Distributed Array Library with Fault Tolerance

Student:
Arbués Pérez V.
National University of Engineering
Faculty of Sciences
e-mail: arbues.perez@uni.pe

Student:
Sergio Pezo J.
National University of Engineering
Faculty of Sciences
e-mail: sergio.pezo@uni.pe

Student:
André Pacheco T.
National University of Engineering
Faculty of Sciences
e-mail: andre.pacheco@uni.pe

July 10, 2025

Abstract

This document presents the design, development, and implementation of a distributed library for processing large numerical arrays. The system is built upon a master-worker architecture and is implemented in both Java and Python, utilizing only native TCP sockets for inter-process communication, without relying on external messaging frameworks. Key features include automatic data segmentation, parallel processing of mathematical operations on worker nodes, and a robust fault tolerance mechanism based on heartbeats, data replication, and automatic replica promotion. The project also demonstrates interoperability between components written in different languages, showcasing the flexibility of a well-defined, language-agnostic communication protocol.

Keywords: Distributed Systems, Fault Tolerance, TCP Sockets, Parallel Processing, Data Replication, Interoperability.

Contents

1 General Introduction

Distributed systems are fundamental to modern computing, providing the scalability and resilience required to handle large-scale data processing tasks. This project focuses on the core principles of distributed systems by building a library from the ground up for managing and processing large numerical arrays across multiple nodes. The primary goal is to create a system that is both high-performing and fault-tolerant, using fundamental technologies rather than high-level frameworks.

The developed system implements a master-worker architecture where a central master node coordinates a cluster of worker nodes. Large arrays provided by a client are automatically segmented and distributed among the workers. Each worker then leverages multi-core processors to perform parallel computations on its assigned data segment. This approach allows for horizontal scalability, where adding more worker nodes directly increases the system's processing capacity.

A key aspect of this project is the implementation of a robust fault tolerance system. Using a heartbeat mechanism, the master node monitors the health of each worker. If a worker fails, the system automatically promotes a replica of the lost data on another worker, ensuring no data loss and maintaining service continuity. The entire project is implemented in both Java and Python to demonstrate that the principles and the communication protocol are language-agnostic, even allowing for a mixed-language cluster.

2 Objectives

2.1 General Objective

To design and develop a robust, scalable, and fault-tolerant distributed library for processing large numerical arrays, using fundamental distributed computing concepts and native TCP socket communication.

2.2 Specific Objectives

- Implement a master-worker architecture capable of managing a cluster of processing nodes.
- Develop the complete system in two separate, language-specific implementations (Java and Python) to ensure protocol integrity.
- Design and implement a custom, JSON-based communication protocol over native TCP sockets.
- Implement automatic data segmentation and distribution of large arrays from a client to the worker nodes.
- Utilize multi-threading on worker nodes to achieve parallel processing of data segments.
- Implement a fault tolerance mechanism with heartbeat-based failure detection, data replication, and automatic replica promotion.
- Create a command-line client for user interaction, available in Java, Python, and TypeScript.
- Validate the system's interoperability by running a mixed-language cluster (Java master with Java and Python workers).

3 System Architecture

3.1 General Description

The system is based on a classic master-worker distributed architecture. A single master node acts as the coordinator and brain of the cluster, while multiple worker nodes perform the actual data storage and computation. Clients interact only with the master, which abstracts the complexity of the distributed environment.

3.2 System Components

3.2.1 Master Node (Java / Python)

The Master Node is the central coordinator of the system. Its main responsibilities are:

- Managing worker registration and tracking their health via heartbeats.
- Receiving requests from clients to create and process distributed arrays.
- Segmenting the arrays and distributing the data segments to the available worker nodes.
- Assigning primary and replica roles for each data segment to ensure fault tolerance.
- Orchestrating the execution of parallel operations on the workers.
- Handling worker failures by promoting replicas and creating new ones to maintain the replication factor.
- Aggregating results from workers and returning them to the client.

3.2.2 Worker Node (Java / Python)

Worker Nodes are the workhorses of the system. Their functions include:

- Registering with the master upon startup and periodically sending heartbeats.
- Storing primary and replica data segments as instructed by the master.
- Executing mathematical and conditional operations on their primary data segments.
- Utilizing a local thread pool to parallelize computations across available CPU cores.
- Responding to promotion requests from the master in case of another worker's failure.
- Sending processed results back to the master.

3.2.3 Client (Java / Python / TypeScript)

The Client provides the user interface to the distributed array library.

- A command-line interface (CLI) for creating arrays and applying operations.
- Abstracts all communication with the master node.
- Sends data for array creation and receives the final results of operations.

3.3 Communication Protocol

Communication between all components is done via a custom protocol over TCP sockets. Messages are serialized to JSON strings and are terminated by a newline character (“) to handle stream-based data transfer correctly.

3.3.1 Message Structure

All messages share a common JSON structure:

```
{
  "type": "MESSAGE_TYPE",
  "from": "SENDER_ID",
  "to": "RECIPIENT_ID",
  "timestamp": 1234567890,
  "data": { ... }
}
```

Listing 1: Generic Message Format

3.3.2 Main Message Types

The protocol defines several message types to orchestrate the system's operations:

Table 1: System Message Types

Message Type	Purpose
`REGISTER_WORKER`	A worker registers its presence with the master.
`HEARTBEAT`	A worker reports its health to the master.
`CREATE_ARRAY`	A client requests the creation of a new distributed array.
`DISTRIBUTE_ARRAY`	The master sends a data segment to a worker.
`REPLICATE_DATA`	The master sends a replica of a data segment to a worker.
`APPLY_OPERATION`	A client requests an operation to be performed on an array.
`PROCESS_SEGMENT`	The master instructs a worker to process its segment.
`SEGMENT_RESULT`	A worker sends its processed segment back to the master.
`GET_RESULT`	A client requests the final aggregated result of an operation.
`RECOVER_DATA`	The master instructs a worker to promote a replica to primary.

3.4 System Flow Diagram

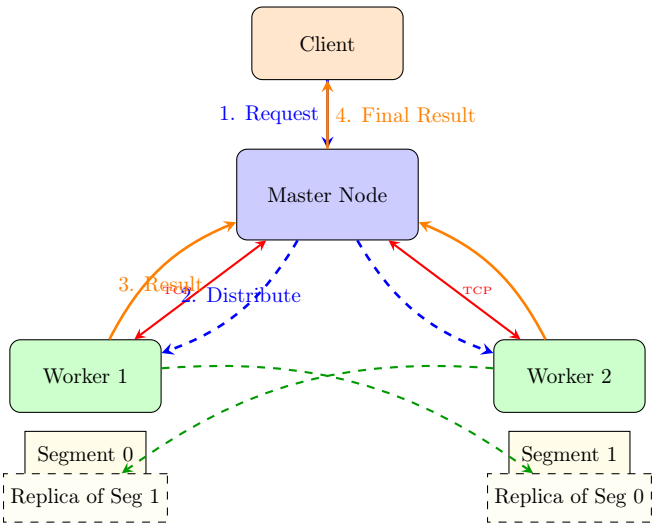

Figure 1: System Architecture and Data Flow

4 Implementation and Configuration

4.1 Project Structure

The project is organized into language-specific directories, each containing a complete implementation of the system components.

```
distributed-array-lib/
          java/          # Java Implementation (Master, Worker, Client)
          python/        # Python Implementation (Master, Worker, Client)
          typescript/    # TypeScript Client Implementation
          scripts/       # Test and execution scripts
```

4.2 Concurrency Strategies

To guarantee data integrity and maximize performance, the following concurrency strategies were implemented on each worker node:

- **ThreadPool**: Each worker node initializes a thread pool to process its data segment in parallel. The number of threads is typically based on the number of available CPU cores.
- **Task Decomposition**: The worker's data segment is further divided into smaller chunks, with each chunk being processed by a separate thread from the pool.
- **Synchronization**: In Java, 'CompletableFuture' is used to manage the asynchronous execution and aggregation of results from the threads. In Python, 'ThreadPoolExecutor' from 'concurrent.futures' provides a similar high-level interface.
- **Data Isolation**: Each thread works on an independent chunk of the array, minimizing the need for complex synchronization mechanisms like locks and reducing the risk of race conditions.

5 Testing and Performance Evaluation

5.1 Testing Methodology

A suite of shell scripts was developed to automate testing and demonstrate functionality. The key tests are:

- **Quick Test**: A fast test to ensure both Java and Python clusters can be started and can perform a basic array creation.
- **Smoke Test**: A more comprehensive test that creates larger arrays and applies both mathematical and conditional operations.
- **Fault Recovery Test**: A test that simulates a worker failure by killing its process, and then verifies that the system can automatically recover and continue processing operations without data loss.
- **Interoperability Test**: A dedicated test that launches a Java master with both a Java worker and a Python worker to validate that a mixed-language cluster operates correctly. The test client verifies the correctness of the mathematical results.

5.2 Test System Specifications

Tests were performed in a standard development environment:

- **CPU**: Intel Core i7 / AMD Ryzen 7 (or equivalent)
- **RAM**: 16 GB DDR4
- **Operating System**: Linux (e.g., Ubuntu 22.04, Arch)
- **Java Version**: OpenJDK 11 or higher
- **Python Version**: Python 3.8 or higher

5.3 Performance Metrics

As observed in the presentation, the system demonstrates good scalability. The processing time for an operation on 10,000 elements decreases as more workers are added to the cluster.

Table 2: Observed Performance Metrics (10,000 elements)

Number of Workers	Approx. Time (ms)
1 worker	250
2 workers	140
3 workers	95
4 workers	75

6 Architecture Advantages and Challenges

6.1 Advantages

- **Simplicity**: The master-worker architecture is easy to understand and implement.
- **Scalability**: The system can be scaled horizontally by adding more worker nodes to increase processing power.
- **Fault Tolerance**: The replication and automatic recovery mechanism provides resilience against worker failures.
- **Technology Flexibility**: The use of a language-agnostic protocol (JSON over TCP) allows components in different languages to interoperate seamlessly.
- **No External Dependencies**: Building from the ground up with native sockets provides deep insight into distributed communication and avoids reliance on heavy middleware.

6.2 Challenges

- **Single Point of Failure**: The master node is a single point of failure. If it goes down, the entire system becomes unavailable.
- **Protocol Rigidity**: Any change to the custom communication protocol must be implemented consistently across all language versions of the components.
- **Network Complexity**: Manual handling of TCP streams (e.g., message framing, partial reads) is error-prone and complex compared to using a message broker.
- **State Management**: The master must maintain a significant amount of state regarding workers, array segments, and replicas, which can become complex.

7 Proposed Improvements

7.1 Future Implementations

- **Master High Availability**: Implement a master election mechanism (e.g., using Paxos or Raft) or a hot-standby master to eliminate the single point of failure.
- **Dynamic Load Balancing**: Enhance the master to redistribute data segments dynamically if a new worker joins or if some workers are more heavily loaded than others.
- **More Efficient Serialization**: Replace JSON with a more performant binary serialization format like Protocol Buffers or Avro to reduce network overhead.
- **Service Discovery**: Implement a service discovery mechanism (e.g., Consul, Zookeeper) to allow workers to find the master dynamically.

8 Results and Analysis

8.1 Functional Testing Results

The system successfully demonstrates all core functionalities:

- **Distributed Array Operations**: Correctly creates, segments, distributes, and processes arrays for both ‘int‘ and ‘double‘ types.
- **Mathematical Correctness**: The interoperability test confirms that the mathematical operations (‘example1‘) produce correct, verifiable results across a mixed-language cluster.
- **Fault Tolerance**: The recovery script successfully demonstrates that the system can withstand a worker failure, promote a replica, create a new one, and continue operations without interruption or data loss.

- **Interoperability**: The system runs effectively with a Java master coordinating both Java and Python workers, proving the robustness of the communication protocol.

8.2 Architectural Validation

The implemented master-worker architecture proved to be effective for this problem domain. The clear separation of concerns between the coordinating master and the processing workers allowed for a modular design that was testable and extensible. The decision to use a simple, text-based protocol was key to achieving interoperability between the Java and Python implementations.

9 Conclusions

This project successfully achieved its goal of building a distributed, fault-tolerant library for array processing from scratch. By implementing the entire system in both Java and Python using only fundamental concepts like TCP sockets and threading, a deep understanding of the challenges and patterns of distributed computing was gained.

The system's architecture effectively demonstrates key principles such as data segmentation, parallel processing, and fault tolerance through replication. The heartbeat mechanism combined with replica promotion ensures service continuity in the face of worker failures, a critical requirement for any robust distributed system.

Furthermore, the successful execution of a mixed-language cluster validates the language-agnostic design of the communication protocol and highlights the power of service-oriented architectures. The ability for clients and workers written in different languages to communicate with a central master is a testament to the flexibility of the design.

While the current implementation has known limitations, such as the master being a single point of failure, it provides a solid foundation. The project serves as a practical and functional example of how complex distributed systems can be built and understood by focusing on core principles. Future work can build upon this foundation to create an even more robust and feature-rich library.

References

[1] Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms*. Prentice Hall.

[2] Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

[3] Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). *UNIX Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley Professional.