

Task Orchestrator

Generated by Doxygen 1.9.1

Chapter 1

Task Orchestrator Service

1.1 Introduction

This task orchestrator service is implemented in C within the Operating Systems environment for the 2023/2024 academic year.

- [Description and Requirements](#)
- [Report](#)

It is a service that allows the asynchronous execution of tasks, composed of two main components:

- **Server:** Responsible for managing the tasks execution or scheduling and transmitting messages to clients.
- **Client:** Has the ability to execute tasks and check their status.

1.2 Scheduling Policies

The service supports the following scheduling policies:

- **FCFS:** First Come First Served
- **SJF:** Shortest Job First
- **PES:** Priority Escalation Scheduling

1.3 Compilation

To compile the project, run the following command:
`make`

The `orchestrator` and `client` executables will be generated in the `bin` directory.

1.4 Usage

1.4.1 Orchestrator Server Usage

Usage: bin/orchestrator <output_dir> <parallel_tasks> [sched_policy]

Options:

<output_dir>	Directory to store task output and history files
<parallel_tasks>	Maximum number of tasks running in parallel
[sched_policy]	Scheduling policy (FCFS, SJF, PES) default: SJF

1.4.2 Client Usage

Usage: bin/client <option [args]>

Options:

execute <est_time priority> <-u -p> "<command [args]>"	Execute a command
est_time	Estimated time of execution, in case scheduling policy is SJF
priority	Priority of the task, in case scheduling policy is PES
-u	Unpipid command
-p	Piped command
command	Command to execute. Maximum size: 300 bytes
status	Status of the tasks
kill	Terminate the server

Note: The estimated time or priority is irrelevant if the scheduling policy is FCFS, but it must be provided as an argument.

1.4.2.1 Execute Task

bin/client execute <est_time|priority> <-u|-p> "<command [args]>"

1.4.2.2 Check Tasks Status

bin/client status

This command will display all the executing, scheduled, and completed tasks. For completed tasks the elapsed time is also displayed.

1.4.2.3 Kill Orchestrator Server

bin/client kill

This command will send a request to the orchestrator server to shutdown.

This terminates all the scheduled tasks and the server itself.

Executing tasks will still be able to finish, and the respective files will be saved in the output directory, as usual.

1.5 Output Files

The output directory has the given structure:

```
<output_dir>
  history
  task_number
  task1
    out
    err
    time
  task2
    out
    err
    time
  ...
  taskN
    out
    err
    time
```

1.6 Testing

You can find a selection of test cases in the `tests` directory.

This tests require the `orchestrator` to be running.

Run then from the project root directory, so the scripts can find the client executable.

1.7 Authors

- Flávia Araújo - [@flaviaraujo](#)
- Miguel Carvalho - [@migueltc13](#)

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

request	The request struct	??
-------------------------	------------------------------	----

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

include/ client.h	??
include/ command.h	??
include/ orchestrator.h	??
include/ request.h	??
include/ task_nr.h	??
src/ client.c	??
src/ command.c	??
src/ orchestrator.c	??
src/ request.c	??
src/ task_nr.c	??

Chapter 4

Class Documentation

4.1 request Struct Reference

The request struct.

Public Attributes

- int [type](#)
- int [est_time](#)
- char [command](#) [[MAX_CMD_SIZE](#)]
- bool [is_piped](#)
- unsigned int [task_nr](#)
- char [client_fifo](#) [[CLIENT_FIFO_SIZE](#)]

4.1.1 Detailed Description

The request struct.

The request struct represents a request sent by the client, or a request sent by the executor to the [orchestrator.c](#), marking commands as completed.

4.1.2 Member Data Documentation

4.1.2.1 type

```
request::type
```

The type of the request.

4.1.2.2 `est_time`

`request::est_time`

The estimated time or the priority of the request, based on the policy being used in the [orchestrator.c](#) server.

4.1.2.3 `command`

`request::command`

The command to be executed or marked as completed.

4.1.2.4 `is_piped`

`request::is_piped`

A boolean indicating if the command is piped.

4.1.2.5 `task_nr`

`request::task_nr`

The task number of the request.

4.1.2.6 `client_fifo`

`request::client_fifo`

The client FIFO name to send the response to the client.

The documentation for this struct was generated from the following file:

- [src/request.c](#)

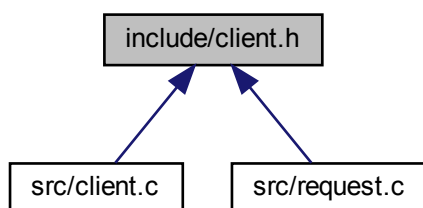
Chapter 5

File Documentation

5.1 doc/mainpage.dox File Reference

5.2 include/client.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- `#define CLIENT_FIFO_SIZE 16`
Maximum size of the client fifo name.

5.2.1 Macro Definition Documentation

5.2.1.1 CLIENT_FIFO_SIZE

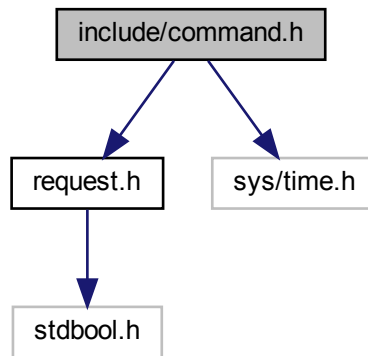
```
#define CLIENT_FIFO_SIZE 16
```

Maximum size of the client fifo name.

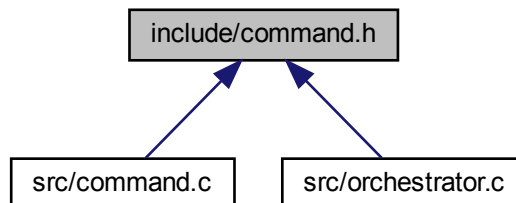
5.3 include/command.h File Reference

```
#include "request.h"
#include <sys/time.h>
```

Include dependency graph for command.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define HISTORY_NAME "history"`
Name of the history file.

Functions

- `int exec (Request *r, char *output_dir, struct timeval start_time)`
Executes a command, single or piped, using process forking, it also handles the redirection of stdout and stderr to respective task files, as well as the writing of the execution time to a file, and the writing of the task number, command and execution time to the history file.

5.3.1 Macro Definition Documentation

5.3.1.1 HISTORY_NAME

```
#define HISTORY_NAME "history"
```

Name of the history file.

5.3.2 Function Documentation

5.3.2.1 exec()

```
int exec (
    Request * r,
    char * output_dir,
    struct timeval start_time )
```

Executes a command, single or piped, using process forking, it also handles the redirection of stdout and stderr to respective task files, as well as the writing of the execution time to a file, and the writing of the task number, command and execution time to the history file.

This is the main function of the [command.c](#) module, used in [orchestrator.c](#) server to execute a command.

First, all the necessary files are created, namely the output directory for the task files, such as the output file, the error file and the time file. Also opens the history file to write the task number, the command and the execution time.

This file's names are defined in the macros [OUTPUT_NAME](#), [ERROR_NAME](#), [TIME_NAME](#) and [HISTORY_NAME](#).

It uses the fork system call to create a child process that will execute the command. The parent process will wait for the child process to finish. This fork call is inside another fork call so the parent process can wait for the child process to finish and send the request to the orchestrator as completed. This way, the server can continue to receive and handle new requests while the child process is executing the command.

The inner parent process, after waiting for the child process to finish executing the command, writes the total time since the request was received and the command finished executing to the time file and the history file.

It uses the [parse_cmd](#) function to parse the command into an array of arguments. It also uses the [parse_cmd_pipes](#) function in case the command is piped.

After the command is parsed, it is executed using the `execvp` system call. When `execvp` fails (e.g returns -1) it writes an error message to stderr using the [write_error](#) function.

Parameters

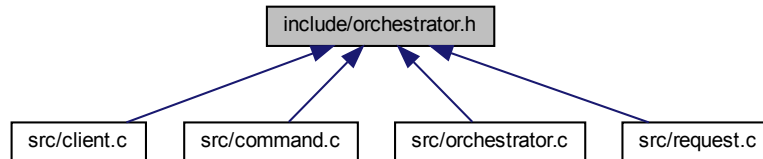
<i>r</i>	The request containing the command to execute, if the command is piped and the task number
<i>output_dir</i>	The output directory to create the task directory and respective task result files
<i>start_time</i>	The time when the execute request was received to calculate the total time since the request was received and the command finished executing

Returns

0 if successful

5.4 include/orchestrator.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- `#define SERVER_FIFO "server_fifo"`
Name of the server fifo.
- `#define FCFS 0`
First Come First Serve.
- `#define SJF 1`
Shortest Job First.
- `#define PES 2`
Priority Escalation Scheduling.
- `#define EXEC_TIME_STRING_SIZE 16`
Maximum size of the execution time string.
- `#define EXECUTE_MSG_SIZE (24 + TASK_NR_STRING_SIZE)`
Maximum size of the execute message transmitted between the orchestrator and the client.

5.4.1 Macro Definition Documentation

5.4.1.1 SERVER_FIFO

```
#define SERVER_FIFO "server_fifo"
```

Name of the server fifo.

5.4.1.2 FCFS

```
#define FCFS 0
```

First Come First Serve.

5.4.1.3 SJF

```
#define SJF 1
```

Shortest Job First.

5.4.1.4 PES

```
#define PES 2
```

Priority Escalation Scheduling.

5.4.1.5 EXEC_TIME_STRING_SIZE

```
#define EXEC_TIME_STRING_SIZE 16
```

Maximum size of the execution time string.

5.4.1.6 EXECUTE_MSG_SIZE

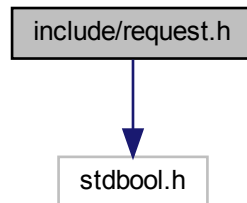
```
#define EXECUTE_MSG_SIZE (24 + TASK_NR_STRING_SIZE)
```

Maximum size of the execute message transmitted between the orchestrator and the client.

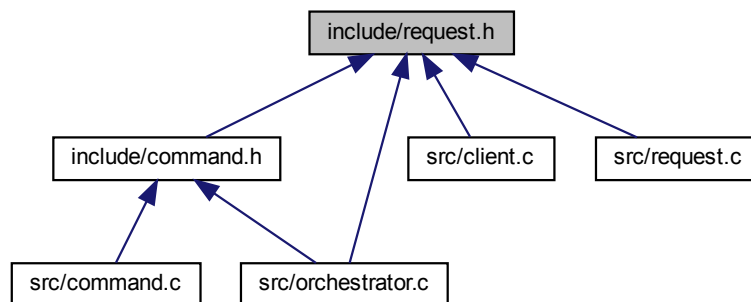
5.5 include/request.h File Reference

```
#include <stdbool.h>
```

Include dependency graph for request.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define **EXECUTE** 0
Execute request code.
- #define **STATUS** 1
Status request code.
- #define **COMPLETED** 2
Completed request code.
- #define **KILL** 3
Kill request code.
- #define **MAX_CMD_SIZE** 300
Maximum size of the command string.

Typedefs

- typedef struct [request](#) [Request](#)

Functions

- [Request](#) * [create_request](#) (int type, int est_time, char *command, bool is_pipe, char *client_fifo)
Creates a new request.
- int [get_type](#) ([Request](#) *r)
Get the type of the request.
- int [get_est_time](#) ([Request](#) *r)
Get the estimated time or priority of the request.
- char * [get_command](#) ([Request](#) *r)
Get the command to be executed.
- bool [get_is_piped](#) ([Request](#) *r)
Get the boolean indicating if the command is piped.
- unsigned int [get_task_nr](#) ([Request](#) *r)
Get the task number of the request.
- char * [get_client_fifo](#) ([Request](#) *r)
Get the client FIFO name to send the response to the client.
- long [get_time](#) ([Request](#) *r)
- void [set_type](#) ([Request](#) *r, int type)
Set the type of the request.
- void [set_task_nr](#) ([Request](#) *r, unsigned int task_nr)
Set the task number of the request.
- void [set_client_fifo](#) ([Request](#) *r, char *client_fifo)
Set the client FIFO name to send the response to the client.
- void [print_request](#) ([Request](#) *r, int policy)
Print the request.
- char * [type_to_string](#) (int type)
Convert the type of the request to a string.
- unsigned long [sizeof_request](#) ()
Get the size of the request.
- [Request](#) * [clone_request](#) ([Request](#) *r)
Clone a request.

5.5.1 Macro Definition Documentation

5.5.1.1 EXECUTE

```
#define EXECUTE 0
```

Execute request code.

5.5.1.2 STATUS

```
#define STATUS 1
```

Status request code.

5.5.1.3 COMPLETED

```
#define COMPLETED 2
```

Completed request code.

5.5.1.4 KILL

```
#define KILL 3
```

Kill request code.

5.5.1.5 MAX_CMD_SIZE

```
#define MAX_CMD_SIZE 300
```

Maximum size of the command string.

5.5.2 Typedef Documentation

5.5.2.1 Request

```
typedef struct request Request
```

5.5.3 Function Documentation

5.5.3.1 create_request()

```
Request* create_request (
    int type,
    int est_time,
    char * command,
    bool is_piped,
    char * client_fifo )
```

Creates a new request.

Allocates memory for a new request and initializes it with the given parameters.

It initializes the task number with 0, which is not a valid task number.

For the creation of a status request, the

Parameters

<i>est_time</i>	should be 0, as it is not used, as well as the
<i>command</i>	should be NULL and
<i>is_piped</i>	should be false.

Example of the creation of an execute request:

```
Request *execute_request = create_request(EXECUTE, 12, "ls -l | cat", true, "client-1234");
```

Example of the creation of a status request:

```
Request *status_request = create_request(STATUS, 0, NULL, false, "client-1234");
```

Example of the creation of a kill request:

```
Request *kill_request = create_request(KILL, 0, NULL, false, "");
```

The caller is responsible for freeing the returned request by this function.

Parameters

<i>type</i>	the type of the request
<i>est_time</i>	the estimated time or priority of the request
<i>command</i>	the command to be executed
<i>is_piped</i>	a boolean indicating if the command is piped
<i>client_fifo</i>	the client FIFO name to send the response to the client

5.5.3.2 get_type()

```
int get_type (
    Request * r )
```

Get the type of the request.

Parameters

<i>r</i>	The request to get the type from
----------	----------------------------------

Returns

The type of the request

5.5.3.3 get_est_time()

```
int get_est_time (
    Request * r )
```

Get the estimated time or priority of the request.

Parameters

<i>r</i>	The request to get the estimated time or priority from
----------	--

Returns

The estimated time or priority of the request

5.5.3.4 get_command()

```
char* get_command (  
    Request * r )
```

Get the command to be executed.

Parameters

<i>r</i>	The request to get the command from
----------	-------------------------------------

Returns

The command to be executed

5.5.3.5 get_is_piped()

```
bool get_is_piped (  
    Request * r )
```

Get the boolean indicating if the command is piped.

Parameters

<i>r</i>	The request to get the is_piped from
----------	--------------------------------------

Returns

The boolean indicating if the command is piped

5.5.3.6 get_task_nr()

```
unsigned int get_task_nr (  
    Request * r )
```

Get the task number of the request.

Parameters

<i>r</i>	The request to get the task number from
----------	---

Returns

The task number of the request

5.5.3.7 get_client_fifo()

```
char* get_client_fifo (  
    Request * r )
```

Get the client FIFO name to send the response to the client.

Parameters

<i>r</i>	The request to get the client FIFO name from
----------	--

Returns

The client FIFO name to send the response to the client

5.5.3.8 get_time()

```
long get_time (  
    Request * r )
```

5.5.3.9 set_type()

```
void set_type (  
    Request * r,  
    int type )
```

Set the type of the request.

Parameters

<i>r</i>	The request to set the type
<i>type</i>	The type to set

5.5.3.10 set_task_nr()

```
void set_task_nr (
    Request * r,
    unsigned int task_nr )
```

Set the task number of the request.

Parameters

<i>r</i>	The request to set the task number
<i>task_nr</i>	The task number to set

5.5.3.11 set_client_fifo()

```
void set_client_fifo (
    Request * r,
    char * client_fifo )
```

Set the client FIFO name to send the response to the client.

Parameters

<i>r</i>	The request to set the client FIFO name
<i>client_fifo</i>	The client FIFO name to set

5.5.3.12 print_request()

```
void print_request (
    Request * r,
    int policy )
```

Print the request.

Prints the request attributes to the standard output.

Based on the policy being used, it prints the estimated time or the priority.

If the policy is FCFS, it doesn't print either the estimated time or the priority.

Parameters

<i>r</i>	The request to print
<i>policy</i>	The policy being used

5.5.3.13 type_to_string()

```
char* type_to_string (
    int type )
```

Convert the type of the request to a string.

Possible types are [EXECUTE](#), [STATUS](#), [COMPLETED](#) and [KILL](#).

Parameters

<i>type</i>	The type of the request
-------------	-------------------------

Returns

The string representation of the type of the request

5.5.3.14 sizeof_request()

```
unsigned long sizeof_request ( )
```

Get the size of the request.

Used due to the fact that the [Request](#) struct is not visible to other modules, so `sizeof(Request)`

wouldn't give the correct size of the struct.

Returns

The size of the Request struct

5.5.3.15 clone_request()

```
Request* clone_request (
    Request * r )
```

Clone a request.

Allocates memory for a new request and initializes it with the attributes of the given request.

Used in the [orchestrator.c](#) to clone a request before adding it to the executing or scheduled array.

The caller is responsible for freeing the returned request by this function.

Parameters

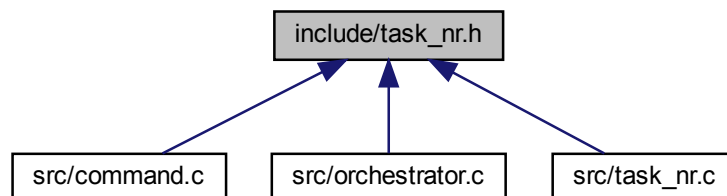
<i>r</i>	The request to clone
----------	----------------------

Returns

The cloned request

5.6 include/task_nr.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- `#define TASK_NR_STRING_SIZE 8`
Maximum size of the task number string.

Functions

- unsigned int `load_task_nr` (char *output_dir)
Load the task number from the file in the output directory.
- unsigned int `save_task_nr` (unsigned int task_nr, char *output_dir)
Save the task number to the file in the output directory.

5.6.1 Macro Definition Documentation

5.6.1.1 TASK_NR_STRING_SIZE

```
#define TASK_NR_STRING_SIZE 8
```

Maximum size of the task number string.

5.6.2 Function Documentation

5.6.2.1 `load_task_nr()`

```
unsigned int load_task_nr (
    char * output_dir )
```

Load the task number from the file in the output directory.

If the file does not exist, create it with the initial value 1.

Used by the [orchestrator.c](#) to load the task number when starting.

Parameters

<code>output_dir</code>	The output directory to load the task number file
-------------------------	---

Returns

The task number, or 0 if an error occurs

5.6.2.2 `save_task_nr()`

```
unsigned int save_task_nr (
    unsigned int task_nr,
    char * output_dir )
```

Save the task number to the file in the output directory.

Used by the [orchestrator.c](#) to save the task number before exiting.

Parameters

<code>task_nr</code>	The task number to save
<code>output_dir</code>	The output directory to write the task number file

Returns

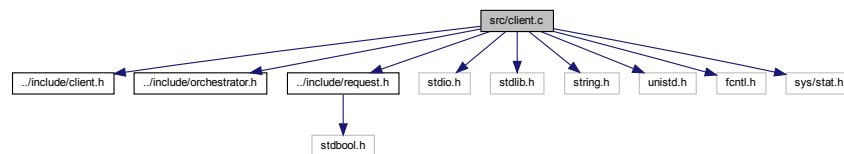
The task number, or 0 if an error occurs

5.7 `src/client.c` File Reference

```
#include "../include/client.h"
#include "../include/orchestrator.h"
```

```
#include "../include/request.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
```

Include dependency graph for client.c:



Macros

- `#define BUF_SIZE 4096`
The size of the buffer to read from the FIFO.

Functions

- `void client_usage (char *name)`
Print usage for the client program.
- `int main (int argc, char **argv)`
The main function of the client program.

5.7.1 Macro Definition Documentation

5.7.1.1 BUF_SIZE

```
#define BUF_SIZE 4096
```

The size of the buffer to read from the FIFO.

5.7.2 Function Documentation

5.7.2.1 client_usage()

```
void client_usage (
    char * name )
```

Print usage for the client program.

Parameters

<i>name</i>	the name of the client executable
-------------	-----------------------------------

5.7.2.2 main()

```
int main (
    int argc,
    char ** argv )
```

The main function of the client program.

The client program is responsible for sending requests to the server.

The client program can execute a command, check the status of the executing, scheduled and completed requests, or kill the server.

It's responsible for parsing the arguments and sending the requests to the server program via the server FIFO.

The status and execute options require the creation of a client FIFO to receive the server response.

Parameters

<i>argc</i>	the number of arguments
<i>argv</i>	the arguments

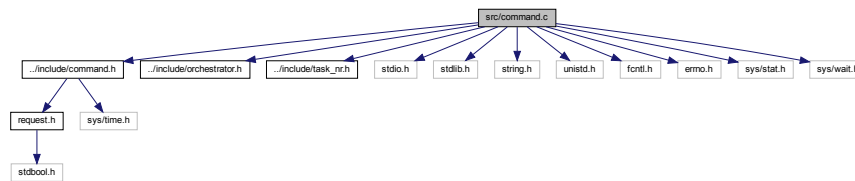
Returns

0 if the program executed successfully, 1 otherwise

5.8 src/command.c File Reference

```
#include "../include/command.h"
#include "../include/orchestrator.h"
#include "../include/task_nr.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/wait.h>
```

Include dependency graph for command.c:



Macros

- `#define MAX_ARGS MAX_CMD_SIZE`
The max number of arguments in a command.
- `#define ERROR_MSG_SIZE MAX_CMD_SIZE + 100`
The max size of the error message.
- `#define TASK_PREFIX_NAME "task"`
The prefix name for the task directories.
- `#define OUTPUT_NAME "out"`
The name of the output file.
- `#define ERROR_NAME "err"`
The name of the error file.
- `#define TIME_NAME "time"`
The name of the time file.

Functions

- `char ** parse_cmd_pipes (char *cmd, int *N)`
Parse a piped command into an array of single commands.
- `char ** parse_cmd (char *cmd)`
Parse a command into an array of arguments.
- `void write_error (char *cmd_name)`
Write an error message to stderr.
- `int exec (Request *r, char *output_dir, struct timeval start_time)`
Executes a command, single or piped, using process forking, it also handles the redirection of stdout and stderr to respective task files, as well as the writing of the execution time to a file, and the writing of the task number, command and execution time to the history file.

5.8.1 Macro Definition Documentation

5.8.1.1 MAX_ARGS

```
#define MAX_ARGS MAX_CMD_SIZE
```

The max number of arguments in a command.

5.8.1.2 ERROR_MSG_SIZE

```
#define ERROR_MSG_SIZE MAX_CMD_SIZE + 100
```

The max size of the error message.

5.8.1.3 TASK_PREFIX_NAME

```
#define TASK_PREFIX_NAME "task"
```

The prefix name for the task directories.

5.8.1.4 OUTPUT_NAME

```
#define OUTPUT_NAME "out"
```

The name of the output file.

5.8.1.5 ERROR_NAME

```
#define ERROR_NAME "err"
```

The name of the error file.

5.8.1.6 TIME_NAME

```
#define TIME_NAME "time"
```

The name of the time file.

5.8.2 Function Documentation

5.8.2.1 parse_cmd_pipes()

```
char ** parse_cmd_pipes (  
    char * cmd,  
    int * N )
```

Parse a piped command into an array of single commands.

Used in the function [exec](#) to parse a piped command into an array of single commands.

It also strips leading and trailing whitespaces from each command, see the sencond and third example below.

It also can handle quoted strings, as demonstrated in the third example.

It can handle new lines as it's possible to have a command with multiple lines, see the fourth example.

Examples:

```
- "ls -l | cat" -> ["ls -l", "cat"]  
- "  ls -l | cat  " -> ["ls -l", "cat"]  
- "echo \"Hello World\" | cat" -> ["echo \"Hello World\"", "cat"]  
- "echo \"Hello\\nWorld\" | cat\\n\\n" -> ["echo \"HelloWorld\"", "cat"]
```

Note: Uncomment the last for loop in this function to print the resulting array of single commands.

Parameters

<i>cmd</i>	The piped command to parse
<i>N</i>	The number of commands in the array

Returns

An array of strings, or NULL if an error occurred

5.8.2.2 parse_cmd()

```
char ** parse_cmd (
    char * cmd )
```

Parse a command into an array of arguments.

Used in the function [exec](#) to parse a command into an array of arguments.

It also strips leading and trailing whitespaces from each argument, see the sencond and third example below.

It also can handle quoted strings, as demonstrated in the third example.

Examples:

```
- "ls -l -a" -> ["ls", "-l", "-a", NULL]
- "  ls -l    -a" -> ["ls", "-l", "-a", NULL]
- "echo \"Hello   World\"" -> ["echo", "Hello   World", NULL]
```

Parameters

<i>cmd</i>	The command to parse
------------	----------------------

Returns

An array of arguments, or NULL if an error occurred

5.8.2.3 write_error()

```
void write_error (
    char * cmd_name )
```

Write an error message to stderr.

Used in the function [exec](#) to write an error message to stderr when a command fails.

It's meant to be used after `execvp` fails, and the redirection of stderr to an error log file, via the `dup2` function.

It checks the `errno` to determine the type of error, being:

- ENOENT: command not found
- EACCES: permission denied
- EINVAL: invalid argument(s)

Parameters

<i>cmd_name</i>	The name of the command that failed
-----------------	-------------------------------------

5.8.2.4 exec()

```
int exec (
    Request * r,
    char * output_dir,
    struct timeval start_time )
```

Executes a command, single or piped, using process forking, it also handles the redirection of stdout and stderr to respective task files, as well as the writing of the execution time to a file, and the writing of the task number, command and execution time to the history file.

This is the main function of the [command.c](#) module, used in [orchestrator.c](#) server to execute a command.

First, all the necessary files are created, namely the output directory for the task files, such as the output file, the error file and the time file. Also opens the history file to write the task number, the command and the execution time.

This file's names are defined in the macros [OUTPUT_NAME](#), [ERROR_NAME](#), [TIME_NAME](#) and [HISTORY_NAME](#).

It uses the fork system call to create a child process that will execute the command. The parent process will wait for the child process to finish. This fork call is inside another fork call so the parent process can wait for the child process to finish and send the request to the orchestrator as completed. This way, the server can continue to receive and handle new requests while the child process is executing the command.

The inner parent process, after waiting for the child process to finish executing the command, writes the total time since the request was received and the command finished executing to the time file and the history file.

It uses the [parse_cmd](#) function to parse the command into an array of arguments. It also uses the [parse_cmd_pipes](#) function in case the command is piped.

After the command is parsed, it is executed using the `execvp` system call. When `execvp` fails (e.g returns -1) it writes an error message to stderr using the [write_error](#) function.

Parameters

<i>r</i>	The request containing the command to execute, if the command is piped and the task number
<i>output_dir</i>	The output directory to create the task directory and respective task result files
<i>start_time</i>	The time when the execute request was received to calculate the total time since the request was received and the command finished executing

Returns

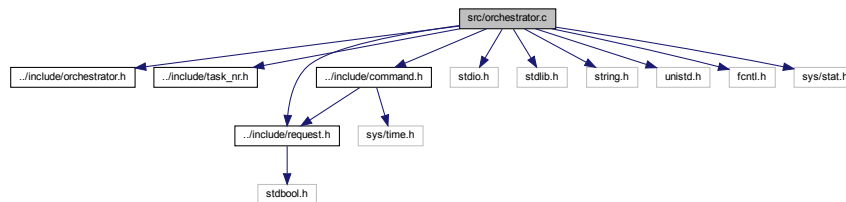
0 if successful

5.9 src/orchestrator.c File Reference

```
#include "../include/orchestrator.h"
```

```
#include "../include/task_nr.h"
#include "../include/request.h"
#include "../include/command.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
```

Include dependency graph for orchestrator.c:



Macros

- `#define` `DEFAULT_POLICY` `SJF`
Default scheduling policy.
- `#define` `MAX_SCHEDULED_REQUESTS` `1024`
Max number of scheduled requests.

Functions

- void `orchestrator_usage` (char *name)
Print the usage of the orchestrator server executable.
- char * `policy_to_string` (int policy)
Convert the scheduling policy to a string.
- int `add_request` (Request *requests[], int *N, int max, Request *r)
Add a request to the requests array.
- int `remove_request` (Request *requests[], int *N, Request *r)
Remove a request from the requests array.
- Request * `select_request` (Request *scheduled[], int N, int policy)
Select a request from the requests array based on the scheduling policy.
- int `handle_execute` (Request *r, Request *executing[], int *N_executing, Request *scheduled[], int *N_scheduled, char *output_dir, unsigned int tasks, unsigned int *task_nr, struct timeval start_time)
Handle the execute request sent by a client.
- int `handle_status` (Request *r, Request *executing[], int N_executing, Request *scheduled[], int N_scheduled, char *output_dir)
Handle the status request sent by a client.
- int `handle_completed` (Request *r, Request *executing[], int *N_executing, Request *scheduled[], int *N_scheduled, char *output_dir, int policy, struct timeval start_time)
Handle the completed request sent by the parent process of the child process that executed the command.
- int `send_status` (char *client_fifo, Request *executing[], int N_executing, Request *scheduled[], int N_scheduled, char *output_dir)

Send the status of the executing, scheduled and completed requests to the client via the client FIFO that sent the status request.

- void `clean_up` (`Request` *executing[], int N_executing, `Request` *scheduled[], int N_scheduled)

Free the memory allocated for the executing and scheduled requests.

- void `clean_up_all` (int fd, `Request` *r, `Request` *executing[], int N_executing, `Request` *scheduled[], int N_scheduled)

Free the memory allocated for the executing and scheduled requests, close the server FIFO file descriptor, and free the memory allocated for the request.

- int `main` (int argc, char **argv)

Main function for the orchestrator server program.

5.9.1 Macro Definition Documentation

5.9.1.1 DEFAULT_POLICY

```
#define DEFAULT_POLICY SJF
```

Default scheduling policy.

5.9.1.2 MAX_SCHEDULED_REQUESTS

```
#define MAX_SCHEDULED_REQUESTS 1024
```

Max number of scheduled requests.

5.9.2 Function Documentation

5.9.2.1 orchestrator_usage()

```
void orchestrator_usage (
    char * name )
```

Print the usage of the orchestrator server executable.

Parameters

<i>name</i>	The name of the orchestrator server executable
-------------	--

5.9.2.2 policy_to_string()

```
char * policy_to_string (
    int policy )
```

Convert the scheduling policy to a string.

Parameters

<i>policy</i>	The scheduling policy
---------------	-----------------------

Returns

The scheduling policy as a string

5.9.2.3 add_request()

```
int add_request (
    Request * requests[],
    int * N,
    int max,
    Request * r )
```

Add a request to the requests array.

If the array is full, the function will return -1.

Used to add requests to the executing and scheduled arrays.

Parameters

<i>requests</i>	The array of requests
<i>N</i>	The number of requests in the array
<i>max</i>	The max number of requests in the array
<i>r</i>	The request to add

Returns

0 if the request was added successfully, -1 otherwise

5.9.2.4 remove_request()

```
int remove_request (
    Request * requests[],
```

```
int * N,
Request * r )
```

Remove a request from the requests array.

If the request is not found in the array, the function will return -1.

Used to remove requests from the executing and scheduled arrays.

Parameters

<i>requests</i>	The array of requests
<i>N</i>	The number of requests in the array
<i>r</i>	The request to remove

Returns

0 if the request was removed successfully, -1 otherwise

5.9.2.5 select_request()

```
Request * select_request (
    Request * scheduled[],
    int N,
    int policy )
```

Select a request from the requests array based on the scheduling policy.

The function will return `NULL` if the array is empty.

The function will return the first scheduled request if the scheduling policy is [FCFS](#).

The function will return the request with the smallest estimated time if the scheduling policy is [SJF](#).

The function will return the request with the highest priority if the scheduling policy is [PES](#).

Parameters

<i>requests</i>	The array of requests
<i>N</i>	The number of requests in the array
<i>policy</i>	The scheduling policy applied

Returns

The selected request

5.9.2.6 handle_execute()

```
int handle_execute (
    Request * r,
    Request * executing[],
    int * N_executing,
    Request * scheduled[],
    int * N_scheduled,
    char * output_dir,
    unsigned int tasks,
    unsigned int * task_nr,
    struct timeval start_time )
```

Handle the execute request sent by a client.

The function will execute the request if the number of executing requests is less than the max number of parallel tasks, otherwise the request will be scheduled.

The function will send a message to the client with the task number and the status of the request - either executing or scheduled.

It uses the function [exec](#) to execute the command, defined in [command.c](#).

Parameters

<i>r</i>	The request to execute
<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests
<i>output_dir</i>	The output directory
<i>tasks</i>	The max number of parallel tasks
<i>task_nr</i>	The task number
<i>start_time</i>	The start time of the request

Returns

0 if the request was executed or scheduled successfully, -1 otherwise

5.9.2.7 handle_status()

```
int handle_status (
    Request * r,
    Request * executing[],
    int N_executing,
    Request * scheduled[],
    int N_scheduled,
    char * output_dir )
```

Handle the status request sent by a client.

The function will send the status of the executing, scheduled and completed requests to the client via the client FIFO located in the request.

This function creates a process to send the status to the respective client, this way, other clients can still send requests to the server and receive responses.

It uses the history file ([HISTORY_NAME](#)) to get the completed requests, and the function [send_status](#) to send the status to the client.

Parameters

<i>r</i>	The request to handle
<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests
<i>output_dir</i>	The output directory

Returns

0 if the status was sent successfully, -1 otherwise

5.9.2.8 handle_completed()

```
int handle_completed (
    Request * r,
    Request * executing[],
    int * N_executing,
    Request * scheduled[],
    int * N_scheduled,
    char * output_dir,
    int policy,
    struct timeval start_time )
```

Handle the completed request sent by the parent process of the child process that executed the command.

The function will remove the request from the executing array and check for scheduled requests to run.

If there are scheduled requests, the function will remove the request from the scheduled array, add it to the executing array, and execute the command.

If there's an available scheduled request, the function will send a message to the client with the task number and the status of the request - either executing or scheduled.

It uses the function [exec](#) to execute the command, defined in [command.c](#), and the function [select_request](#) to select the next scheduled request, if any exists, to execute.

Parameters

<i>r</i>	The request to handle
<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests
<i>output_dir</i>	The output directory
<i>policy</i>	The scheduling policy applied

Returns

0 if the request was handled successfully, -1 otherwise

5.9.2.9 send_status()

```
int send_status (
    char * client_fifo,
    Request * executing[],
    int N_executing,
    Request * scheduled[],
    int N_scheduled,
    char * output_dir )
```

Send the status of the executing, scheduled and completed requests to the client via the client FIFO that sent the status request.

The function is used by the [handle_status](#) function as an auxiliary function to send the status to the client.

See also

[handle_status](#)

Parameters

<i>client_fifo</i>	The client FIFO name to send the status response to the client
<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests
<i>output_dir</i>	The output directory to get the history file in in order to get and send the completed requests

Returns

0 if the status was sent successfully, -1 otherwise

5.9.2.10 clean_up()

```
void clean_up (
    Request * executing[],
    int N_executing,
    Request * scheduled[],
    int N_scheduled )
```

Free the memory allocated for the executing and scheduled requests.

Parameters

<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests

5.9.2.11 clean_up_all()

```
void clean_up_all (
    int fd,
    Request * r,
    Request * executing[],
    int N_executing,
    Request * scheduled[],
    int N_scheduled )
```

Free the memory allocated for the executing and scheduled requests, close the server FIFO file descriptor, and free the memory allocated for the request.

See also

[clean_up](#)

Parameters

<i>fd</i>	The server FIFO file descriptor
<i>r</i>	The request to free
<i>executing</i>	The array of executing requests
<i>N_executing</i>	The number of executing requests
<i>scheduled</i>	The array of scheduled requests
<i>N_scheduled</i>	The number of scheduled requests

5.9.2.12 main()

```
int main (
    int argc,
    char ** argv )
```

Main function for the orchestrator server program.

The orchestrator program is responsible of:

- **parsing the command-line interface arguments;**

- **creating the server FIFO** based on the [SERVER_FIFO](#) macro;
- **receiving and handling client requests** sent via the server FIFO;
- **sending the appropriate responses** to the clients via their respective FIFOs;

It receives the output directory, the number of parallel tasks, and, optionally, the scheduling policy as arguments.

If the output directory doesn't exist, it will be created.

If the number of parallel tasks is not provided or is less than or equal to zero, the program will print an error message and exit.

If the scheduling policy is not provided, the default policy is applied via the [DEFAULT_POLICY](#) macro.

It's responsible for executing or scheduling requests with type [EXECUTE](#) sent from the clients. If the max number of parallel tasks is reached, the request will be scheduled. Based on this action, the orchestrator program will send a response to the client with the task number and the status of the request - either executing or scheduled.

See also

[handle_execute](#)

Once a command finishes its execution, the orchestrator program will receive a request with type [COMPLETED](#) sent from the parent process of the child process that executed the command, and will execute next available scheduled request, if any exists, based on the scheduling policy applied.

See also

[handle_completed](#)

The orchestrator program can also receive requests with type [STATUS](#) from the clients, to check the status of the executing, scheduled and completed requests. The response will be sent via the client FIFO.

See also

[handle_status](#)

The orchestrator program can also receive requests with type [KILL](#) from the clients, to shutdown the server.

When the orchestrator program is shutting down, it will save the task number to a file in the output directory (see [save_task_nr](#)), close the open FIFOs file descriptors, and free the memory allocated for the executing and scheduled requests using the [clean_up](#) function.

If any of the orchestrator components or called functions fail, the program will print an error message, clean up the memory and close file descriptors using the [clean_up_all](#) function, and return 1.

The orchestrator program will run until it receives a request with type [KILL](#) from a client, or until it receives a signal to terminate or interrupt the program.

Parameters

<i>argc</i>	The number of arguments
<i>argv</i>	The arguments

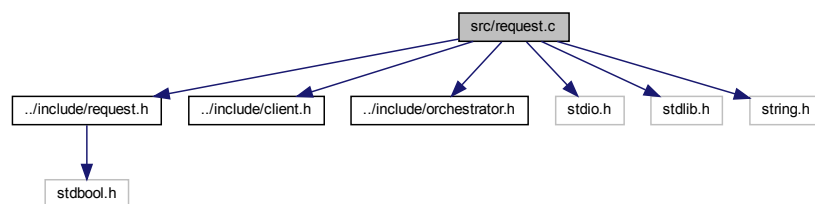
Returns

0 if the program runs successfully, 1 otherwise

5.10 src/request.c File Reference

```
#include "../include/request.h"
#include "../include/client.h"
#include "../include/orchestrator.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Include dependency graph for request.c:



Classes

- struct [request](#)

The request struct.

Functions

- [Request](#) * [create_request](#) (int type, int est_time, char *command, bool is_piped, char *client_fifo)
Creates a new request.
- int [get_type](#) ([Request](#) *r)
Get the type of the request.
- int [get_est_time](#) ([Request](#) *r)
Get the estimated time or priority of the request.
- char * [get_command](#) ([Request](#) *r)
Get the command to be executed.
- bool [get_is_piped](#) ([Request](#) *r)
Get the boolean indicating if the command is piped.
- unsigned int [get_task_nr](#) ([Request](#) *r)
Get the task number of the request.
- char * [get_client_fifo](#) ([Request](#) *r)
Get the client FIFO name to send the response to the client.
- void [set_type](#) ([Request](#) *r, int type)
Set the type of the request.
- void [set_task_nr](#) ([Request](#) *r, unsigned int task_nr)
Set the task number of the request.

- void `set_client_fifo` (`Request` *r, char *client_fifo)
Set the client FIFO name to send the response to the client.
- void `print_request` (`Request` *r, int policy)
Print the request.
- char * `type_to_string` (int type)
Convert the type of the request to a string.
- unsigned long `sizeof_request` ()
Get the size of the request.
- `Request` * `clone_request` (`Request` *r)
Clone a request.

5.10.1 Function Documentation

5.10.1.1 `create_request()`

```
Request* create_request (
    int type,
    int est_time,
    char * command,
    bool is_piped,
    char * client_fifo )
```

Creates a new request.

Allocates memory for a new request and initializes it with the given parameters.

It initializes the task number with 0, which is not a valid task number.

For the creation of a status request, the

Parameters

<i>est_time</i>	should be 0, as it is not used, as well as the
<i>command</i>	should be NULL and
<i>is_piped</i>	should be <code>false</code> .

Example of the creation of an execute request:

```
Request *execute_request = create_request(EXECUTE, 12, "ls -l | cat", true, "client-1234");
```

Example of the creation of a status request:

```
Request *status_request = create_request(STATUS, 0, NULL, false, "client-1234");
```

Example of the creation of a kill request:

```
Request *kill_request = create_request(KILL, 0, NULL, false, "");
```

The caller is responsible for freeing the returned request by this function.

Parameters

<i>type</i>	the type of the request
-------------	-------------------------

Parameters

<i>est_time</i>	the estimated time or priority of the request
<i>command</i>	the command to be executed
<i>is_piped</i>	a boolean indicating if the command is piped
<i>client_fifo</i>	the client FIFO name to send the response to the client

5.10.1.2 get_type()

```
int get_type (  
    Request * r )
```

Get the type of the request.

Parameters

<i>r</i>	The request to get the type from
----------	----------------------------------

Returns

The type of the request

5.10.1.3 get_est_time()

```
int get_est_time (  
    Request * r )
```

Get the estimated time or priority of the request.

Parameters

<i>r</i>	The request to get the estimated time or priority from
----------	--

Returns

The estimated time or priority of the request

5.10.1.4 get_command()

```
char* get_command (  
    Request * r )
```

Get the command to be executed.

Parameters

<i>r</i>	The request to get the command from
----------	-------------------------------------

Returns

The command to be executed

5.10.1.5 get_is_piped()

```
bool get_is_piped (  
    Request * r )
```

Get the boolean indicating if the command is piped.

Parameters

<i>r</i>	The request to get the is_piped from
----------	--------------------------------------

Returns

The boolean indicating if the command is piped

5.10.1.6 get_task_nr()

```
unsigned int get_task_nr (  
    Request * r )
```

Get the task number of the request.

Parameters

<i>r</i>	The request to get the task number from
----------	---

Returns

The task number of the request

5.10.1.7 get_client_fifo()

```
char* get_client_fifo (  
    Request * r )
```

Get the client FIFO name to send the response to the client.

Parameters

<i>r</i>	The request to get the client FIFO name from
----------	--

Returns

The client FIFO name to send the response to the client

5.10.1.8 set_type()

```
void set_type (
    Request * r,
    int type )
```

Set the type of the request.

Parameters

<i>r</i>	The request to set the type
<i>type</i>	The type to set

5.10.1.9 set_task_nr()

```
void set_task_nr (
    Request * r,
    unsigned int task_nr )
```

Set the task number of the request.

Parameters

<i>r</i>	The request to set the task number
<i>task_nr</i>	The task number to set

5.10.1.10 set_client_fifo()

```
void set_client_fifo (
    Request * r,
    char * client_fifo )
```

Set the client FIFO name to send the response to the client.

Parameters

<i>r</i>	The request to set the client FIFO name
<i>client_fifo</i>	The client FIFO name to set

5.10.1.11 print_request()

```
void print_request (
    Request * r,
    int policy )
```

Print the request.

Prints the request attributes to the standard output.

Based on the policy being used, it prints the estimated time or the priority.

If the policy is FCFS, it doesn't print either the estimated time or the priority.

Parameters

<i>r</i>	The request to print
<i>policy</i>	The policy being used

5.10.1.12 type_to_string()

```
char* type_to_string (
    int type )
```

Convert the type of the request to a string.

Possible types are [EXECUTE](#), [STATUS](#), [COMPLETED](#) and [KILL](#).

Parameters

<i>type</i>	The type of the request
-------------	-------------------------

Returns

The string representation of the type of the request

5.10.1.13 sizeof_request()

```
unsigned long sizeof_request ( )
```

Get the size of the request.

Used due to the fact that the [Request](#) struct is not visible to other modules, so `sizeof(Request)`

wouldn't give the correct size of the struct.

Returns

The size of the Request struct

5.10.1.14 clone_request()

```
Request* clone_request (
    Request * r )
```

Clone a request.

Allocates memory for a new request and initializes it with the attributes of the given request.

Used in the [orchestrator.c](#) to clone a request before adding it to the executing or scheduled array.

The caller is responsible for freeing the returned request by this function.

Parameters

<i>r</i>	The request to clone
----------	----------------------

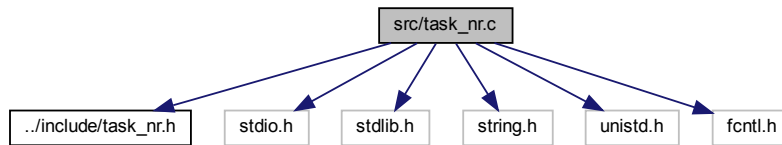
Returns

The cloned request

5.11 src/task_nr.c File Reference

```
#include "../include/task_nr.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

Include dependency graph for task_nr.c:



Macros

- `#define TASK_NR_FILENAME "task_number"`
The filename of the task number file.

Functions

- `char * get_task_nr_filename (char *output_dir)`
Get the filename of the task number file in the output directory.
- `unsigned int load_task_nr (char *output_dir)`
Load the task number from the file in the output directory.
- `unsigned int save_task_nr (unsigned int task_nr, char *output_dir)`
Save the task number to the file in the output directory.

5.11.1 Macro Definition Documentation

5.11.1.1 TASK_NR_FILENAME

```
#define TASK_NR_FILENAME "task_number"
```

The filename of the task number file.

5.11.2 Function Documentation

5.11.2.1 get_task_nr_filename()

```
char * get_task_nr_filename (
    char * output_dir )
```

Get the filename of the task number file in the output directory.

The task number filename is defined in the [TASK_NR_FILENAME](#) macro.

Used both by [load_task_nr](#) and [save_task_nr](#) as an auxiliary function.

The caller is responsible for freeing the returned string by this function.

Parameters

<i>output_dir</i>	The output directory to append the task number filename
-------------------	---

Returns

The filename of the task number file, or NULL if an error occurs

5.11.2.2 load_task_nr()

```
unsigned int load_task_nr (
    char * output_dir )
```

Load the task number from the file in the output directory.

If the file does not exist, create it with the initial value 1.

Used by the [orchestrator.c](#) to load the task number when starting.

Parameters

<i>output_dir</i>	The output directory to load the task number file
-------------------	---

Returns

The task number, or 0 if an error occurs

5.11.2.3 save_task_nr()

```
unsigned int save_task_nr (
    unsigned int task_nr,
    char * output_dir )
```

Save the task number to the file in the output directory.

Used by the [orchestrator.c](#) to save the task number before exiting.

Parameters

<i>task_nr</i>	The task number to save
<i>output_dir</i>	The output directory to write the task number file

Returns

The task number, or 0 if an error occurs