}$.

§4.1 Reading from the database. The server maintains an additional pool manager of *reader* threads. (You cannot use Java's built in pool manager API, you must construct this pool manager on your own). Only those are allowed to read the files of the database and return an answer to $\mathcal{T}$ whether a replay to x exists in the database or not. The server has Y reader threads where Y is a parameter passed to the server through the command line upon its creation.

A search thread $\mathcal{T}$ passes its query to the reader pool manager which assigns a reader thread to search for the query in the database. The thread $\mathcal{T}$ waits until a reply is returned to it.

Note that the reader threads must take part in the updating algorithm of the cache.

§4.2 Writing to the database. The server maintains an additional *writer* thread $\mathcal{W}$ which is the sole thread allowed to write to the database. This thread is also allowed to read the database of course in order to perform updates to the database.

The writer thread $\mathcal{W}$ awaits for the event that a S-thread $\mathcal{T}$ searching for a reply to some x failed to find a reply in both the cache and the database. Recall that in such a case $\mathcal{T}$ draws uniformly at random a reply y to x and seeks to write y in into the database.

As x was not found this means it is the first time it has been requested. Consequently $\mathcal{T}$ passes to $\mathcal{W}$ the "message" $(x, y, 1)$ to be written to the database. After passing this message to $\mathcal{W}$ thread $\mathcal{T}$ does not wait and immediately sends y through the socket to the requesting client and marks itself free for the next search request. It is the responsibility of $\mathcal{W}$ to process this message and write to the database.

§4.3 Organisation of the database files. The organisation of the files of the database is left to you. Clearly one can use a single file for the entire database. However, this would be a very poor design choice. As writing and reading from the database are two operations that must not occur concurrently.

Please design your database and its file structure carefully with the aim of maximising its performance. The implementation details are left to you. You will have to defend your choices upon examination.

§4.4 Updating the database. Let us consider some scenarios to guide you through your design. Suppose a S-thread $\mathcal{T}$ is looking for a reply to x and has found (x, y, z) in the database. This means that x has now been searched for the $z + 1$st time. How do you intend to update the database in this case?

Recall that writing to the database has to be done by $\mathcal{W}$ only. This means that all updates to the database also have to pass through $\mathcal{W}$. One option then is to request $\mathcal{W}$ to write $(x, y, z + 1)$ and overwrite (x, y, z). However, imagine a scenario where several clients are querying for x at the same time. If for each such reading of x from the database you will insist on updating the database this can result in a scheduling scenario in which search threads are being delayed by the update procedure of $\mathcal{W}$.

On the other hand if you postpone updates to the database and prefer to answer clients first this can result in having the cache becoming less accurate and consequently seeing a surge in the number of queries that must be sent to the database. This is because the cache does not reflect accurately the most frequently searched queries. What do you think Google would prefer to do in this case?

Design an algorithm for updating the database that will address such issues. You should also think about how does this algorithm for updating the databases effects your cache updating algorithm.

§5. What to submit?

We expect two executable files from you. One representing the client program and the other representing the server program. These programs will be called *client* and *server* respectively.

Each such program receives arguments at the command line upon their creation (that is your *args* array in the main method will not be empty). Here

is a summary of the arguments that the server and client program expect. If we missed something here feel free to let us know and complete it yourself as well.

§5.1 Server arguments.

1. S - number of allowed S-threads.
2. C - size of the cache.
3. M - the least number of times a query has to be requested in order to be allowed to enter the cache.
4. L - to specify the range $[1, L]$ from which missing replies will be drawn uniformly at random.
5. Y - number of reader threads.

§5.2 Client arguments.

1. R_1, R_2 - the integer numbers that specify the range $[R_1, R_2]$.
2. A file name containing $(p_x : x \in [R_1, R_2])$.

References

[1] Oracle. Lesson: All About Sockets. https://docs.oracle.com/javase/tutorial/networking/sockets/.