University of Minho

Integrated Masters in Software Engineering

Operating Systems

Practical Assignment

Filipe Felício
(A85983)

Gonçalo Rodrigues
(A90439)

Miguel Brandão
(A82349)

17th June, 2021

Contents

List of Figures

1 Introduction

The objective of this report is to explain our choices/solutions to the problems given in this assignment, within the scope of this course. For this task we had to implement a Client/Server Architecture capable of handling multiple concurrent requests to apply a filter or a sequence of filters input audios and store the output according to the user's specifications.

2 Usage

2.1 Client

The client is the program that accepts the user input and sends it to the server to be processed.

The client has two functionalities: `status` and `transform`.

On the server-side we carefully made sure that any incorrect input - that did not follow the desired format - does not go to the server, instead we analyzed in which way it is faulty and sent the client an appropriate error message. All cases where consider:

- A command is neither a `transform` nor a `status`.
- A `status` command has extra arguments.
- A `transform` command does not have an input, output and at least one filter.

Upon the absentee of arguments the client gets a helper explaining the application interface.

2.1.1 status

This task is run using

```
$ bin/aurras status
```

and it outputs the current state of the server: which tasks are being run and the current resource allocation:

```
$ bin/aurras status
task #1: transform samples/Ievan Polkka (Loituma).m4a
    ↪ tmp/output.mp3 rapido rapido
filter alto: 0/2 (running/max)
filter baixo: 0/2 (running/max)
filter eco: 0/1 (running/max)
filter rapido: 2/2 (running/max)
```

```
filter lento: 0/1 (running/max)
pid: 109504
```

2.1.2 `transform`

This task is run using

```
$ bin/aurras transform PATH_TO_INPUT_FILE
   ↪ PATH_TO_OUTPUT_FILE FILTER ...
```

and outputs the result of running the input file through one or multiple filters. It also outputs the state of the task to the console:

```
pending
processing
processed
```

2.2 Server

The Server is capable of receiving incoming requests, process them and send notification to the user.

The server must be run before any client. It is executed with

```
$ bin/aurrasd etc/aurrasd.conf bin/aurrasd-filters/
```

and runs until manually closed. It ends gracefully upon receiving a `SIGINT` (mostly by means of a `CTRL+C`). On launch it needs two arguments:

- a configuration file from where it receives all of the information/restrictions on filters, such as:
  1. filter name,
  2. executable name,
  3. maximum number of filters that can be allocated by processes at the same time.
- a filters folder from where (alongside the executable name) the server can get the filters to apply to the audios.

The process of interpreting both these arguments must be completely abstract so a server instance can work with any (valid) input.

On the server-side there was also a careful verification of the requests sent by users (further we will review the protocol of this structure in depth), in the case of a transform request, for instance, if the combined number of any filter is bigger than the maximum instances of that particular filter (as

specified by the configuration file) there is no point in processing this request - because the server will never in its life circle have enough resources to fulfill this request - so the sophisticated approach when dealing with such request is to send back an appropriate error message to the user. Also if any of the given filter names does not match with a configuration file given filter this request is also denied and a comprehensive error message with this information is given to the user.

2.2.1 Configuration File

Like mention in the previous section, a configuration file rules all of the aspects of the server's life cycle such as: name of the filters, executable name and the maximum number of instances per filter. The correct approach when dealing with such specification was to load the configuration file information into a structure. For reference this is how a configuration file might look like:

```
alto aurrasd-gain-double 2
baixo aurrasd-gain-half 2
eco aurrasd-echo 1
rapido aurrasd-tempo-double 2
lento aurrasd-tempo-half 1
```

As we can see, for each line there is the information that we discussed prior concerning one filter. What he did was storing this information in a Filter structure with each of this fields and also an additional with the number of running instances for the filter, this allow us to have a one way spot where we could manage the filters resources. Moreover all the filters were stored in an Filter Array global variable so we could always have access to this crucial information from every line of code in the Server. This structure is not optimal (like a lot of the structures with the work) since it has O(N) complexity when searching and O(1) when inserting. A better structure would be something with Hashing like a Hash Table, but this concerns were put to second plain since they were not in the scope of this work/course. A further explanation on this matter will be on the Conclusion.

3 Architecture

3.1 Client-to-server communication

When executed, the server makes a named pipe for clients-to-server communication. Because of the request protocol, a single pipe can support communications for all clients.

To enable server-to-client communication, the client makes a named pipe named after it's own PID.

The client then sends to the server a data structure containing its PID (so later the server can track the request's owner and send the appropriate message), an array with the arguments it was given on execution and the number of those arguments also. Afterwards, it closes its end of the pipe and starts reading (blocking) from the server-to-client pipe it made earlier.

In the case of a transform request, after reading the request structure sent by the client, the server opens the pipe named after the client's PID and writes the message 'pending', which the client reads and prints to its `stdout`, return to it's previous blocking read state, this starts the processing phase of a filter applying request that we will explain in detail later.

3.2 Request parsing

After getting the request and informing the client of it's pending status, the server begins to work on the request.

It starts by extracting the filter's names from the request. Then, it verifies them to make sure they are valid (according to the server configuration file): if they aren't, it sends a message back to client goes back to reading from to client-to-server pipe, like we explain prior.

However, if they are valid, it proceeds to merge them; in other words, it creates s new structure which contains a tuples of a name of a filter and the number of times it is used. This structure is crucial to process the requests in a abstract faction, since we can engage in multiple verification and resource allocation and deallocate for both single and multiple filter application requests.

So, in accordance with the logistic mention aboved, this list of tuples is then used to verify if it is possible to process the request: if in a request a filter is used more times than the server config allows, that request is deemed impossible, dropped, and the client notified (like we mention above, as well).

3.3 Process

If the request passes all former verification, it becomes represented in a structure `process`. This structure holds all data about a request that will be processed: the client PID (used to track the named from which the server-to-client communication will happen), the number of filter that it will require (helpful when checking the server availability towards a process and quickly

allocate and deallocate resources, always in accordance with the configuration file), its input and output files and its tuple list. This structure is then placed in a queue and waits for processing.

3.4 Queue

The queue is a multidimensional array. We think of it as a matrix with 16 rows and 1024 columns. A process is stored in it according to how many filters it uses. A process with *n* filters will be appended into row *n-1*;if it has 16 filters or more, it will be stored in row 15. The reason why this method was chosen will be discussed in the next section.

3.5 Processing

After parsing the request and putting its `process` representation in a queue, the server starts processing said queue.

The server has a counter that stores the number of available *filter slots*, that is, filter slots are not reserved by another process.

Since it knows the maximum number of filter slots it can allocate at that time, say, *n*, then it only needs to process the queue from rows *0* to *n-1*. This way, we don't need to traverse rest of the queue and validate those processes, since we know from the start that there will never be enough available filter slots.

Having established the range for which to traverse the queue, the server starts traversing at row 0 and validating and processing each process. It does this until the queue is empty or there are no more available filters. When the queue is empty, the server waits on a blocking read (from the client-to-server pipe) until another client sends another request.

For each of those processes, the following happens:

1. it checks the filter availability: the server checks if all the filters the process will need are available (if not, the process remains in the queue);
2. the filters the process needs are reserved;
3. a message is sent to the client notifying it that the process will begin processing;
4. the process execution is started.

3.5.1 Execute single-filter process

The logistics for this execution were quite simple. As we only had the need to perform one filter application, we only had to merely open both the input

and output files, duplicate the `STD_IN` and the `STD_OUT` to point to their file descriptors, respectively, close them and them run the filter using the concatenation of the folder given in the server execution and the filter binary given by the configuration file and obtained from its named that was used by the user to invoke it.

3.5.2 Execute multiple-filter process

We things were a bit more tricky. To do multiple filter application we needed to run each execution in its own process and communicate between filters using anonymous pipes to create a streaming pipeline to optimally perform this task. Let say we need to perform N filters, that means that we have to use N-1 pipes, where we need to only code 3 different situations:

1. **the first filter** reads from the input file (using open and file descriptor duplication like above) and writes to the input end of the first pipe (using open and file descriptor duplication like above)
2. **the last filter** reads from the writing end of the last pipe and writes to the output file (using open and file descriptor duplication like above).
3. **all the others** in between, if needed (3 or more filters), read from the previous pipe and write to the next

This was done with a pipe matrix using a for loop from 0 to N -1 filters and mind that the not used pipes ends in each step needed to be closed before any filter execution (using execl likewise).

3.6 Signals and response

The `SIGUSR1` is captured by the server and handled. The handling consists of a `wait`, that is used to recoup the PID of the fork that ended. With this PID, we can find the process that ended, free its resources (the filter slots) and send a message to the client that issued the request saying the the process is complete.

Since the server now has more resources, we trigger another queue processing round.

4 Scripts

To aid with testing and development, we developed some Python scripts to help with running multiple clients simultaneously and to provide some benchmarks.

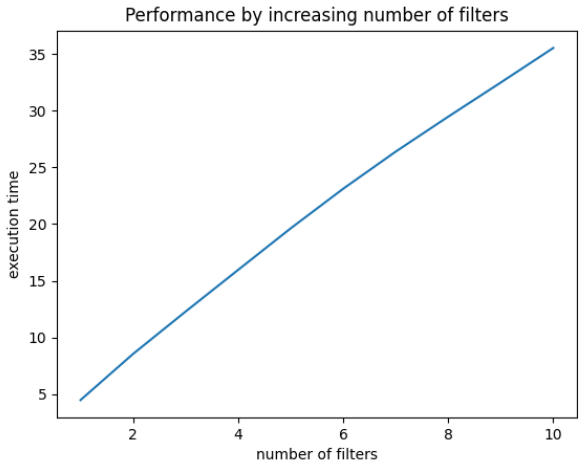

Figure 1: An example of the output of the 'performance by number of filters' script

4.1 Performance by increasing number of filters

This script runs several consecutive client instances with an increasing number of filters per request, in order to evaluate how performance is affected by the number of filters of a request. See figure 1 for an example of this script's output.

4.2 Server

This script launches a server after making some preparations, like deleting artifacts of previous testing and recompiling the server.

There are some variants of this script, like `unavailable_server`, which does the same as above, but launches a server with no available filter slots, and `unlimited_server`, which does the same as the previous one, but with a very large amount of filter slots.

5 Conclusion

This work is far from perfect, as it is out the scope of the class, we ignored the use of dynamic memory in our structures throughout the project. This means there will always be some hard limit to what the server and clients can do, but we feel it is a worthwhile trade-off for this case, as it significantly lowers complexity and code size and allowed us to focus on the essential problems that we felt were more important to this course. There are also, for sure, many other optimizations that could be made in terms of memory and

performance, such as implementing some of our structures as hash maps or tables, binary trees, min-heaps (for choosing processes efficiently on a number of filters logistic) or linked lists, instead of regular arrays.

We are proud of our solution and believe that it is adequate and certainly excited to this discuss it. We believe we accomplished most if not all of the goals that were proposed and all the programs do what they are supposed to do. We are also particularly pleased with our test and benchmarking suite, which, although a bit out of scope, was a great learning experience and a valuable tool when testing our implementation to its limits and find errors.