Distributed FasS System

Introduction:

In the realm of distributed systems, our Faas (Function as a Service) system stands out as an intricate and resilient architecture designed to streamline task execution and processing. Comprising a client, server, task dispatcher, push workers, and pull workers, our system is orchestrated to seamlessly integrate various components for enhanced efficiency, fault tolerance, and adaptability.

Client:

Our client facilitates communication with the server through RESTful API calls. When defining a desired function in the client, it can be seamlessly registered with our server. The client offers serialization capabilities for the function, aiding in its efficient transfer to the server. Furthermore, it provides deserialization for any results received from the server. Upon registration, the server assigns a unique function ID for future reference.

Once the function ID is obtained, the client has the capability to execute the specific function with varying parameters, acquiring a corresponding task ID. Subsequent API calls allow for checking the status and retrieving the result of this task using its unique identifier. This comprehensive set of functionalities, all managed through API calls, enables a streamlined and efficient interaction between the client and the server, providing a seamless experience for function registration, execution, and result retrieval.

Server:

Our server is implemented using FastAPI to expose RESTful APIs. It seamlessly integrates with the distributed key-value storage system, Redis, for function registration, handling function execution requests, and maintaining task status records. Leveraging the Redis PUB/SUB system, we publish tasks to a specific topic, allowing the task dispatcher to efficiently receive task execution requests.

To enhance the fault tolerance of our system against processing failures, we adopt a robust strategy. A duplicate set of executed task IDs is stored in Redis. Upon receiving the results, we update the system and promptly delete the corresponding IDs. This approach ensures the reliability and resilience of our Faas (Function as a Service) system in the face of potential processing challenges since we can recover from this set of unfinished tasks.

Task Dispatcher:

Our task dispatcher is a versatile system designed to accommodate different worker modes, providing users with the flexibility to choose between local, push, and pull worker modes. Each mode entails specific actions tailored to optimize task distribution. The task dispatcher seamlessly communicates with remote push or pull workers using ZMQ in a non-blocking manner, facilitating the efficient exchange of tasks and results. Moreover, the task dispatcher can function as a local worker, enabling the processing of tasks locally for enhanced performance.

The task dispatcher operates with a communicator, a task queue, and a result queue, all managed by dedicated processes. In this configuration, tasks are received from Redis and results are sent back to Redis in parallel, contributing to an overall improvement in system performance.

To bolster fault tolerance, we have incorporated a task recovery mechanism. This feature enables the task dispatcher to retrieve and resume unfinished tasks in the event of a system failure. The mechanism involves persistently storing copies of task IDs in Redis until results are updated. Upon recovery, the task dispatcher checks Redis for remaining task IDs and resends those tasks to workers, ensuring the continuity of task processing.

Additionally, our task dispatcher automatically registers all workers and implements load balancing based on worker status. This dynamic approach ensures optimal task distribution, preventing bottlenecks and enhancing the overall efficiency of the system. The task dispatcher acts as a central orchestrator, intelligently managing the workload and maintaining a resilient framework for task execution.

Push worker:

Our push workers operate seamlessly within ZMQ's DEALER/ROUTER pattern, a design choice made for optimal efficiency in handling workloads received from the task dispatcher. These workers are meticulously crafted with a focus on efficiency, load balancing, and fault tolerance to ensure a resilient and high-performance task processing system.

To streamline integration, the task dispatcher takes a proactive approach by automatically registering running workers. This is achieved by monitoring messages received by the ROUTER, enabling the establishment of a dynamic worker pool for effective task distribution. Each push worker is configurable to process tasks in parallel, ensuring flexibility in handling varying workloads.

Adding a layer of intelligence to the task assignment process, each worker communicates its current workload, including task processing and the number of active processes, to the task dispatcher. Armed with this information, the task dispatcher can intelligently assign the next task to the worker with the minimum workload, optimizing overall task processing efficiency and maintaining a balanced distribution.

Within this dynamic framework, a push worker manager embedded in the task dispatcher constantly monitors worker health through heartbeat messages. If a worker fails to transmit a heartbeat within a specified timeframe, the worker is gracefully removed from the pool. Additionally, the task dispatcher keeps meticulous track of tasks assigned to each worker, facilitating the identification and rerunning of failed tasks in the event of a worker's failure.

The communication protocol between the task dispatcher and push workers is deliberately designed as non-blocking, eliminating the need to wait for tasks or results. This non-blocking communication strategy significantly enhances the overall responsiveness and efficiency of the system, ensuring that tasks are processed seamlessly without unnecessary delays. The push workers, operating within this well-thought-out framework, contribute to a robust and adaptive task-dispatching system.

Pull workers:

The pull worker is meticulously designed to efficiently interact with the task dispatcher, incorporating key features for seamless communication and task handling. Employing the Lazy-Pirate Pattern, the pull worker ensures reliable communication by implementing a robust mechanism for detecting and handling network failures. This pattern allows the pull worker to automatically reconnect to the task dispatcher in case of a connection loss, enhancing the overall resilience of the system.

One of the distinctive features of the pull worker is its proactive approach to task acquisition. Rather than relying on constant polling, the pull worker adopts a Poll-Dispatcher-on-Interval strategy. This design choice strikes a balance between responsiveness and resource efficiency. By periodically polling the task dispatcher for more work, the pull worker minimizes unnecessary network traffic and resource consumption while remaining responsive to new tasks. This approach enhances the pull worker's ability to adapt to varying workloads and ensures optimal resource utilization.

In terms of communication protocol, the pull worker and the task dispatcher engage in a well-defined interaction using REQ (Request) and REP (Reply) sockets. The workers employ REQ sockets to submit requests for tasks, while the dispatcher responds using REP sockets. This bidirectional communication occurs in a loop, forming a continuous and efficient exchange between the pull worker and the task dispatcher. This design not only

simplifies the communication model but also ensures that each task is processed in a coordinated and orderly fashion, contributing to the overall reliability and predictability of the system. The pull worker's design reflects a thoughtful combination of communication patterns and strategies, resulting in a resilient and responsive component within the larger task-dispatching system.