MPCS 52040 Distributed Systems (Fall 2024)
Project: Function as a Service (FaaS) platform
Due: 12/6 @ 11:59 PM (No extensions)

The goal of this project is to implement a Function-as-a-Service (FaaS) platform, MPCSFaaS, that allows users to run Python functions in a serverless environment. You will implement MPCSFaaS in Python using the FastAPI framework for the REST API and ZMQ for communication with a pool of worker processes. MPCSFaaS should be optimized for performance to handle many concurrent requests and provide fast response times.

As shown in Figure 1, MPCSFaaS consists of two main components: the MPCSFaaS service (exposing a REST API with Redis database and task dispatcher) and the worker pool. The MPCSFaaS service will receive HTTP requests from users, parse the requests, and perform actions. For example, it will register functions (and store in Redis) and enable users to execute those functions by sending them to available workers in the worker pool via the task dispatcher. The worker pool will receive tasks from the MPCSFaaS service, execute the task, and return the result back to the MPCSFaaS service to be stored in Redis until the user retrieves the task result.

FastAPI is a modern web framework for building APIs with Python. FastAPI provides automatic validation of request and response data, which makes it easier to develop and maintain the API. It also automatically generates API documentation and provides a web-based interface for testing the REST API.

The backend architecture of MPCSFaaS will use ZMQ to establish connections between the MPCSFaaS service and the worker pool, allowing for fast and efficient communication between the two components. You will implement both a push and a pull model for communicating tasks to workers.

The worker pool will consist of a set of worker processes, each of which will execute tasks. When a request is received, the worker process will execute the task and return the result to the MPCSFaaS service. Worker processes will be implemented using Python's multiprocessing module, which allows for easy creation of multiple worker processes.

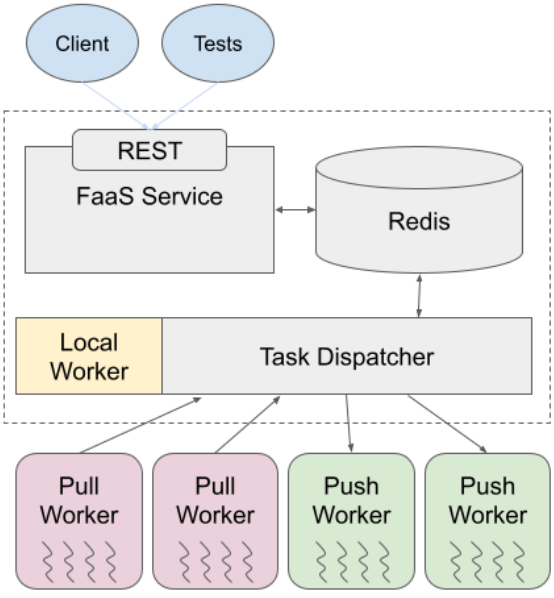

Figure 1: MPCSFaaS architecture

Requirements

- The project should be implemented in Python 3.
- The project may be conducted by teams of up to two students. You are welcome to do the project alone.
- You cannot use existing FaaS libraries.

- You must use the specified technologies (i.e., FastAPI, ZMQ, Redis).
- Please use the provided test suite to ensure your implementation will work with our tests.
- Your workers should communicate via the task dispatcher. They should not access Redis directly.

1 Architecture

We describe in the following the main components of the system. Please refer to Figure 1 to understand how these components interact with one another.

1.1 MPCSFaaS service

The FastAPI-based service should provide functionality to register functions, execute functions (called tasks), retrieve the status of tasks, and fetch results from tasks. State should be stored in a Redis database.

When registering functions, the client should serialize (i.e., with dill, see example code at the end of the document) the function body. You will need to store the function in Redis including some basic metadata (e.g., function name) and the serialized function body. The service should create a unique UUID for each function when it is registered and return the UUID to the client (to be used for invocation).

When executing a function, the client should send the function UUID and the input arguments for that function (also to be serialized by the client) to the service. The service should look up the function in Redis and retrieve the function body. It should then compose a task by creating a unique task UUID and storing the payload in Redis (task_id, fn_payload, param_payload). The task should also have fields for the status (see task lifecycle described below) and the result. Finally, you should publish (i.e., Redis.pubsub) the task (i.e., using a "Tasks" channel) with the task_id. You will later subscribe to this channel in your task dispatcher.

The status interface, should retrieve the status from Redis for a given task UUID.

The results interface should retrieve the result (which may be an exception) for a given task UUID and return that serialized object to the client.

Distributed systems fail in unusual ways and fault tolerance is critical. If a task fails due to execution failure your service should report the failures to the client with useful information about the failure presented as exceptions.

1.2 Task Lifecycle

. You should store the state of a task in Redis. Tasks should progress through the following lifecycle.

1. QUEUED: task is created and stored in the database
2. RUNNING: task has been sent to a worker
3. COMPLETE: task has completed execution successfully
4. FAILED: task has completed running unsuccessfully

1.3 Redis

Redis is an in-memory key-value store that is often used as a database or message broker. As an in-memory database it provides very fast read/write (all data are stored in memory and there is no need to read/write to disk).

Redis implements a key-value model which offers a simple interface for your service and task dispatcher. Each entry is a unique key and an arbitrary value (the data). Redis supports various data types for keys and values, including strings, hashes, lists, sets, and sorted sets. In MPCSFaaS we will use simple strings

for keys (to represent UUIDs) and JSON objects for the data (representing, for example, the function body, input arguments, status, and results).

Redis supports various commands for operating on key-value pairs such as SET and GET to create and retrieve records.

We use Redis for two purposes:

1. Distributed key-value to store registered functions. This data is accessed via the MPCSFaaS service and the task dispatcher.
2. Distributed message queue to store tasks throughout their lifecycle (from when they are created until results are retrieved). Note: you do not need to delete tasks from the Redis queue.

Note: your workers should not interact with Redis directly. All communication with workers should go via the task dispatcher.

1.4 Task dispatcher

The Task Dispatcher is responsible for routing tasks from Redis to workers, detecting failures, and updating Redis when results are available.

We implement the task dispatcher as a separate component to avoid complexity handling multi-threading or multi-processing within the asynchronous MPCSFaaS service.

The task dispatcher must be launched separately alongside the MPCSFaaS service. The required CLI is described in Section 2.2

The task dispatcher should be configured to listen on the Redis tasks channel (i.e., Redis.sub("tasks")) for "task_ids" when in local or push modes. This asynchronous interface will notify the task dispatcher when new tasks are registered in Redis. When in pull mode, the task dispatcher can simply check Redis for new tasks.

When a "task_id" is received, the task dispatcher should obtain the function body and input arguments and dispatch them for execution on available workers. You will implement three different methods for running the workload. Note: in all cases you should explicitly check that functions are running concurrently on several workers and that there are no barriers that force sequential execution.

1. **Local** a baseline implementation for development and testing using a local MultiProcessingPool (`https://docs.python.org/3/library/multiprocessing.html`). Here the task dispatcher can pass a function with the following signature.

   ```
   execute_task(task_id, fn_payload, param_payload) -> (task_id, status, result_payload)
   ```

   It should perform deserialization and execute the function

2. **Pull** the task should be passed via ZMQ REQ/REP to workers. That is, the task dispatcher should listen in a REP loop. Workers in this model will request tasks to execute when they are free. The task dispatcher should send tasks as responses to these requests using the serialized task representation.
3. **Push** the task should be sent to waiting workers using the ZMQ DEALER/ROUTER pattern. The task dispatcher will push tasks as it receives them to available workers.

As described below, the Task dispatcher should detect when workers fail, so that the task dispatcher can determine when tasks may be lost. You can assume that fault-tolerance is not required for the Local execution model.

1.5 Worker pool

You will implement two types of workers: pull workers and push workers, corresponding to the task dispatcher above. The workers should run a process pool with a configurable number of processes to execute tasks (i.e.,

each worker will be able to execute several tasks concurrently). See the code in Section 3.2 for an example of how to execute the task.

When starting a worker, it should send an initial registration message to the task dispatcher. You should store the registered workers in an in-memory data structure in the task dispatcher.

You are free to define your own message structure for registration and passing workload. However, we suggest you use a dictionary and use the same serializer to send/receive as a string (i.e., ZMQ send_string).

The **Pull worker** will use a ZMQ REQ type socket to request tasks from the task-dispatcher. The pull worker should, in a loop, request tasks from the task dispatcher, execute the task, and return the result.

```
while True:
     # Poll the server for tasks (wait some time between requests)
     task = fetch_task(timeout=0.01)
     # Execute the task on a process pool
     # Return result
```

Note: Fetching tasks and returning results should use ZMQ.REQ/REP sockets. As we are using a single socket, you will need to use a lock to do this safely, otherwise you may see errors from ZMQ.

The **Push worker** should use ZMQ's DEALER/ROUTER pattern to push workload to the worker. The task dispatcher should track available workers (based on registrations), send tasks for execution on a worker, and receive results from workers when they complete task execution.

```
while True:
     # receive task
     # execute task asynchronously
     # return result if available
     # heartbeat to server
```

In the push model you should think about how to effectively load balance across workers and implement a solution that minimizes overheads.

1.6 Fault tolerance

You should make your system robust against failures. We consider two types of failures: task failures (e.g., serialization errors or functions with errors) and worker failures (e.g., worker is terminated during task execution). In both cases, you should return appropriate errors and error codes representing the state of the system.

- Failed functions: A function executed on the worker, but failed raising an exception. This exception should be serialized and reported to the user.
- Worker failure: A worker failed during a function execution, or failed to complete the function execution within the specified deadline. The failed function should be reported as a `WorkerFailure` Exception.

For worker failures, you will need to think about how to detect failures (given the challenges doing so in a distributed system). For example, it would be reasonable to implement a heartbeat or timer-based mechanisms and conclude that a worker has failed. In such cases you will need to record which workers are alive/dead and route workload appropriately. You will also need to track where tasks are executing so that you can return errors if workers fail.

In the **Pull** model, the workers connect and request tasks from the dispatcher and reply with results. Implement a configurable deadline-based mechanism where tasks are tracked so that you can mark those that do not complete within the deadline as failed.

For the **Push** model you can implement heartbeats in the communication protocol such that workers that either do not report task completion or miss a heartbeat are marked as failed.

1.7 Performance evaluation

You should implement a client to evaluate the performance of your push and pull implementation. You can use the local backend as a baseline implementation for your system (as it executes tasks immediately on available processes) and compare the overheads of the push and pull workers.

You should think about what dimensions you want to evaluate and how you might quantify the different aspects of system performance (e.g., latency vs throughput). You should consider what types of functions to run as part of this study (e.g., "no-op" tasks that return immediately and/or "sleep" tasks that sleep for a defined period of time).

We suggest implementing a weak scaling study (a common approach in distributed systems). Here you would increase the amount of work as you increase the number of worker processes. That is, you can set the number of tasks to execute per worker (e.g., 5 per worker process) and as you scale the number of workers/processes you increase the problem size (1 worker == 5 tasks, 4 workers == 20 tasks, 8 workers == 40 tasks).

You should submit the client used to run your experiments and a write up of your experiments and findings.

Note: the performance evaluation is worth 10% of your grade and we expect it to be comprehensive.

2 Submission

You should write three reports: 1) technical_report.md: describing how you implemented your solution and limitations of your approach; 2) testing_report.md: a document outlining how you tested your solution; 3) performance_report.md: a document outlining your experiments and results.

Submit via GitHub classroom: `https://classroom.github.com/a/rYzUawR9`

2.1 Grading

Grading of this project is out of 100 points:

- 10 points: REST Service
- 5 points: Task dispatcher (local)
- 10 points: Task dispatcher (pull)
- 10 points: Task dispatcher (push)
- 5 points: Worker (pull)
- 5 points: Worker (push)
- 5 points: Fault tolerance (pull)
- 5 points: Fault tolerance (push)
- 10 points: performance_report.md and client.
- 10 points: technical_report.md
- 05 points: testing_report.md and tests

2.2 Integration tests for grading

We will grade the project using our integration testing framework. This will allow us to automatically stop and start workers as well as register and execute functions. Please follow the interfaces as described in this document.

We provide an initial set of test cases so that you can ensure your implementation follows the defined interfaces.

https://github.com/mpcs-52040/MPCS-FaaS-Tests

You should include these tests exactly as it is specified in the repository. It must be included as a module (in the test directory) to work with automated testing. You should use these tests as a basis for your own testing. That is, you should extend with your own test cases. Please submit all tests with your submission. It will also serve as a framework for you to write your own

3 Implementation and API Specifications

The following section outlines the interfaces you should follow when implementing your system.

Starting Redis. You can run Redis locally with defaults running on port 6379

```
> redis-server
```

Start the MPCSFaaS service

```
> uvicorn main:app --reload
```

Start the task dispatcher The task dispatcher should include arguments to define which mode it is operating [local, push, pull] and with optional port (only for push/pull) and optional number of workers (only for local).

```
> python3 task_dispatcher.py -m [local/pull/push] -p <port> -w <num_worker_processors>
```

Start the workers You can start the workers manually or using a script. You should specify the number of worker processes for each worker and the URL for the dispatcher (e.g., tcp://localhost:5555).

```
> python3 pull_worker.py <num_worker_processors> <dispatcher url>
> python3 push_worker.py <num_worker_processors> <dispatcher url>
```

Testing We use pytests for testing the implementation.

```
> pip install .
> pytest tests
```

Note: Tests should be organized as a Python module.

3.1 REST API

The REST API is specified below using the FastAPI model.

3.1.1 Register function

```
class RegisterFn(BaseModel):
    name: str
    payload: str

class RegisterFnRep(BaseModel):
    function_id: uuid.UUID

/register_function(RegisterFn) -> RegisterFnRep
```

3.1.2 Execute function

```
class ExecuteFnReq(BaseModel):
    function_id: uuid.UUID
    payload: str
```

```
class ExecuteFnRep(BaseModel):
    task_id: uuid.UUID

/execute_function(ExecuteFnReq) -> ExecuteFnRep
```

3.1.3 Task status

```
class TaskStatusRep(BaseModel):
    task_id: uuid.UUID
    status: str

/status/<task_id> -> TaskStatusRep
```

3.1.4 Task result

```
class TaskResultRep(BaseModel):
    task_id: uuid.UUID
    status: str
    result: str

/result/<task_id> -> TaskResultRep
```

3.1.5 Errors

There are many circumstances in which the system will fail. You should generate HTTP error codes for errors and include clear descriptions of the failure.

3.2 Helpful code

Below we provide example code to serialize/deserialize functions and to execute functions.

3.3 Serialization code

```
import dill
import codecs

def serialize(obj) -> str:
    return codecs.encode(dill.dumps(obj), "base64").decode()

def deserialize(obj: str):
    return dill.loads(codecs.decode(obj.encode(), "base64"))
```

3.4 Executing tasks

```
def execute_fn(task_id: uuid.UUID, ser_fn: str, ser_params: str):
    # deserilaize function and params

    # execute the function..
    result = fn(*args, **kwargs)

    # serialize and return the  results
```