FLMNGR

Release v0.1

Guilherme Araujo, Lucas Airam, Fernando Dias

Apr 08, 2025

CONTENTS:

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

TUTORIAL

The purpose of this artifact is to exemplify the use of the AGATA tool through an example and two experiments:

- The basic example involves manually initializing microservices and starting a task via a web interface.
- The first experiment initializes the server's microservices and then creates and initializes a simple learning task on the server, executed on the local machine. Then, the client microservices are initialized on the same machine, automatically transferring the task to the client.
- The second experiment differs from the first by initializing one task with an error (E) and another correct one ©. When an error occurs in E, the client automatically switches the task to C.

1.1 Basic Information

The experiments were conducted on different physical and virtual machines with the following specifications:

- VM with 4 CPUs, 8GB RAM, and Debian 12, instantiated on a server with an Intel Xeon E5-2650 CPU, 8 cores, and 16 threads, 2.80GHz, and 32GB RAM.
- PC with Intel i9-10900, 2.80 GHz CPU, 20 threads, 32GB RAM, and Ubuntu 20.04.

Since no performance issues were observed in any of these configurations, execution is assumed to be guaranteed under the following conditions:

- Operating System: Ubuntu 20.04 or Debian 12
- Minimum CPU: Intel i5 8th Generation
- Minimum RAM: 8GB

1.2 Dependencies

The requirements are:

- Python 3.12.7
- Conda (miniconda3)
- Docker 24.0.7

1.3 Installation

The following tutorial presents the manual initialization of microservices. Executing these commands manually, without an automated script, provides a clearer understanding of AGATA's architecture. However, more automated scripts have been provided for running subsequent experiments, greatly simplifying the installation process. All commands must be executed from the root of the repository.

1.3.1 New Conda Environment

Create a Conda environment:

```
conda create -n agata python=3.12.7
```

Activate the Conda environment for the first time:

```
conda activate agata
```

Install dependencies within the Conda environment:

```
conda install pip
pip install -r requirements.txt
```

1.3.2 Configuration File

The `config.ini` file should be configured as follows:

```
[client.broker]
host=localhost
port=8000

[server.broker]
host=localhost
port=9000

[server.gateway]
port=9001

[events]
register_events=false

[client.params]
request_interval=10
```

1.3.3 Server Initialization

Start the Broker

Before executing the command, check if a container named `server-broker-rabbit` exists using `docker ps`. If it does, stop and remove it with:

```
sudo docker stop server-broker-rabbit
sudo docker rm server-broker-rabbit
```

The broker will listen on port 9000 of the host. Ensure no other application is using this port, then execute:

```
sudo docker run -d --hostname broker --rm --name server-broker-rabbit -p 9000:5672␣
↪rabbitmq:3
```

Start the Cloud Gateway

Open a new terminal, activate the environment (`conda activate agata`), and run:

```
python3 -u -m cloud_gateway.http_gateway
```

Start the User Manager

Open a new terminal, activate the environment (`conda activate agata`). Before executing the command below, delete the file `user_manager/db/users.db` if it exists. Then execute:

```
python3 -u -m user_manager.service_user_manager
```

Start the Cloud Task Manager

Open a new terminal, activate the environment (`conda activate agata`). Before executing the command below, delete the file `cloud_task_manager/db/tasks.db` if it exists. Then execute:

```
python3 -u -m cloud_task_manager.service_cloud_ml
```

Start the Task Manager Download Server

Open a new terminal, activate the environment (`conda activate agata`), and execute:

```
python3 -u -m cloud_task_manager.host_tasks $(pwd)/cloud_task_manager
```

1.3.4 Client Initialization

Start the Broker

Before executing the command, check if a container named `client-broker-rabbit` exists using `docker ps`. If it does, stop and remove it with:

```
sudo docker stop client-broker-rabbit
sudo docker rm client-broker-rabbit
```

The broker will listen on port 8000 of the host. Ensure no other application is using this port, then execute:

```
sudo docker run -d --hostname broker --rm --name client-broker-rabbit -p 8000:5672␣
↪rabbitmq:3
```

Start the Client Gateway

Open a new terminal, activate the environment (`conda activate agata`), and execute:

```
python3 -u -m client_gateway.amqp_gateway
```

Start the Client Task Manager

Open a new terminal, activate the environment (`conda activate agata`). This is the **client task manager**, not the one previously started on the server. Before executing the command below, delete all files inside the directories `client_task_manager/tasks/` and `client_task_manager/client_info/`, if they exist. Do **not** delete the directories themselves. Then execute:

```
python3 -u -m client_task_manager.service_client_ml
```

At this point, the client will begin sending requests to the server to:

- Register its information
- Request tasks to download, if available

1.4 Minimum Test

The minimum test depends on manual initialization, as described in the previous section.

1.4.1 Task Execution

Access the Graphical Interface

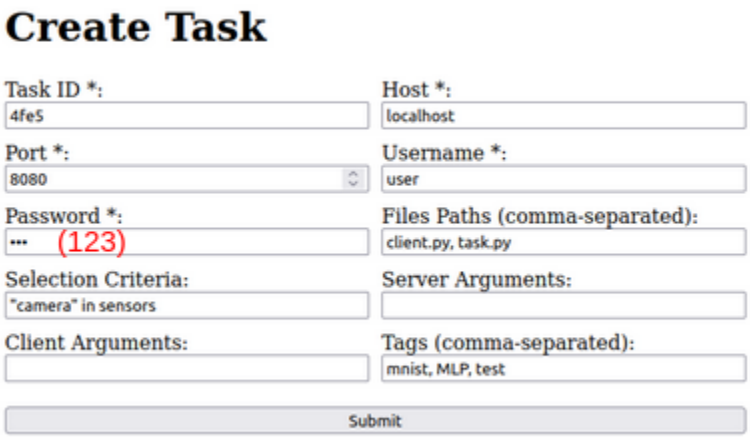
In a new terminal, run the following command and open the local web browser on port 9999 to access a web interface for interacting with the cloud environment. **Acess the web interafce using http://localhost:9999**

```
python3 -m http.server -d cloud_web_interface 9999
```

Register the Task

Click on the `Create task` link and fill in the form fields as shown in the image below.

After clicking `Submit`, the cloud task manager will register a new task in its database. The files for this task are located in `cloud_task_manager/tasks/task_4fe5/*`. The file upload could be done by accessing the `Upload task files` option in the main menu, but this step was omitted for simplicity.



Start the Task on the Server

In the main menu, click on the `Start task` link, provide the previously added ID (`4fe5`), and leave the arguments field empty. Upon submitting the form, the task should start in the cloud task manager. At this point, the client task manager should download the task and start it.

Start a Second Test Client

The task server is configured to require at least two clients to progress through rounds. Therefore, we will start a new Flower process directly. Open a new terminal, activate the environment (`conda activate agata`), and run:

```
python3 -u -m cloud_task_manager.tasks.task_4fe5.client cli
```

The task should progress through only one round. Once completed, it should be noted that the task automatically becomes inactive on both the client and server, and it can be triggered again via the task start interface.

1.4.2 Completion

After the minimum test, services can be stopped (Ctrl + C), as well as brokers (`docker stop [container_name] ; docker rm [container_name]`). The environment can be deactivated (`conda deactivate`).

1.5 Experiments

The experiments are executed using automated scripts. Before running the first experiment, ensure that any Python processes previously started in this artifact are terminated. Do not worry, as containers will be stopped and databases will be deleted by the experiment scripts. The most important thing is that no Python microservice is running to avoid network port conflicts, for example.

1.5.1 Configuration Modifications

Modify the following line in the `config.ini` file:

```
[events]
register_events=true
```

In both cases, the results are presented in the files `experiments/events.json`, `experiments/exp_*_raw_times`, and `logs_*/*`

1.5.2 Experiment 1

To run the first experiment, described at the beginning of this artifact, execute:

```
conda activate agata
bash experiments/exp1.sh
```

Superuser permission will be required to run Docker. The experiment takes approximately 4 minutes. The reviewer can follow the progress of the experiment in the terminal. Details about the experiment are found in the article. Task registration and initialization occur via command line instead of a graphical interface. The most relevant results are:

- The `experiments/events.json` file presents, in execution order, the main steps required for executing the federated learning task, along with the corresponding timestamp and component where they occur.
- The `experiments/exp1_raw_times` file summarizes the time taken for the most important operations.

1.5.3 Experiment 2

To run the second experiment, described at the beginning of this artifact, execute:

```
conda activate agata
bash experiments/exp2.sh
```

Similarly, observe the files `experiments/events.json` and `experiments/exp2_raw_times`.

CHAPTER

TWO

INTERNAL MODULES

2.1 Flower Tasks Daemon Library (FTDL)

This is a library for starting a Flower task as a child process. It also receives information from this task for logging and for finishing the task.

The task initializers uppon execution, create a Flower child process that reports message to the task initializer using a UDP socket (client is the task reporter and server is the task listener).

Here are the commands to run the server in one terminal, followed by two clients in other terminals:

```
python -m task_daemon_lib.server_side_task
```

```
python -m task_daemon_lib.client_side_task
```

```
python -m task_daemon_lib.client_side_task
```

2.1.1 How to develop Flower tasks compatible with FTDL?

The following requisites must be met:

- Client and server main files which will be executed must have paths: `tasks/task_[id]/client.py` and `tasks/task_[id]/server.py`
- The path to the tasks files dir (p.e. `tasks/task_[id]`) must be added to the Python system path for imports. This can be achieved by adding the following code snippet at the begining of `client.py` and `server.py` files:

```
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '.')))
```

- The first CLI argument after the file name, stored at `sys.argv[1]` is the working directory of the component from where the client/server were called. It must also be added to the Python system path, as follows:

```
sys.path.append(sys.argv[1])
```

- Import `TaskReporter` from FTDL for reporting information and errors at both client and server:

```
from task_daemon_lib.task_reporter import TaskReporter
task_reporter = TaskReporter()
```

- The main loop of both client and server must be inside a `try` block. Uppon an unhandled exception is raised, an error must be reported to FTDL daemon who started the task.

```
try:
    start_client(
        server_address="127.0.0.1:8080",
        client=FlowerClient().to_client(),
    )
except Exception as e:
    task_reporter.send_error(e)
```

2.2 Microservice interconnection library

You must have Docker installed. For testing, run:

```
docker stop server-broker-rabbit
docker rm server-broker-rabbit
docker run -d --hostname broker --name server-broker-rabbit -p 5000:5672
```

Now, run the microservice:

```
python -m microservice_interconnect.sample_service
```

Finally, in other terminal, run the tests:

```
python -m microservice_interconnect.test_sample_service
```

2.3 Cloud Task Manager

This microservice is responsible for:

- Implementing CRUD operations over all tasks, running or not
- Mantaining client and server codes for inference, training, and aggregation for all tasks
- Starting and stopping Flower aggreagtion services at the server by demand
- Logging Flower process information and errors (NOT YET IMPLEMENTED)

2.3.1 Test Flower task

In one terminal, run the Flower server

```
python -m cloud_task_manager.tasks.task_4fe5.server cli
```

In two other terminals, run two Flower clients after waiting the server to start

```
python -m cloud_task_manager.tasks.task_4fe5.client cli
```

```
python -m cloud_task_manager.tasks.task_4fe5.client cli
```

2.3.2 Test Cloud Task Manager service

For testing the service with RabbitMQ broker, run:

```
docker stop server-broker-rabbit
docker rm server-broker-rabbit
docker run -d --hostname broker --name server-broker-rabbit -p 9000:5672 rabbitmq:3
python -m cloud_task_manager.test_cloud_ml_logic
```

2.4 Client Task Manager

This microservice is responsible for:

- Periodically providing local client statistics for the cloud
- Periodically requesting available compatible tasks to the cloud
- Downloading client codes for training received tasks
- Starting and stopping Flower training services at the client in real time
- Logging Flower process information and errors (NOT YET IMPLEMENTED)

2.4.1 Test Client Task Manager service

For testing the service with RabbitMQ broker, run in different terminals:

```
docker stop server-broker-rabbit
docker rm server-broker-rabbit
docker run -d --hostname broker --name server-broker-rabbit -p 9000:5672 rabbitmq:3
python -m cloud_task_manager.service_cloud_ml
```

```
python -m cloud_task_manager.create_and_run_server_task
python -m cloud_task_manager.host_tasks $(pwd)/cloud_task_manager
```

```
python -m cloud_gateway.http_gateway
```

```
docker stop client-broker-rabbit
docker rm client-broker-rabbit
docker run -d --hostname broker --name client-broker-rabbit -p 8000:5672 rabbitmq:3
python -m user_manager.service_user_manager
```

```
python -m client_gateway.amqp_gateway
```

```
python -m client_task_manager.service_client_ml
```

```
python -m cloud_task_manager.tasks.task_4fe5.client cli
```

CHAPTER

THREE

APIS

3.1 Flower Tasks Daemon Library (FTDL)

3.1.1 Task

class task_daemon_lib.task.**Task**(*work_path: str*, *task_id: str*, *enable_listener: bool = True*)

A Task is an interface for running code as a child process This is a base class, inherited by ClientSideTask and ServerSideTask

Parameters

- **work_path** (*str*) – project location, within which "tasks" dir resides
- **task_id** (*str*) – hexadecimal number identifying task

Raises

FileNotFoundError – no directory named "{work_path}/tasks/task_{task_id}"

run_task(*filename: str*, *message_handler: Callable[[bytes], None]*, *arguments: list[str]*, *add_work_path: bool = True*)

Start child process with 'filename', as well as the message listener

Parameters

- **filename** (*str*) – complete path for executable file
- **message_handler** (*Callable[[bytes],None]*) – handler function for receiving bytes sent by process
- **arguments** (*list[str]*) – CLI args after "python3 {filename} {work_path} …"
- **add_work_path** (*bool*) – if true, call "python3 {filename} {work_path} {args}". Else, call "python3 {filename} {args}"

Raises

- *TaskAlredyRunning* – starting a task that is alredy running
- **PermissionError** – doesn't have permission to run the task script

stop_task()

Stop child process started before, as well as the message listener

Raises

TaskAlredyStopped – stopping a task that is not running

3.1.2 Client-side task

class task_daemon_lib.client_side_task.**ClientSideTask**(*work_path: str*, *task_id: str*, *message_handler: Callable[[bytes], None]*, *arguments: list[str] = None*)

Interface for running Flower client code as a child process

Parameters

- **work_path** (*str*) – project location, within which "tasks" dir resides.
- **task_id** (*Callable[[bytes],None]*) – task ID
- **message_handler** – handler function for receiving bytes sent by process
- **arguments** (*list[str]*) – CLI args after "python3 {filename} {work_path} ..."

Raises

FileNotFoundError – "{work_path}/tasks/task_{task_id}/client.py" does not exist

run_task_client()

Run "{work_path}/tasks/task_{task_id}/client.py" as a subprocess and starts message listener, which calls self.message_handler(bytes) uppon receiving bytes from child

Raises

- *TaskAlredyRunning* – starting a task that is alredy running
- **PermissionError** – doesn't have permission to run the task script

stop_task_client()

Stop process and message listener

Raises

TaskAlredyStopped – stopping a task that is not running

3.1.3 Server-side task

class task_daemon_lib.server_side_task.**ServerSideTask**(*work_path: str*, *task_id: str*, *message_handler: Callable[[bytes], None]*, *arguments: list[str] = None*)

Interface for running Flower server code as a child process

Parameters

- **work_path** (*str*) – project location, within which "tasks" dir resides
- **task_id** (*str*) – task ID
- **message_handler** (*Callable[[bytes],None]*) – handler function for receiving bytes sent by process
- **arguments** (*list[str]*) – CLI args after "python3 {filename} {work_path} ..."

Raises

FileNotFoundError – "{work_path}/tasks/task_{task_id}/server.py" does not exist

run_task_server()

Run "{work_path}/tasks/task_{task_id}/server.py" as a subprocess and starts message listener, which calls self.message_handler(bytes) uppon receiving bytes from child

Raises

- *TaskAlredyRunning* – starting a task that is alredy running
- **PermissionError** – doesn't have permission to run the task script

stop_task_server()

Stop process and message listener

Raises

TaskAlredyStopped – stopping a task that is not running

3.1.4 Task listener

class task_daemon_lib.task_listener.**TaskMessageListener**(*handler: Callable[[bytes], None]*, *process: Popen*)

Interface for listen and forward bytes received from processes through a two POSIX pipes to a handler function

Parameters

- **handler** (*Callable[[bytes],None]*) – function called every time a line (byte package) is received
- **process** (*subprocess.Popen*) – running subprocess which contains the write ends of sterr and stdout pipes

start()

Creates a thread for listening pipes and forwarding bytes to handler

Raises

Exception – error

stop()

Stops the listening thread

IMPORTANT NOTE: Depending on the received message, the handler function can kill the process and, therefore, kill this thread. This would cause a thread to terminate itself, which is not allowed (RuntimeError). This case is handled.

3.1.5 Task reporter

class task_daemon_lib.task_reporter.**TaskReporter**

Used by the process to send messages through the stdout pipe

send_error(*excpetion: Exception*)

Format: {"type": "error","exception": str(excpetion)}

Parameters

excpetion (*Exception*) – unhandled exception that occured on child process. Typically, causes message listener termination

send_info(*info: str*)

Format: {"type": "info","info": info}

Parameters

info (*str*) – or warning, or generic information

send_print(*msg: str*)

Format: {"type":"print","message":msg}

Parameters

msg (*str*) – a log, or generic message

send_stats(*task_round: int*, *acc: int*)

Format: {"type": "model","round": str(task_round),"acc": str(acc)}

Parameters

- **task_round** (*int*) – Flower communication round
- **acc** (*int*) – current model accuracy

trigger(*trigger_name: str*, *trigger_arguments: str = ''*)

Format : {"type":"trigger","trigger_name":{trigger_name},"trigger_arguments":{trigger_arguments}}

Parameters

- **trigger_name** (*str*) – name of the code that is executed uppon receiving this trigger
- **trigger_arguments** (*str*) – arguments for running trigger

3.1.6 Task exceptions

exception task_daemon_lib.task_exceptions.**TaskAlredyRunning**(*task_id: str*)

exception task_daemon_lib.task_exceptions.**TaskAlredyStopped**(*task_id: str*)

exception task_daemon_lib.task_exceptions.**TaskIdAlredyInUse**(*task_id: str*)

exception task_daemon_lib.task_exceptions.**TaskIdNotFound**(*task_id: str*)

exception task_daemon_lib.task_exceptions.**TaskUnknownMessageType**

3.2 Cloud Task Manager

3.2.1 Service Cloud ML

exception cloud_task_manager.service_cloud_ml.**CouldNotRetrieveUser**(*user_id: str*)

class cloud_task_manager.service_cloud_ml.**ServiceCloudML**(*workpath: str*, *broker_host: str = 'localhsot'*, *broker_port: str = 5672*)

Main class for Cloud Task Manager microservice that executes the methods for starting/stopping a task at server side

Parameters

- **workpath** (*str*) – project location, within which "tasks" dir resides
- **broker_host** (*str*) – IP or hostname of RPC broker
- **broker_port** (*int*) – RPC broker port

handle_error_from_task(*task_id: str*)

This function is executed to handle an error received by the task message forwarder, which handles messages from the Flower subprocess

Parameters
task_id (*str*) – task ID

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- *tasks_db_interface.TaskNotRegistered* – task not found in database
- **cloud_ml.TaskIdNotFound** – task not found in map

rpc_call_query_client_info(*client_id: str*) → dict

Send an RPC message for client manager service with client ID requesting for its info

Parameters
client_id (*str*) – client ID

Returns
returned JSON from RPC with client info

Return type
dicts

rpc_exec_client_requesting_task(*received: dict*) → list

Receives a validated JSON message from client and verify if there is a compatible task

Parameters
received (*dict*) – JSON containing client ID

Raises

- *criteria_evaluation_engine.InvalidSelCrit* – selection criteria expression is invalid
- **sqlite3.IntegrityError** – could not perform DB statement
- *tasks_db_interface.TaskNotRegistered* – task not found in DB

Returns
list of ditionaries. Each dict is a task with keys such as ID, host, port, tags, ...

Return type
list

rpc_exec_create_task(*received: dict*)

Receives a validated JSON message for configuring a new task in database, but not start yet

Parameters
received (*dict*) – JSON containing task ID, host, port, arguments, selection criteria, tags, ...

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- *tasks_db_interface.TaskNotRegistered* – task not found

rpc_exec_get_task_by_id(*received: dict*) → dict

Get task info from DB using task ID

Parameters
received (*dict*) – JSON containing task_id

Returns received
JSON with task_id, last_mod_date, host, port, running, selection_criteria, server_arguments, client_arguments, username, password, tags, files_paths

Return type
dict

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- ***TaskNotRegistered*** – task not found in DB

rpc_exec_start_server_task(*received: dict*)

Receives a validated JSON message for starting a server task Validation occurs using our RPC library

Parameters
received (*dict*) – JSON containing task ID and optional arguments

Raises

- **FileNotFoundError** – task file not found
- **cloud_ml.TaskIdNotFound** – task ID not found in map
- **cloud_ml.TaskIdAlredyInUse** – tried to start a task that alredy exists in map
- **cloud_ml.TaskAlredyRunning** – tried to start a task that is alredy running
- **PermissionError** – task file cannot be run due to lack of permissions
- **sqlite3.IntegrityError** – could not perform DB statement
- ***tasks_db_interface.TaskNotRegistered*** – task not found in DB

rpc_exec_stop_server_task(*received: dict*)

Receives a validated JSON message for stopping a server task

Parameters
received (*dict*) – JSON containing task ID

Raises

- **cloud_ml.TaskIdNotFound** – task ID not found in map
- **cloud_ml.TaskAlredyStopped** – tried to stop a task that is not running
- **sqlite3.IntegrityError** – could not perform DB statement
- ***tasks_db_interface.TaskNotRegistered*** – task not found in DB

rpc_exec_update_task(*received: dict*)

Receives a validated JSON message with new info for a task

Parameters
received (*dict*) – JSON containing optional task info

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- ***TaskNotRegistered*** – task not found in DB

3.2.2 Process messages from task

class cloud_task_manager.process_messages_from_task.**ForwardMessagesFromTask**(*task_id: str*, *uppon_receiving_error: Callable[[str], None]*, *uppon_receiving_finish: Callable[[str], None]*, *work_path: str*)

Handles messages received from subprocess using task listener from FTDL

Parameters

- **task_id** (*str*) – task ID
- **uppon_receiving_error** (*Callable[[str],None]*) – function that receives task ID for finishing it when "Finished" is received
- **uppon_receiving_error** – function that receives task ID when an error happens

call_coresponding_func_by_type(*message: dict*)

Use received message's type field to find the corresponding private method and call it

Parameters
message (*bytes*) – received message that can be anything

Raises
TaskUnknownMessageType – value corresponding to key "type" is not model, info, message, exception, or trigger

process_error(*exception_message: str*)

Call uppon_receiving_error external function to finish the task

Parameters
exception_message (*str*) – received exception message from task

process_info(*info: str*)

Receives generic textual info. If "Finished", call finishing function

Parameters
info (*str*) – textual information to print

process_messages(*message: bytes*)

Main method for receiving messages from FTDL task listener

Parameters
message (*bytes*) – received message that can be anything

process_model(*message: dict*)

Receives info from model for printing it

Parameters
message (*dict*) – JSON message representing model information

process_print(*message: str*)

Receives text for printing with "P " in front

Parameters
message (*str*) – message for printing

process_trigger(*trigger_name: str*, *trigger_arguments: str*)

Run a triggered code

Parameters

- **trigger_name** (*str*) – name of the code that is executed uppon receiving this trigger
- **trigger_arguments** (*str*) – arguments for running trigger

Raises
FileNotFoundError

Raises
PermissionError

3.2.3 Cloud ML

class cloud_task_manager.cloud_ml.**CloudML**(*work_path: str*)

Start and stop MULTIPLE server-side tasks. Mantain server-side tasks OBJECTS in a map using task ID as key

Parameters
work_path (*str*) – project location, within which "tasks" dir resides

finish_all()

Finishes all tasks

start_new_task(*task_id: str*, *message_handler: Callable[[bytes], None]*, *arguments: str*)

Start mew task and inserts in the map

Parameters

- **task_id** (*str*) – task ID
- **message_handler** (*Callable[[bytes],None]*) – function to forward received messages from task
- **arguments** (*str*) – string to append to the command with arguments separated by " "

Raises

- **FileNotFoundError** – "{work_path}/tasks/task_{task_id}/server.py" does not exist
- ***TaskIdAlredyInUse*** – could not start a task that is alredy in the map
- ***TaskAlredyRunning*** – try to start a task that was not stopped
- **PermissionError** – doesn't have permission to run the task script

stop_task(*task_id: str*)

Stop task and removes from the map

Parameters
task_id (*str*) – task ID

Raises

- ***TaskIdNotFound*** – task to be stopped is not registerd
- ***TaskAlredyStopped*** – try to stop a task that was alredy stopped

3.2.4 Tasks DB interface

exception cloud_task_manager.tasks_db_interface.**TaskNotRegistered**(*task_id: str*)

class cloud_task_manager.tasks_db_interface.**TasksDbInterface**(*work_path: str*)

Tasks DB handler

Parameters
workpath (*str*) – project location, within which "tasks" dir resides

get_task_selection_criteria_map() → dict

Retrieve a dictionary mapping each task ID to its selection criteria.

Returns
dictionary where keys are task IDs and values are selection criteria.

Return type
dict

insert_task(*task_id: str*, *host: str*, *port: int*, *username: str*, *password: str*, *files_paths: list*, *selection_criteria: str = ''*, *server_arguments: str = ''*, *client_arguments: str = ''*, *tags: list = None*)

Insert a new task into the tasks table and optionally insert tags.

Parameters

- **task_id** (*str*) – Unique identifier for the task (4 hex digits).
- **host** (*str*) – hostname or IP of the server
- **port** (*int*) – network port number of the server (unsigned 16-bit integer).
- **selection_criteria** (*str*) – boolean expression for selecting clients using its atributes
- **server_arguments** (*str*) – command line arguments used when starting the task server (optional)
- **client_arguments** (*str*) – command line arguments used when starting the task client (optional)
- **username** (*str*) – username for downloading files for client tasks
- **password** (*str*) – clear password used by the client to download task files
- **files_paths** (*list*) – list of files path that will be downloaded and used by client
- **tags** (*list[str]*) – list of tags associated with the task (optional).

Raises
sqlite3.IntegrityError – could not perform DB statement

query_task(*task_id: str*) → dict

Query a task by its ID, including all associated attributes and tags.

Parameters
task_id (*str*) – ID of the task to query.

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- *TaskNotRegistered* – task not found

Returns
dictionary with task details and associated tags, or None if the task does not exist

Return type
dict

`set_task_not_running`(*task_id: str*)

Set a task as not running by its ID.

Parameters
task_id (*str*) – ID of the task to update

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- ***TaskNotRegistered*** – task not found

`set_task_running`(*task_id: str*)

Set a task as running by its ID.

Parameters
task_id (*str*) – ID of the task to update

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- ***TaskNotRegistered*** – task not found

`update_task`(*task_id: str*, *host: str = None*, *port: int = None*, *running: bool = None*, *selection_criteria: str = None*, *server_arguments: str = None*, *client_arguments: str = None*, *username: str = None*, *password: str = None*)

Update an existing task with new values. Arguments with None are not updated. Not used yet

Parameters

- **task_id** (*str*) – The ID of the task to update
- **host** (*str*) – New hostname or IP address (optional)
- **port** (*int*) – New port number (optional)
- **running** (*bool*) – New running status (optional)
- **selection_criteria** (*str*) – New selection criteria (optional)
- **server_arguments** (*str*) – new command line server_arguments used when starting the task (optional)
- **username** (*str*) – username for downloading files for client tasks
- **password** (*str*) – clear password used by the client to download task files

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- ***TaskNotRegistered*** – task not found

3.2.5 Selection Criteria Evaluation Engine

exception cloud_task_manager.criteria_evaluation_engine.**InvalidSelCrit**(*expression: str*, *e: Exception*)

cloud_task_manager.criteria_evaluation_engine.**eval_select_crit_expression**(*expression: str*, *info: dict*) → bool

Verifies if a client matches the criteria for the task

NOTE: unsafe function

Parameters

- **expression** (*str*) – boolean expression
- **info** (*dict*) – client info

Returns
Ture if the client matches the criteria

Return type
bool

Raises
InvalidSelCrit – expression sysntax is not a valid python statement

3.2.6 Host tasks

3.3 Client Task Manager

3.3.1 Service Client ML

class client_task_manager.service_client_ml.**ServiceClientML**(*workpath: str*, *client_info: dict*, *autorun: bool = True*, *policy: str = 'one'*, *download_url: str = 'http://127.0.0.1:5000'*, *client_broker_host: str = 'localhost'*, *client_broker_port: int = 5672*)

Main class for Client Task Manager microservice This is a stub that executes the methods for starting/stopping a task at server side, but they are not yet connected to the RabbitMQ RPC system

Parameters

- **workpath** (*str*) – project location, within which "tasks" and "client_info" directories reside
- **client_info** (*dict*) – JSON with client basic info
- **autorun** (*bool*) – if True, service constructor blocks the rest of the code and runs a sequence of actions by default
- **policy** (*str*) – if "one", start the one task, using received order as priority. If "all", starts all received tasks
- **download_url** (*str*) – hostname (or IP) and port of the server that hosts tasks to be downloaded. E.g. http://127.0.0.1:5000
- **client_broker_host** (*str*) – hostname or IP of the broker at the client
- **client_broker_port** (*int*) – port of the broker at the client

Raises
NotImplementedError – policy not implemented

handle_error_from_task(*task_id: str*)

This function is executed to handle an error received by the task message forwarder, which handles messages from the Flower subprocess

Parameters
task_id (*str*) – task ID

Raises
TaskIdNotFound – task not found

rpc_call_request_task() → list

Sends an RPC message to cloud task manager requesting compatible tasks

Returns
list of tasks info

Return type
list

rpc_call_send_client_stats()

Get client info stored in client_info dir inside workpath and sends it to the server

start_client_task(*task_id: str*, *arguments: str*)

After downloading task, starts it

Parameters

- **task_id** (*str*) – task ID
- **arguments** (*str*) – command line arguments when startting child task

Raises

- **FileNotFoundError** – "{work_path}/tasks/task_{task_id}/client.py" does not exist
- *TaskIdAlredyInUse* – could not start a task that is alredy in the map
- *TaskAlredyRunning* – try to start a task that was not stopped
- **PermissionError** – doesn't have permission to run the task script

3.3.2 Process messages from client task

class client_task_manager.process_messages_from_client_task.**ForwardMessagesFromClientTask**(*task_id: str*, *upon_receiving_er Callable[[str], None]*, *upon_receiving_fi Callable[[str], None]*)

Handles messages received from subprocess using task listener from FTDL

Parameters

- **task_id** (*str*) – task ID

- **uppon_receiving_error** (*Callable[[str],None]*) – function that receives task ID for finishing it

call_coresponding_func_by_type(*message: dict*)

Use received message's type field to find the corresponding private method and call it

Parameters
message (*bytes*) – received message that can be anything

Raises
TaskUnknownMessageType – value corresponding to key "type" is not model, info, message, exception, or trigger

process_error(*exception_message: str*)

Call uppon_receiving_error external function to finish the task

Parameters
message (*str*) – received exception message from task

process_messages(*message: bytes*)

Main method for receiving messages from FTDL task listener

Parameters
message (*bytes*) – received message that can be anything

3.3.3 Client ML

class client_task_manager.client_ml.**ClientML**(*work_path: str*)

Start and stop client-side tasks. Mantain client-side tasks OBJECTS in a map using task ID as key

Parameters
work_path (*str*) – project location, within which "tasks" dir resides

finish_all()

Finishes all tasks

get_running_tasks() → list

Returns a list with running task's IDs

Returns
list of task IDs

Return type
list

start_new_task(*task_id: str*, *message_handler: Callable[[bytes], None]*, *arguments: str*)

Start mew task and inserts in the map

Parameters

- **task_id** (*str*) – task ID
- **message_handler** (*Callable[[bytes],None]*) – function to forward received messages from task
- **arguments** (*str*) – string to append to the command with arguments separated by " "

Raises

- **FileNotFoundError** – "{work_path}/tasks/task_{task_id}/client.py" does not exist

- *TaskIdAlredyInUse* – could not start a task that is alredy in the map
- *TaskAlredyRunning* – try to start a task that was not stopped
- **PermissionError** – doesn't have permission to run the task script

stop_task(*task_id: str*)

Stop task and removes from the map

Parameters
task_id (*str*) – task ID

Raises

- *TaskIdNotFound* – task to be stopped is not registerd
- *TaskAlredyStopped* – try to stop a task that was alredy stopped

3.3.4 Task files downloader

exception client_task_manager.task_files_downloader.**TaskDownloadAuthFail**(*complete_url: str*)

exception client_task_manager.task_files_downloader.**TaskDownloadGenericError**(*complete_url: str*, *response*)

exception client_task_manager.task_files_downloader.**TaskNotFoundInServer**(*complete_url: str*)

client_task_manager.task_files_downloader.**download_task_training_files**(*task_id: str*, *work_path: str*, *username: str*, *password: str*, *files_paths: list*, *download_server_url: str*)

Download files in list from a web server using credentials

Parameters

- **task_id** (*str*) – task ID
- **work_path** (*str*) – path for client "tasks" dir, inside of which directories will be created for each task
- **username** (*str*) – username used to download file
- **password** (*str*) – password used to download file
- **files_paths** (*str*) – files list
- **download_server_url** (*str*) – URL in the format http://{hostname or IP}:{port}

Raises

- *TaskDownloadGenericError* – received status 50X, which suggests an internal server error
- *TaskDownloadAuthFail* – invalid username and password
- *TaskNotFoundInServer* – task files not found in server

3.3.5 Client info manager

class client_task_manager.client_info_manager.**ClientInfoManager**(*work_path: str*, *id: str*)

Stores in disk and retrieves from disk client info

Parameters

- **workpath** (*str*) – project location, within which "client_info" dir will reside
- **id** (*str*) – client ID

get_info() → dict

Returns JSON read from "{workpath}/client_info/{client_id}_info.json"

Returns
JSON with client info or None

Return type
dict

Raises

- **FileNotFoundError** – client info file not found
- **JSONDecodeError** – client info file invalid format

get_info_if_changed() → dict

Returns not None client info only if it has changed since last call of this method

Returns
JSON with client info or None

Return type
dict

Raises

- **FileNotFoundError** – client info file not found
- **JSONDecodeError** – client info file invalid format

save_complete_info(*client_info: dict*)

Store all client info at "{workpath}/client_info/info.json"

Parameters
client_info (*dict*) – JSON with client info

update_info(*client_info_to_change: dict*)

Change some client info, mantaining the other as they are

Parameters
client_info_to_change (*dict*) – JSON with just client info that may change

Raises

- **FileNotFoundError** – client info file not found
- **JSONDecodeError** – client info file invalid format

3.4 User Manager

3.4.1 User DB interface

class user_manager.user_db_interface.**UserDbInterface**(*work_path: str*)

User DB handler

Parameters
work_path (*str*) – project location

insert_user(*user_id: str*, *sensors: list = []*, *data_qnt: int = 0*, *avg_acc_contrib: float = None*, *avg_discon_per_round: float = None*)

Insert a new user into the stats table and optionally insert sensors

Parameters

- **user_id** (*str*) – Unique identifier for the user (nickname).
- **sensors** (*list*) – list of string with sensors names (strings). Can be empty
- **data_qnt** (*int*) – amount of data used for training (default is 0)
- **avg_acc_contrib** (*float*) – avarege of accuracy increment along tasks rounds (optional)
- **avg_discon_per_round** (*float*) – avarege number of disconnections along tasks rounds (optional)

Raises
sqlite3.IntegrityError – could not perform DB statement

query_user(*user_id: str*) → dict

Query a user by its ID, including all associated attributes and sensors

Parameters
user_id (*str*) – the ID of the user to query.

Raises

- **sqlite3.Error** – could not perform DB statement
- ***UserNotRegistered*** – user not found

Returns
a dictionary with user attributes and sensors list, or None if the user does not exists.

Return type
dict

update_user(*user_id: str*, *data_qnt: int = None*, *avg_acc_contrib: float = None*, *avg_disconnection_per_round: float = None*, *received_sensors: list = None*, *insert_if_dont_exist: bool = True*)

Update an existing client with new attrbutes and sensors. Arguments with None are not updated.

Parameters

- **user_id** (*str*) – the ID of the user to update
- **data_qnt** (*int*) – amount of data used for training (default is 0)
- **avg_acc_contrib** (*float*) – avarege of accuracy increment along tasks rounds (optional)
- **avg_discon_per_round** (*float*) – avarege number of disconnections along tasks rounds (optional)

- **sensors** (*list*) – list of string with sensors names (strings). Can be empty
- **insert_if_dont_exist** (*bool*) – insert new user if it does not exist. Neve raises UserNotRegistered

Raises

- **sqlite3.Error** – could not perform DB statement
- *UserNotRegistered* – user not found

exception user_manager.user_db_interface.**UserNotRegistered**(*user_id: str*)

3.4.2 Service user manager

class user_manager.service_user_manager.**ServiceUserManager**(*workpath: str*, *server_broker_host: str = 'localhost'*, *server_broker_port: int = 5672*)

Main class for User Manager microservice that executes the methods for CRUD operations on users DB

Parameters

- **workpath** (*str*) – project location, within which "tasks" dir resides
- **server_broker_host** (*str*) – hostname or IP of broker
- **server_broker_port** (*int*) – broker port

rpc_exec_get_user_info(*received: dict*)

Receives a validated JSON message for querying for a user in database

Parameters
received (*dict*) – JSON containing user ID

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- *user_db_interface.UserNotRegistered* – user not found

Returns
user attributes and sensors list

Return type
dict

rpc_exec_update_user_info(*received: dict*)

Receives a validated JSON message for inserting or updating a user in database

Parameters
received (*dict*) – JSON containing user ID, user attributes and sensors

Raises

- **sqlite3.IntegrityError** – could not perform DB statement
- *user_db_interface.UserNotRegistered* – user not found

PYTHON MODULE INDEX

c

t

u

INDEX

T

U