

Design Document

Dispatcher Simulator

Sagalpreet Singh

17th September, 2021



Introduction

1. What's it?
2. Logic Design



What's it?

"Dispatcher Simulator" is a project written in C which simulates the functioning of a dispatcher using a client-server network. The server responds to client requests under the following constraints:

- a) Server receives the requests on different threads, however there is a limit to the number of connections that a server can communicate on concurrently.
- b) Whenever the server receives a request, it enqueues the request into a request queue.
- c) Dispatcher, as its name suggests dequeues the requests and executes them. Dispatcher does so by assigning the requests to different threads. There is a predefined limit on the number of threads a dispatcher can use.
- d) On each of the threads that dispatcher assigns a task to, there is a predefined limit on number of open files and memory consumed. If execution of request exceeds this, the request is not processed further, and an error statement is output after which thread continues with its normal execution.

The request sent by client has to be of the following form:

```
<Dynamic Library Name (with absolute address)>  
<name of function to be invoked>  
<argument to the function>
```

For this project, the only supported functions are those with a single argument of type (double) and return type (double).

However, any library can be used.



File Structure:

- bin : contains executables
- obj : contains object files
- src : contains source code (c and header files)
- scripts : bash scripts for automated compilation, linking and execution of entire project (build and run)
- execution_results : contains output of all the requests handled by dispatcher (separate file for each thread)
- test_data : contains file containing request data for the request made by client
- README.txt
- design.pdf

Logic Design

Different modules have been used to build this project. Following are their implementation details.

a) client

- This program contains the implementation of the client. The client sends the request to the server and communicates over the network. Client uses the `sys/socket.h` and `arpa/inet.h` apis.
- It is a standalone piece of code.
- PORT over which the client has to send requests is taken as a command line argument. The client attempts to build a socket and connect to the server over the port. Proper checks are made if the client is able to connect to the server else error messages are logged onto the console.
- An optional second command line argument can be used to specify the name client which is "client" by default.
- The program takes input from stdin which specifies the request that has to be communicated over to the server.

b) server

- This program contains the implementation of a server. The server receives the request over the network sent by clients and sends a confirmation as to whether the request has been queued or not. Note that it may so happen that the request is queued but not executed later as it may exceed the resource limit of the thread it is assigned to or it may be an illegal request. In any case, an attempt to execute it is made.
- It includes functions spread across various other files but has a main function.
- The main function takes in input 6 command line arguments: PORT, queue_size, max_connections, max_threads, max_files, max_size (in order)
- This program spawn two threads
- One of the thread functions sets up the server and ultimately accepts requests from clients and enqueues them or rejects them.
- The other thread function is a dispatcher
- Following are the 3 functions of this program:

- *runDispatcher*: It spawns the max_threads(the command line argument) number of threads each of which check for the requests is pending in the queue and service them.
- *runServer*: It attempts to make a socket, connect it to the port and then listens to the client requests. It listens to client requests concurrently on different threads. However the limit to the number of concurrent connections is set while the listen function is called. It uses an accept function in an infinite loop. accept is a blocking call. Whenever, a new connection request from client is made, a check is made if it is successful and then the request using its socket descriptor is assigned to thread function - handleConnections
- *handleConnections*: This is a thread function which reads the request from the client and attempts to queue the request if the queue isn't already filled. In case the queue is already full, the request is rejected and 0 is sent to the client over connection. On successful enqueue, 1 is sent to the client. In either case, after communicating the response to the client, the socket connection is closed.
- The server program uses various functions from other modules which are discussed below. These include the implementation of request queue, functions for deep copy and freeing memory pointed to by pointers, request listener which reads the request sent by client stores it as a message (message is the name of structure used to store requests) and most importantly the dispatcher.

c) request_listener

- This module contains a single function - request_listener.
- The client sends the request in the form of a string (as it is read from stdin in the client program). The string is read as a stream in request_listener, from which dynamic library name, function name and arguments are read and stored in message structure. The stream is closed and the message struct is returned by the request_listener function.

d) memory

- This module contains a helper function used across different modules. The functions help in memory allocation and deallocation.
- The functions are duplicate_sptr, duplicate_dptr, free_sptr, free_dptr
- duplicate_sptr takes a (char*) as input. It allocates equal memory on heap and deep copies contents from argument pointer's memory to newly allocated memory. This is equivalent to deep copying a string.


- Similarly, `duplicate_dptr` takes `(char**)` as input which can be seen as an array of strings. Here, the depth of copy is one more than `duplicate_sptr`. The function copies all the strings into newly allocated memory.
- `free_sptr` frees the memory pointed to by a `(char *)` pointer.
- `free_dptr` frees the memory pointed to by `(char **)` pointer. `char**` indicates an array of strings, so `free_sptr` is called for each of the strings before freeing the memory pointed to by `char*` pointer.

e) request_queue

- This is the implementation of queue (linked-list based), which contains message structure as its contents. As already mentioned, message structure is used to store requests (see last point in server(b))
- Queue has a bound on its maximum size. The queue allows enqueue and dequeue. In addition to this it has a function which can be used to initialize the queue on heap and return pointer to allocated memory.

f) dispatcher

- dispatcher module holds the functions to handle the core functionality of a dispatcher (dispatch function). The helper functions are `execute_command` and `itoa`. `itoa` converts an integer to string. (not all compilers support `itoa`, so I implemented my own function). `execute_command` takes a shell command in the form of string, executes it and returns the value (supports only integer outputs).
- dispatch function is essentially a thread function which was invoked by the server's `runDispatcher` routine. For each thread, a file is created in the `/execution_results` directory. This directory has files with name as `thread_id` and stores the result of functions that were run on that thread by dispatcher. If some function couldn't be run, the error is written to that file.
- The function dequeues requests from the request queue and executes them. It loads dynamic libraries and calls functions. The function also keeps track of memory used and files open through system calls and breaks the function call if resource usage exceeds the permissible limit.



** Since the queue is common, anytime a dequeue or enqueue is performed on it, a mutex lock is put over it to prevent undesirable results because of concurrent operations in multithreading.*