Bruno Felalaga and Jovanay Carter, Distributed Systems, Fall 2024

Technical Report

Modern distributed systems often rely on serverless computing to execute user-defined functions efficiently and at scale. The MPCSFaaS framework implements three key models—Push, Pull, and Local—each tailored for different use cases:

The Push Model assigns tasks to workers as they become available, optimizing throughput and minimizing idle time. The Pull Model allows workers to request tasks when ready, ensuring flexible load distribution. The Local Model provides a simple baseline, executing tasks on a local multiprocessing pool for development and testing.

This report examines the design, fault tolerance mechanisms, and performance characteristics of these models.

REST API Service -- across all 3 models

The REST API service serves as the gateway for user interactions with MPCSFaaS. Endpoints include:

1. Registering functions: Users can register Python functions, which are serialized and stored in Redis along with metadata such as the function name and a unique UUID.
2. Executing tasks: Users provide the UUID of a registered function along with input parameters. The service creates a unique task ID, stores it in Redis, and publishes the task to a Redis channel.
3. Querying task status: Users can check the lifecycle stage of a task (e.g., QUEUED, RUNNING, COMPLETED, FAILED).
4. Fetching results: Once a task is completed, users can retrieve its serialized result or error details.

Redis, an in-memory database, ensures low-latency access to function and task metadata, making the REST API service highly responsive.

Push Model Architecture

Task Dispatcher

The Task Dispatcher acts as the system's brain, bridging the REST API and the workers. Its primary responsibilities include:

- Worker Management: It maintains a registry of available workers, tracking their capacity and health via a heartbeat mechanism.
- Task Assignment: Tasks are dispatched to workers based on availability and load balancing strategies.
- Result Handling: Once a worker completes a task, the dispatcher updates Redis with the task's final status and result.

The dispatcher uses ZMQ's DEALER/ROUTER pattern to establish asynchronous communication with workers, enabling non-blocking task assignment and result collection.

Push Workers

Push Workers execute tasks assigned by the dispatcher. Each worker:

1. Registers itself with the dispatcher upon initialization, declaring its capacity.
2. Listens for incoming tasks from the dispatcher.
3. Executes tasks asynchronously using Python's multiprocessing module.
4. Sends results or errors back to the dispatcher upon task completion.
5. Sends periodic heartbeats to indicate its availability.

This architecture ensures that tasks are executed efficiently, even in scenarios involving multiple workers or computationally intensive functions.

The lifecycle of a task in the Push Model consists of four distinct stages:

1. QUEUED: A task is created and stored in Redis.
2. RUNNING: The task is assigned to a worker and is in execution.
3. COMPLETED: The task is executed successfully, and the result is stored in Redis.
4. FAILED: The task encounters an error during execution or is lost due to a worker failure.

Push Model Implementation Highlights

Communication Protocols

The Push Model relies heavily on ZMQ for communication between the dispatcher and workers. The DEALER/ROUTER pattern ensures efficient task assignment while enabling bidirectional communication. Redis acts as both a state manager and a message broker, facilitating task publishing and retrieval.

Fault Tolerance Mechanisms

Fault tolerance is critical in distributed systems, where failures are inevitable. The Push Model addresses two primary failure scenarios:

1. Task Failures:
    - If a function raises an exception during execution, the error is serialized and reported back to the user.
    - Common errors, such as serialization issues or runtime exceptions, are encapsulated in custom error types for detailed diagnostics.
2. Worker Failures:
    - Workers send heartbeats every 0.5 seconds to signal their health.
    - If a worker misses three consecutive heartbeats, it is marked as dead, and its tasks are requeued or marked as FAILED if retries exceed a configurable limit.
    - This mechanism ensures the system remains operational even in the face of worker crashes.

Concurrency and Load Balancing

Workers execute tasks using Python's multiprocessing.Pool, allowing them to handle multiple tasks concurrently. The dispatcher uses a round-robin strategy to distribute tasks evenly across workers, minimizing idle time and maximizing throughput.

Testing and Validation

Unit Testing

Comprehensive unit tests were conducted to validate:

- Function registration, execution, and result retrieval through the REST API.
- Serialization and deserialization of Python objects.
- Error handling for invalid inputs and unexpected failures.

Integration Testing

End-to-end tests simulated real-world workflows, including:

- Task creation and assignment.
- Worker failures during execution.
- Requeuing of tasks for fault recovery.

Fault Tolerance Testing

Scenarios tested included:

1. Functions raising runtime exceptions.
2. Workers crashing mid-execution.
3. Prolonged worker unavailability due to heartbeat failures.

Performance Testing

A weak scaling study was conducted to evaluate the system's scalability. Tasks were increased proportionally with the number of workers, measuring metrics such as throughput and latency.

Performance Evaluation

Experimental Setup

1. Metrics:
    - Throughput: Tasks completed per second.
    - Latency: Time from task creation to result retrieval.
    - Fault Recovery Time: Time taken to reassign tasks from failed workers.
2. Workload:
    - Mix of lightweight ("no-op") tasks and heavyweight (e.g., sleep) tasks.
    - Worker pool sizes ranging from 1 to 16.

Results

1. Throughput:
    - Linear scaling observed, with throughput reaching 500 tasks/sec at 16 workers.
2. Latency:
    - Average latency: 10 ms for lightweight tasks.
    - Slight overhead compared to local execution due to communication and serialization costs.
3. Fault Recovery:
    - Tasks reassigned within 2.5 seconds of detecting worker failures.

Observations

- The Push Model excels in high-throughput scenarios, efficiently utilizing all available workers.
- Fault recovery mechanisms are robust, though dispatcher resilience remains a potential bottleneck.

Limitations

1. Dispatcher as a Single Point of Failure:
    - If the dispatcher crashes, the entire system halts.
2. Static Load Balancing:
    - Current round-robin strategy does not account for worker performance variations.
3. Latency Overhead:
    - ZMQ communication adds minor overhead compared to local execution.

Future Improvements

1. High Availability:
    - Introduce dispatcher redundancy using consensus algorithms like Raft.
2. Dynamic Scaling:
    - Add or remove workers based on task queue length and worker utilization.
3. Advanced Load Balancing:
    - Incorporate worker health and historical performance metrics for smarter task allocation.
4. Task Prioritization:
    - Allow users to specify task priorities for critical workloads.

Pull Model Implementation Highlights

Pull Model Implementation Highlights

Task Dispatcher

The Pull Model dispatcher is designed to provide tasks only when workers explicitly request them, enabling dynamic worker scalability and resource efficiency. The primary responsibilities of the Pull Model dispatcher include:

- Worker Communication: Workers initiate communication by requesting tasks using ZeroMQ's REQ/REP pattern. The dispatcher maintains a responsive task queue to handle these requests efficiently.
- Task Queue Management: The dispatcher listens to task submissions via Redis channels and maintains a queue of available tasks.
- Result Handling: Upon task completion, workers report the results back to the dispatcher, which updates Redis with the task's final status and result.

This request-driven mechanism ensures that tasks are distributed only to available workers, reducing potential bottlenecks and idle worker states.

Pull Workers

Pull Workers play a proactive role in requesting and executing tasks. Their key functions include:

- Task Requests: Workers repeatedly poll the dispatcher for tasks when idle.
- Task Execution: Upon receiving a task, the worker executes it using Python's multiprocessing module.

- Result Reporting: Workers send the task's result or error back to the dispatcher once execution completes.

This design ensures workers maintain control over when they receive tasks, making the system well-suited for environments where worker availability fluctuates or where workload demands are unpredictable.

Task Lifecycle in the Pull Model

The Pull Model follows a structured lifecycle for task management:

1. QUEUED: A task is created and stored in Redis.
2. REQUESTED: A worker requests the task from the dispatcher.
3. RUNNING: The task is being executed by a worker.
4. COMPLETED/FAILED: The task finishes execution, and its status is updated in Redis.

Pull Model Advantages

- Dynamic Load Balancing: By allowing workers to request tasks as needed, the Pull Model naturally adapts to changes in worker availability.
- Reduced Task Overhead: Idle workers request tasks only when ready, minimizing unnecessary communication.
- Fault Tolerance: If a worker fails mid-execution, its task remains in Redis for reassignment or recovery.

Pull Model Testing and Validation

- Unit Testing: Verified task request handling, worker communication, and Redis task lifecycle updates.
- Integration Testing: Simulated scenarios with varying worker availabilities to ensure smooth task assignment and recovery.
- Fault Recovery Testing: Confirmed tasks were requeued or reassigned after worker failures.

The Pull Model's flexibility and responsiveness make it ideal for scenarios requiring on-demand task handling with unpredictable workloads.

Local Model Implementation Highlights

Local Model Implementation Highlights

Task Dispatcher

The Local Model dispatcher processes tasks on a single machine using Python's multiprocessing library. Tasks are executed concurrently, and their statuses and results are stored in Redis for easy tracking. This model emphasizes simplicity and low-latency execution without requiring inter-node communication.

Task Lifecycle

1. QUEUED: Tasks are added to the queue and await execution.
2. RUNNING: Tasks are picked up by worker processes for execution.
3. COMPLETED/FAILED: Results or errors are recorded in Redis upon completion.

Advantages

- Simple and efficient, avoiding communication overhead.
- Provides a robust baseline for comparing distributed models.

Testing and Validation

- Tasks were executed with diverse inputs, validating correctness and fault handling.
- Redis entries for task results were verified after execution.

The Local Model offers high performance and simplicity, making it ideal for testing and development while highlighting the need for distributed solutions in larger-scale environments.