Bruno Felalaga and Jovanay Carter, Distributed Systems, Fall 2024

Push Model Testing Report

This report summarizes the comprehensive testing of the **Push Model** implementation for MPCSFaaS. All 16 tests were executed successfully, demonstrating the robustness, fault tolerance, and scalability of the system. Below is a detailed account of the tests performed, their objectives, and the results.

Testing Objectives

The main objectives of the tests were:

1. Validate **basic functionality** (e.g., function registration and execution).
2. Ensure correct handling of **edge cases** (e.g., invalid inputs, large payloads).
3. Verify **fault tolerance**, including task and worker failure recovery.
4. Measure the system's **scalability and concurrency** under high loads.

Test Results

Basic Functionality Tests

1. **`test_fn_registration_invalid`**:
    - Objective: Validate behavior when a function is registered with an invalid (non-serialized) payload.
    - Result: Passed. The system correctly returned HTTP 400/500, indicating invalid input handling.
2. **`test_fn_registration`**:
    - Objective: Test successful registration of a valid serialized function.
    - Result: Passed. The system returned a `function_id` with HTTP 200/201.
3. **`test_execute_fn`**:
    - Objective: Verify task creation and execution for a registered function.
    - Result: Passed. Task transitioned through `QUEUED`, `RUNNING`, and `COMPLETED` states with correct results.
4. **`test_roundtrip`**:
    - Objective: Ensure that inputs to a registered function are correctly processed and results are accurately returned.
    - Result: Passed. Results matched expected outputs, confirming serialization/deserialization integrity.

Edge Case and Fault Tolerance Tests

5. **`test_memory_limit`**:
    - Objective: Test handling of tasks exceeding memory limits by simulating a `MemoryError`.
    - Result: Passed. Task failed with appropriate `FunctionExecutionFailure`, and error details were captured in Redis.
6. **`test_invalid_function_id`**:
    - Objective: Validate behavior when executing a task with a non-existent function ID.
    - Result: Passed. Returned HTTP 404 for the invalid `function_id`.
7. **`test_invalid_task_id`**:
    - Objective: Test querying results for a non-existent `task_id`.
    - Result: Passed. Returned HTTP 404, confirming robust error handling.
8. **`test_function_error`**:
    - Objective: Verify handling of a function that raises an exception.
    - Result: Passed. Task transitioned to `FAILED`, with the `FunctionExecutionFailure` error type and a detailed traceback.

Scalability and Performance Tests

9. **`test_concurrent_execution`**:
    - Objective: Ensure proper handling of multiple concurrent task executions.
    - Result: Passed. All tasks completed successfully with accurate results.
10. **`test_large_payload`**:
    - Objective: Test system behavior when processing large input/output payloads.
    - Result: Passed. Function processed 100,000 items, confirming efficient handling of large data.
11. **`test_concurrent_tasks_with_delay`**:
    - Objective: Evaluate task handling with varying delays to simulate real-world scenarios.
    - Result: Passed. Results confirmed correct sequencing and concurrency.
12. **`test_cpu_bound_task`**:
    - Objective: Test execution of computationally intensive tasks (e.g., Fibonacci sequence).

    - Result: Passed. Task completed with expected results, demonstrating CPU-intensive task support.
13. **`test_chain_of_tasks`**:
    - Objective: Verify sequential task dependencies, where each task depends on the result of the previous one.
    - Result: Passed. Final output matched expected values, confirming task chaining integrity.
14. **`test_recursion_limit`**:
    - Objective: Test handling of deeply recursive functions that exceed the recursion limit.
    - Result: Passed. Task transitioned to `FAILED` with appropriate `RecursionError` details.

Fault Tolerance Tests

15. **`test_PUSH_worker_failure_manual`**:
    - Objective: Simulate worker failures during task execution and verify task requeuing.
    - Result: Passed. Tasks were correctly requeued, demonstrating recovery from worker failures.
16. **`test_PUSH_worker_failure_automated`**:
    - Objective: Automate worker failure simulation with retries up to the maximum allowed threshold.
    - Result: Passed. Tasks were reattempted across four worker restarts. Final failure state (`FAILED`) was correctly recorded after exceeding `MAX_RETRIES`, with the error type `WorkerFailureError`.

Observations

1. **Robustness**:
    - The system effectively handled all tested scenarios, including invalid inputs, task failures, worker failures, and resource constraints.
    - Fault tolerance mechanisms (e.g., task requeuing and retry thresholds) worked as expected.
2. **Performance**:
    - Concurrent execution tests demonstrated excellent scalability, with tasks completing efficiently under load.

    - The system handled large payloads and CPU-intensive computations without significant delays.
3. **Error Reporting**:
    - Detailed error messages and tracebacks provided clarity for debugging failed tasks.
    - Task statuses were consistently updated in Redis, ensuring reliable state tracking.

Conclusion

The Push Model implementation of MPCSFaaS has demonstrated its robustness, fault tolerance, and scalability through rigorous testing. The successful execution of all 16 tests confirms that the system meets the required functional and non-functional specifications.

This testing process highlights the system's readiness for deployment in distributed environments and its capability to handle real-world workloads effectively. Further optimizations, such as dynamic worker scaling and enhanced logging, can further improve the system's performance and maintainability.

Pull Model Tests

Web Service Testing Report

This report documents the testing outcomes for the **Web Service** model of MPCSFaaS, with a focus on both passed and failed test cases. The testing framework verified various functionalities, including basic operations, error handling, concurrency, and fault tolerance.

Testing Overview

Total Tests: 15

Tests Passed: 12

Tests Failed: 3

The failed tests were related to **task concurrency, CPU-bound operations, and task chaining**, which are detailed below.

Passed Tests

Basic Functionality Tests

1. **`test_fn_registration_invalid`**:
    - **Objective**: Test function registration with invalid payloads.
    - **Result**: Passed. The system returned HTTP 400/500 as expected.
2. **`test_fn_registration`**:
    - **Objective**: Verify successful function registration with serialized payloads.
    - **Result**: Passed. Returned valid `function_id`.
3. **`test_execute_fn`**:
    - **Objective**: Confirm task creation and execution with valid inputs.
    - **Result**: Passed. Tasks transitioned to `QUEUED` state and executed as expected.
4. **`test_large_payload`**:
    - **Objective**: Test handling of functions with large input/output data.
    - **Result**: Passed. The system successfully processed large payloads.
5. **`test_recursion_limit`**:
    - **Objective**: Validate behavior for recursive functions exceeding stack depth.
    - **Result**: Passed. The system returned appropriate `RecursionError` details.

Fault Tolerance Tests

6. **`test_worker_failure_manual`**:
    - **Objective**: Simulate manual worker failures and verify task requeuing.
    - **Result**: Passed. Tasks were requeued successfully after worker failures.
7. **`test_pull_worker_failure`**:
    - **Objective**: Automate worker failure handling for pull workers.
    - **Result**: Passed. Worker failures were detected, and tasks were appropriately marked as `FAILED`.

Failed Tests

1. **`test_concurrent_tasks_with_delay`**
    - **Objective**: Test concurrent execution of tasks with varying delays.
    - **Expected**: All tasks should complete with correct results.
    - **Outcome**: Failed due to incomplete task results.
    - **Error**: `AssertionError: Not all tasks completed`.
        - **Details**:
            - `len(results) = 0`, but `len(delays) = 3`.

        - The system failed to retrieve results for any tasks, indicating issues with concurrency or task result retrieval.
2. **`test_cpu_bound_task`**

    - **Objective**: Validate execution of CPU-intensive tasks (e.g., Fibonacci sequence).
    - **Expected**: Task result should match the correct Fibonacci(10) output (55).
    - **Outcome**: Failed due to missing result data.
    - **Error**: `AssertionError: None == 55`.
        - **Details**: Task status remained incomplete, likely due to insufficient worker resources or execution timeout.
3. **`test_chain_of_tasks`**

    - **Objective**: Verify execution of tasks in sequence, where outputs from one task are inputs to the next.
    - **Expected**: Final result should match calculated value `400` for the given chain: `((5 * 2) + 10) ** 2`.
    - **Outcome**: Failed due to incomplete execution.
    - **Error**: `AssertionError: 5 == 400`.
        - **Details**: Execution halted at the first step of the chain, suggesting an issue with intermediate task result propagation.

Root Cause Analysis

Test Failures

1. **Concurrency Issues**:

    - The `test_concurrent_tasks_with_delay` failure indicates problems with handling simultaneous task execution.
    - Likely causes:
        - Insufficient worker availability.
        - Synchronization or queue management errors.
2. **Resource Allocation**:

    - The `test_cpu_bound_task` failure suggests resource bottlenecks for CPU-intensive operations.
    - Workers may lack sufficient processing time or priority for computationally expensive tasks.
3. **Intermediate Result Handling**:

    - The `test_chain_of_tasks` failure points to a breakdown in passing results between chained tasks.
    - Possible issues:

    - Serialization/deserialization inconsistencies.
    - Task execution interruptions.

Recommendations

1. **Concurrency Improvements**:
    - Increase the number of available workers to handle concurrent tasks.
    - Optimize worker task scheduling to ensure timely execution.
2. **Resource Scaling**:
    - Implement dynamic resource scaling for CPU-bound tasks to prevent timeouts.
    - Introduce task prioritization for computationally intensive workloads.
3. **Chained Task Debugging**:
    - Enhance logging for task chaining to trace intermediate results.
    - Verify serialization/deserialization logic and fix potential state management bugs.
4. **Testing Environment**:
    - Test with a higher number of workers to mimic production-like scenarios.
    - Use profiling tools to identify bottlenecks in task scheduling and execution.

Conclusion

While the **Web Service** model passed 12 out of 15 tests, the failures in concurrency, CPU-bound task handling, and chaining highlight areas for improvement in scalability and reliability. Addressing these issues will ensure robust handling of real-world workloads and better fault tolerance. Further debugging and optimization are recommended to achieve full compliance.

Local Model Testing Report

This report summarizes the testing of the Local Model implementation of MPCSFaaS. The tests focused on validating the correctness of single-task execution and concurrency handling. All tests were successfully executed, demonstrating the accuracy and performance of the Local Model in a single-node environment.

Testing Objectives

1. Validate the correctness of task execution and result storage in Redis.
2. Ensure that multiple tasks are executed concurrently to maximize CPU utilization.

Test Results

Single-Task Execution Tests

1. **Single Task Execution**
    - **Objective**: Verify that a single task executes correctly and its result is stored in Redis.
    - **Test**: `test_local_dispatcher_single_task`
    - **Result**: Passed. The `add(1, 2)` task completed successfully, and the result (3) was stored in Redis with the correct status.

Concurrency Tests

2. **Concurrent Task Execution**
    - **Objective**: Validate that multiple tasks execute concurrently, reducing the total execution time.
    - **Test**: `test_local_dispatcher_with_sleep_concurrency`
    - **Result**: Passed. Four `sleep_test(2)` tasks completed within 2.5 seconds, demonstrating concurrency.

Observations

- **Accuracy**:
  The Local Model successfully executed tasks with correct results, as verified against expected outputs.
- **Concurrency**:
  The dispatcher effectively leveraged multiprocessing to execute multiple tasks in parallel, ensuring efficient CPU utilization.
- **Reliability**:
  Redis provided consistent and accurate state management, with all task statuses and results correctly stored and retrievable.

Conclusion

The Local Model implementation of MPCSFaaS demonstrated its accuracy and performance through rigorous testing. Its ability to handle concurrent tasks efficiently makes it a reliable baseline for comparison with distributed models.

Future improvements could focus on dynamic worker pool sizing and fault tolerance to enhance robustness in more demanding scenarios.