Distributed HTTP Server Implementation

Marco Rodríguez Vargas, Gerald Vargas Vásquez

***Abstract*—This paper presents the design and extension of a custom HTTP server originally developed using raw TCP sockets in Go. The extended system introduces distributed processing capabilities by orchestrating multiple instances of the same server deployed as isolated Docker containers. A centralized dispatcher coordinates the distribution of computationally intensive tasks, monitors worker availability via periodic health checks, and dynamically reallocates work upon failure detection. The solution demonstrates concurrency through parallel execution, robustness through fault-tolerant mechanisms, and scalability through containerized deployment. This project showcases the practical application of distributed systems concepts in a modular, concurrent, and resilient architecture.**

***Index Terms*—HTTP**

I. Architecture of the Distributed System

THE distributed HTTP server system implements a modular architecture composed of a centralized Dispatcher and multiple Worker instances, all deployed as independent Docker containers. This design enforces a clear separation of responsibilities: the Dispatcher handles task coordination, while the Workers are in charge of execution. Each Worker runs an autonomous replica of the original HTTP server developed in Project 1, enabling it to process tasks independently and concurrently.

The Dispatcher serves as the single entry point for incoming HTTP requests and is responsible for distributing them among available Workers. For specific endpoints that allow parallelization such as the Monte Carlo estimation of Pi and Count Words in a large file, requests are divided into smaller subtasks and dispatched concurrently to multiple Workers. The distribution strategy can follow a round-robin approach or adapt based on each Worker's current load.

Beyond task delegation, the Dispatcher continuously monitors the health of each Worker by sending periodic /ping requests. If a Worker fails to respond, it is marked as inactive and excluded from the routing pool. Once a previously failed Worker becomes responsive again, it is automatically reintegrated without requiring manual intervention. The /workers endpoint, exposed by the Dispatcher, provides real-time information on each Worker's status, including whether it is active, how many tasks it has completed, and its current load. This facilitates runtime observability, fault detection, and system behavior analysis.

The entire system is containerized using Docker ensuring portability. With Docker Compose, the complete environment can be launched effortlessly, allowing for horizontal scalability through Worker replication and seamless component communication. This distributed architecture is designed to be scalable and fault-tolerant making it well suited for concurrent intensive workloads.

M. Rodríguez Vargas and G. Vargas Vásquez are with the Department of Computing Engineering, Instituto Tecnológico de Costa Rica, Cartago, Costa Rica.

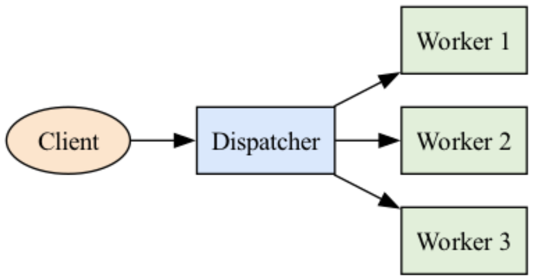


Fig. 1. Architecture

II. Communication Protocols

Communication between dispatcher and workers is handled using plain HTTP/1.0 over raw TCP sockets. The dispatcher constructs and forwards HTTP requests to worker addresses defined via environment variables. All HTTP commands such as /fibonacci, /simulate, /hash, or parallel workloads like /montecarlo and /countwords are routed dynamically. The dispatcher also implements periodic /ping checks to assess worker availability and update their status. This protocol ensures both stateless communication and easy recovery from worker failures.

III. Fault Tolerance and Scalability Analysis

The system incorporates fault tolerance by continuously monitoring the health of each Worker through periodic /ping requests. If a Worker fails to respond it is automatically marked as inactive and excluded from the routing process. This ensures that tasks are not lost or delayed due to unresponsive instances. Moreover, if a task is assigned to a failed Worker, it is retried and reassigned to an active one, maintaining the system's reliability without manual intervention.

Scalability is achieved through a container-based replication model. By simply increasing the number of Worker replicas using Docker Compose, the system can handle a larger volume of concurrent requests. This scaling process does not require any modification to the existing business logic or routing mechanisms, making it flexible. As demand grows, the architecture supports horizontal scaling seamlessly ensuring consistent performance under varying workloads.

IV. Health check

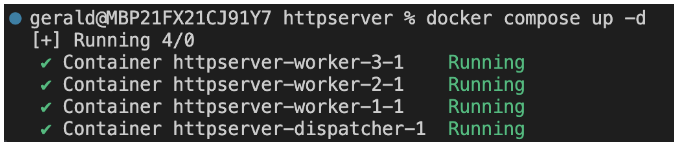
The health check mechanism continuously monitors the availability of registered workers by periodically sending ping requests to each one. Implemented as a goroutine in the dispatcher, this process ensures that unresponsive workers are automatically marked as inactive and removed from the task distribution pool. If a previously inactive worker becomes responsive again, it is reactivated. This dynamic health monitoring improves fault tolerance and helps maintain a reliable and efficient distributed processing system.

V. Results

A. Docker Containers

All service instances were successfully initialized using their respective Docker images. Below is a screenshot showing the result of executing the following commands, which build and launch the entire distributed system:

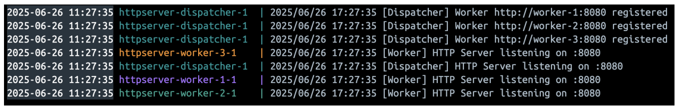
```
docker compose build
docker compose up -d
```



```
● gerald@MBP21FX21CJ91Y7 httpserver % docker compose up -d
 [+] Running 4/0
  ✔ Container httpserver-worker-3-1    Running
  ✔ Container httpserver-worker-2-1    Running
  ✔ Container httpserver-worker-1-1    Running
  ✔ Container httpserver-dispatcher-1  Running
```

Fig. 2. Docker compose

This confirms that the container images for both the dispatcher and the workers were correctly built and deployed, and that the services are running as expected.



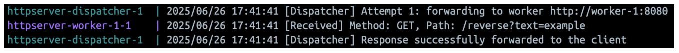
```
2025-06-26 11:27:35 httpserver-dispatcher-1  | 2025/06/26 17:27:35 [Dispatcher] Worker http://worker-1:8080 registered
2025-06-26 11:27:35 httpserver-dispatcher-1  | 2025/06/26 17:27:35 [Dispatcher] Worker http://worker-2:8080 registered
2025-06-26 11:27:35 httpserver-dispatcher-1  | 2025/06/26 17:27:35 [Dispatcher] Worker http://worker-3:8080 registered
2025-06-26 11:27:35 httpserver-worker-3-1    | 2025/06/26 17:27:35 [Worker] HTTP Server listening on :8080
2025-06-26 11:27:35 httpserver-dispatcher-1  | 2025/06/26 17:27:35 [Dispatcher] HTTP Server listening on :8080
2025-06-26 11:27:35 httpserver-worker-1-1    | 2025/06/26 17:27:35 [Worker] HTTP Server listening on :8080
2025-06-26 11:27:35 httpserver-worker-2-1    | 2025/06/26 17:27:35 [Worker] HTTP Server listening on :8080
```

Fig. 3. Docker logs

In the logs above, we can also observe that the workers were initialized and successfully registered with the dispatcher and they began listening for requests on their designated ports.

B. Distributed Execution

The application successfully distributes the workload among the workers and functions as an intermediary between the client and the workers. For non-parallelizable tasks, it simply forwards the request to any available active worker, waits for the response, and then relays it back to the client. For example, when executing the request **/reverse?text=example**, the following logs are produced:



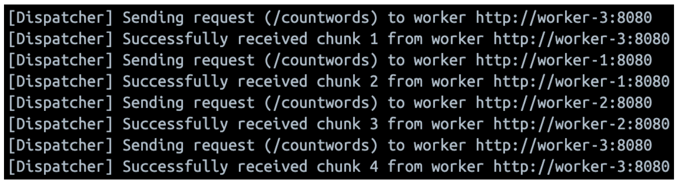
```
httpserver-dispatcher-1  | 2025/06/26 17:41:41 [Dispatcher] Attempt 1: forwarding to worker http://worker-1:8080
httpserver-worker-1-1    | 2025/06/26 17:41:41 [Received] Method: GET, Path: /reverse?text=example
httpserver-dispatcher-1  | 2025/06/26 17:41:41 [Dispatcher] Response successfully forwarded to the client
```

Fig. 4. Request forwarded

Here we can observe that the dispatcher received the request, forwarded it to **worker-1**, which processed the request and sent the response back to the dispatcher. The dispatcher then relayed the response to the client.

For parallelizable problems, the process is slightly more complex. In this case, the dispatcher relies on a handler that splits the workload, checks for available workers, and sends smaller chunks of work to them. Once all individual responses are received, the dispatcher combines them into a single response.

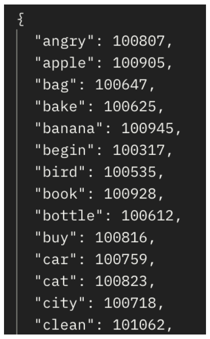
1) Word Count: For this problem, a 50MB text file was used, configured to be split into 15MB chunks. The logs below show the results of this execution.



```
[Dispatcher] Sending request (/countwords) to worker http://worker-3:8080
[Dispatcher] Successfully received chunk 1 from worker http://worker-3:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-1:8080
[Dispatcher] Successfully received chunk 2 from worker http://worker-1:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-2:8080
[Dispatcher] Successfully received chunk 3 from worker http://worker-2:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-3:8080
[Dispatcher] Successfully received chunk 4 from worker http://worker-3:8080
```

Fig. 5. Count words logs

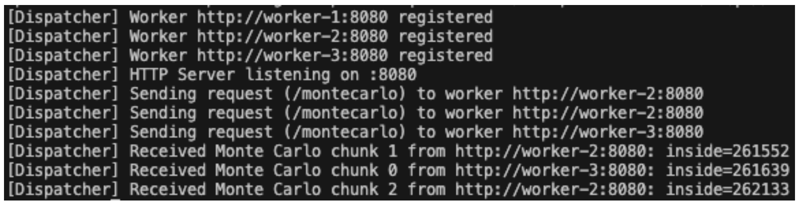
Here we can determine that the dispatcher is forwarding the work to different workers to solve the problem. As a result of this execution, we receive a response containing a list of words and how many times each appeared in the file.



```
{
  "angry": 100807,
  "apple": 100905,
  "bag": 100647,
  "bake": 100625,
  "banana": 100945,
  "begin": 100317,
  "bird": 100535,
  "book": 100928,
  "bottle": 100612,
  "buy": 100816,
  "car": 100759,
  "cat": 100823,
  "city": 100718,
  "clean": 101062,
```

Fig. 6. Count words result

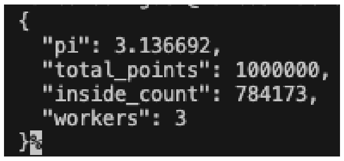
2) Monte Carlo: For this problem, a simulation was configured to estimate the value of π using the Monte Carlo method. The dispatcher split the total number of points into smaller chunks and distributed them across the available workers. The logs below show the results of this execution.



```
[Dispatcher] Worker http://worker-1:8080 registered
[Dispatcher] Worker http://worker-2:8080 registered
[Dispatcher] Worker http://worker-3:8080 registered
[Dispatcher] HTTP Server listening on :8080
[Dispatcher] Sending request (/montecarlo) to worker http://worker-2:8080
[Dispatcher] Sending request (/montecarlo) to worker http://worker-2:8080
[Dispatcher] Sending request (/montecarlo) to worker http://worker-3:8080
[Dispatcher] Received Monte Carlo chunk 1 from http://worker-2:8080: inside=261552
[Dispatcher] Received Monte Carlo chunk 0 from http://worker-3:8080: inside=261639
[Dispatcher] Received Monte Carlo chunk 2 from http://worker-2:8080: inside=262133
```

Fig. 7. Monte Carlo logs

Here we can determine that the dispatcher is forwarding the work to different workers to solve the problem. As a result of this execution, we receive a response containing the approximation of π, along with the number of points that fell inside the unit circle and the number of workers that participated in the computation.

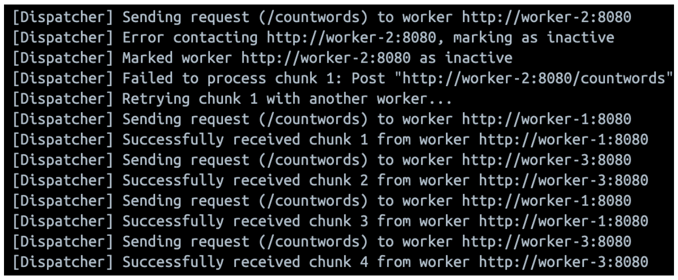


```
{
  "pi": 3.136692,
  "total_points": 1000000,
  "inside_count": 784173,
  "workers": 3
}
```

Fig. 8. Monte Carlo logs

C. Fault Tolerance

To ensure the fault tolerance of the distributed system, we can run a scenario where, in the middle of execution, we turn off one of the workers. Below, you can find a screenshot of the logs captured during this test.



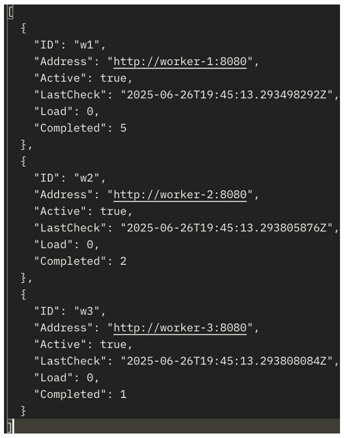
```
[Dispatcher] Sending request (/countwords) to worker http://worker-2:8080
[Dispatcher] Error contacting http://worker-2:8080, marking as inactive
[Dispatcher] Marked worker http://worker-2:8080 as inactive
[Dispatcher] Failed to process chunk 1: Post "http://worker-2:8080/countwords"
[Dispatcher] Retrying chunk 1 with another worker...
[Dispatcher] Sending request (/countwords) to worker http://worker-1:8080
[Dispatcher] Successfully received chunk 1 from worker http://worker-1:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-3:8080
[Dispatcher] Successfully received chunk 2 from worker http://worker-3:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-1:8080
[Dispatcher] Successfully received chunk 3 from worker http://worker-1:8080
[Dispatcher] Sending request (/countwords) to worker http://worker-3:8080
[Dispatcher] Successfully received chunk 4 from worker http://worker-3:8080
```

Fig. 9. Fault tolerance logs

When the dispatcher attempted to send a request to **worker-2**, it encountered a DNS error indicating that the worker was no longer available, due to being shut down during execution. The dispatcher correctly marked **worker-2** as inactive and immediately retried the failed chunk with **worker-1**, which successfully processed it. The remaining chunks were then distributed between **worker-1** and **worker-3**, all completing without issues. This confirms that the system can detect worker failures and dynamically reassign tasks to ensure successful completion of the overall job.

D. Monitoring

The endpoint **/monitoring** it's available to check the status of all the workers, for example this is the response after running the word count endpoint two times:



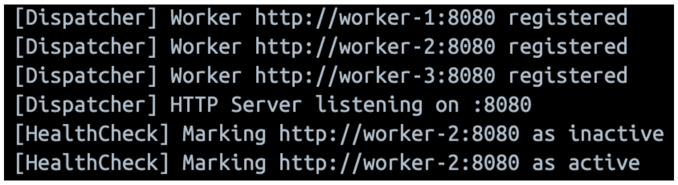
```
[
  {
    "ID": "w1",
    "Address": "http://worker-1:8080",
    "Active": true,
    "LastCheck": "2025-06-26T19:45:13.293498292Z",
    "Load": 0,
    "Completed": 5
  },
  {
    "ID": "w2",
    "Address": "http://worker-2:8080",
    "Active": true,
    "LastCheck": "2025-06-26T19:45:13.293805876Z",
    "Load": 0,
    "Completed": 2
  },
  {
    "ID": "w3",
    "Address": "http://worker-3:8080",
    "Active": true,
    "LastCheck": "2025-06-26T19:45:13.293808084Z",
    "Load": 0,
    "Completed": 1
  }
]
```

Fig. 10. Monitoring

This snapshot shows that all three workers are currently active with no load, and have successfully completed a total of 8 tasks, **worker-1** handled the most (5), followed by **worker-2** (2), and **worker-3** (1).

E. Health check

The logs show that all three workers were registered and the dispatcher started correctly. Shortly after, the health check detected that worker-2 became unresponsive and marked it as inactive. It later recovered and was marked active again, confirming the health check is functioning as expected.



```
[Dispatcher] Worker http://worker-1:8080 registered
[Dispatcher] Worker http://worker-2:8080 registered
[Dispatcher] Worker http://worker-3:8080 registered
[Dispatcher] HTTP Server listening on :8080
[HealthCheck] Marking http://worker-2:8080 as inactive
[HealthCheck] Marking http://worker-2:8080 as active
```

Fig. 11. Health Check

VI. Conclusions

This project extended the functionality of the original HTTP server by transforming it into a distributed system capable of handling parallel workloads with built in scalability and fault tolerance. Through the integration of a centralized dispatcher and multiple worker containers, the system became capable of distributing tasks dynamically while maintaining responsiveness. Moreover, deploying and orchestrating the system using Docker and Docker Compose offered valuable insights into modern deployment strategies. These concepts are foundational in current cloud computing platforms such as Azure and AWS where containerization, service replication and fault detection mechanisms support RESTful microservices architectures. Tools like Kubernetes are often used to orchestrate these services at scale.

Overall this implementation served as a practical exploration of distributed systems and their relationship to real-world technologies. It provided hands-on experience in designing fault tolerant and scalable services.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.