Distributed Array Library

Concurrent and Distributed Processing

CC4P1 Concurrent and Distributed Programming

André Pacheco, Arbues Perez, Sergio Pezo

July 2025

Agenda

o

- Project Goal
- Architecture and Design
- Implementation
- Communication Protocol
- Operation Examples
- Replication and Recovery
- Fault Tolerance
- Demonstration
- Conclusions

Project Goal

O

Develop a distributed library

- Distributed arrays: `DArrayInt` and `DArrayDouble`

Project Goal

o

Develop a distributed library

- Distributed arrays: `DArrayInt` and `DArrayDouble`
- Concurrent and parallel processing

Project Goal

O

Develop a distributed library

- Distributed arrays: `DArrayInt` and `DArrayDouble`
- Concurrent and parallel processing
- Communication via native TCP sockets

Project Goal

o

Develop a distributed library

- Distributed arrays: `DArrayInt` and `DArrayDouble`
- Concurrent and parallel processing
- Communication via native TCP sockets
- No external frameworks

Project Goal

O

Develop a distributed library

- Distributed arrays: `DArrayInt` and `DArrayDouble`
- Concurrent and parallel processing
- Communication via native TCP sockets
- No external frameworks
- Basic fault tolerance

Project Goal

o

Develop a distributed library

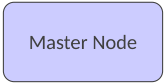
- Distributed arrays: `DArrayInt` and `DArrayDouble`
- Concurrent and parallel processing
- Communication via native TCP sockets
- No external frameworks
- Basic fault tolerance

Implementations

- Java 8+
- Python 3.6+
- TypeScript (Client)

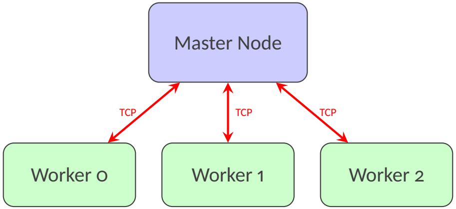
System Architecture

o



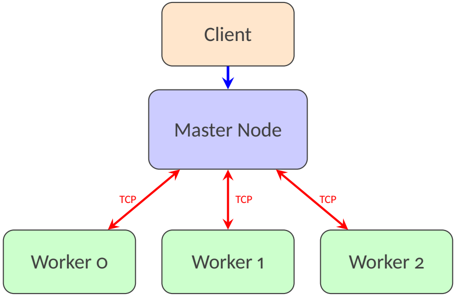
System Architecture

o



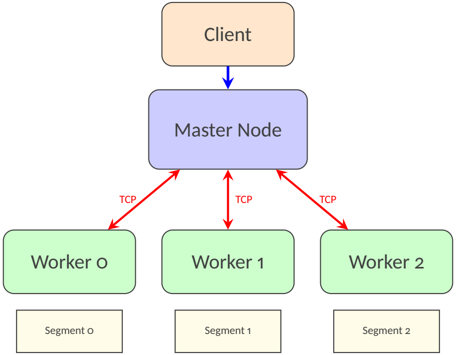
System Architecture

o



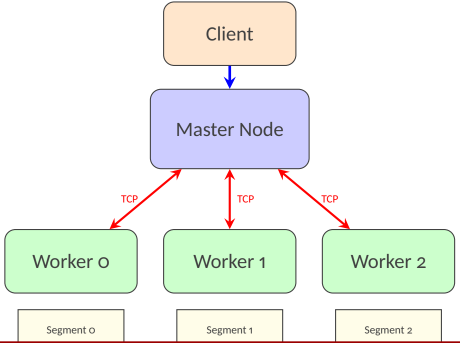
System Architecture

o



System Architecture

0



Features

- Master-worker architecture
- Automatic data distribution
- Bidirectional communication

Implementation - Structure

o

Java	Python / TypeScript
• `MasterNode.java`	• `master_node.py`

Implementation - Structure

o

Java

- `MasterNode.java`
- `WorkerNode.java`

Python / TypeScript

- `master_node.py`
- `worker_node.py`

Implementation - Structure

O

Java

- `MasterNode.java`
- `WorkerNode.java`
- `DArrayInt.java`
- `DArrayDouble.java`

Python / TypeScript

- `master_node.py`
- `worker_node.py`
- `darray.py`

Implementation - Structure

o

Java

- `MasterNode.java`
- `WorkerNode.java`
- `DArrayInt.java`
- `DArrayDouble.java`
- `Message.java`

Python / TypeScript

- `master_node.py`
- `worker_node.py`
- `darray.py`
- `message.py`

Implementation - Structure

o

Java

- `MasterNode.java`
- `WorkerNode.java`
- `DArrayInt.java`
- `DArrayDouble.java`
- `Message.java`
- `DistributedArrayClient.java`

Python / TypeScript

- `master_node.py`
- `worker_node.py`
- `darray.py`
- `message.py`
- `distributed_array_client.py`

Implementation - Structure

o

Java

- `MasterNode.java`
- `WorkerNode.java`
- `DArrayInt.java`
- `DArrayDouble.java`
- `Message.java`
- `DistributedArrayClient.java`

Python / TypeScript

- `master_node.py`
- `worker_node.py`
- `darray.py`
- `message.py`
- `distributed_array_client.py`
- `DistributedArrayClient.ts`

Array Segmentation

o

Original Array (10,000 elements)

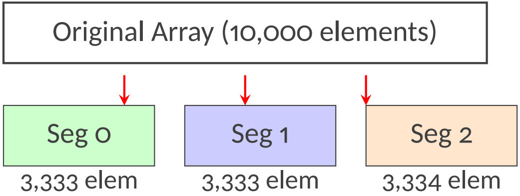
Array Segmentation

o

Original Array (10,000 elements)

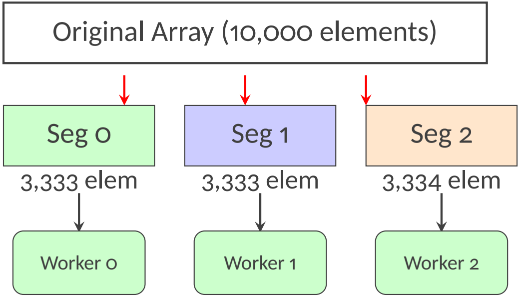
Array Segmentation

o



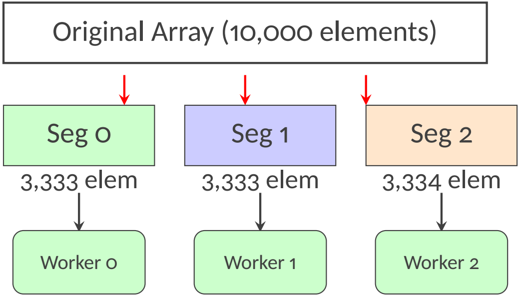
Array Segmentation

0



Array Segmentation

o



Segmentation Algorithm

- Equal division: $\frac{\text{total}}{\text{workers}}$
- Distributed remainder handling
- Round-robin assignment

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- `REGISTER_WORKER` - Worker registration

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- `REGISTER_WORKER` - Worker registration
- `DISTRIBUTE_ARRAY` - Segment distribution

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- REGISTER_WORKER - Worker registration
- DISTRIBUTE_ARRAY - Segment distribution
- PROCESS_SEGMENT - Processing order

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- REGISTER_WORKER - Worker registration
- DISTRIBUTE_ARRAY - Segment distribution
- PROCESS_SEGMENT - Processing order
- HEARTBEAT - Health check

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- REGISTER_WORKER - Worker registration
- DISTRIBUTE_ARRAY - Segment distribution
- PROCESS_SEGMENT - Processing order
- HEARTBEAT - Health check
- REPLICATE_DATA - Segment replication

Communication Protocol

o

JSON Format

```
{
  type: MESSAGE_TYPE,
  from: NODE_ID,
  to: NODE_ID,
  timestamp: 1234567890,
  data: {},
  status: OK
}
```

Message Types

- REGISTER_WORKER - Worker registration
- DISTRIBUTE_ARRAY - Segment distribution
- PROCESS_SEGMENT - Processing order
- HEARTBEAT - Health check
- REPLICATE_DATA - Segment replication
- RECOVER_DATA - Failure recovery

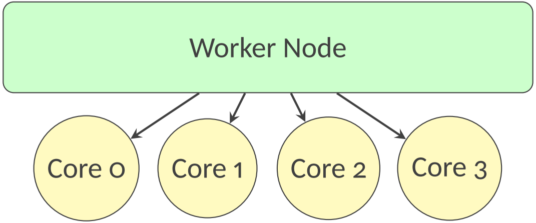
Parallel Processing

o

Worker Node

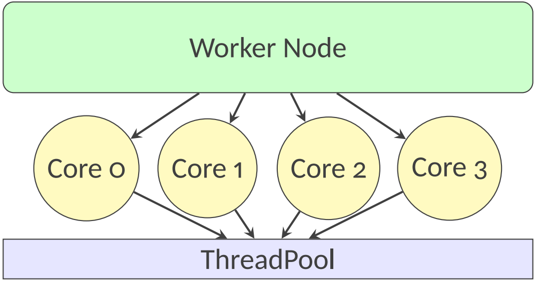
Parallel Processing

o



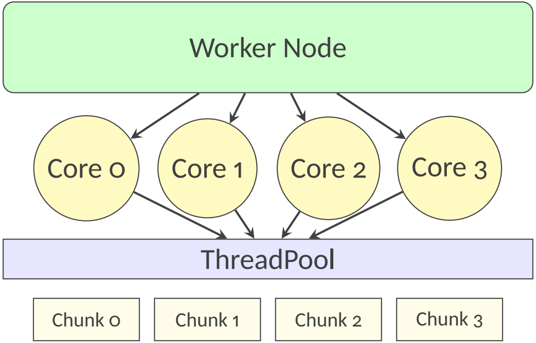
Parallel Processing

o



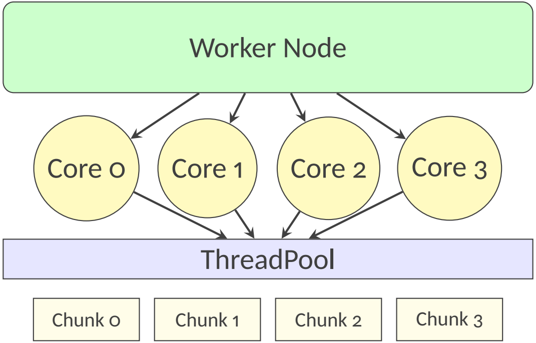
Parallel Processing

o



Parallel Processing

o



Strategy

- Automatic core detection: `Runtime.availableProcessors()`
- Division of the segment into chunks
- Concurrent processing with ThreadPool
- Synchronization using `Future<T>`

Example 1: Mathematical Operations

O

Formula

$$\text{result} = \frac{(\sin(x) + \cos(x))^2}{\sqrt{|x|} + 1}$$

Example 1: Mathematical Operations

o

Formula

$$\text{result} = \frac{(\sin(x) + \cos(x))^2}{\sqrt{|x|} + 1}$$

Java Implementation

- Parallel processing with ThreadPool
- Division of the segment into chunks
- Each thread processes its chunk independently

Example 1: Mathematical Operations

o

Formula

$$\text{result} = \frac{(\sin(x) + \cos(x))^2}{\sqrt{|x|} + 1}$$

Java Implementation

- Parallel processing with ThreadPool
- Division of the segment into chunks
- Each thread processes its chunk independently

Python Implementation

- Use of ThreadPoolExecutor
- NumPy for vectorized operations
- Concurrent processing by chunks

Example 2: Conditional Evaluation

o

Condition

If $x \bmod 3 = 0$ or $500 \leq x \leq 1000$:

$$\text{result} = (x \cdot \log(x)) \bmod 7$$

Example 2: Conditional Evaluation

o

Condition

If $x \bmod 3 = 0$ or $500 \leq x \leq 1000$:

$$\text{result} = (x \cdot \log(x)) \bmod 7$$

Processing

- Conditional evaluation for each element
- Application of logarithmic transformation
- Preservation of values that do not meet the condition

Example 2: Conditional Evaluation

o

Condition

If $x \bmod 3 = 0$ or $500 \leq x \leq 1000$:

$$\text{result} = (x \cdot \log(x)) \bmod 7$$

Processing

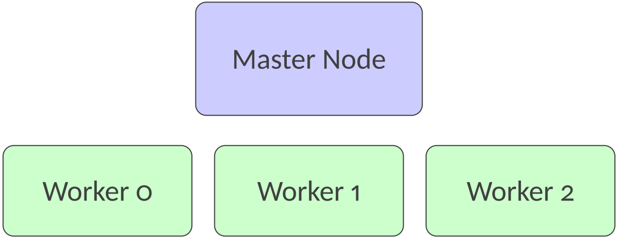
- Conditional evaluation for each element
- Application of logarithmic transformation
- Preservation of values that do not meet the condition

Resilience

- Exception handling per thread
- Continuation in case of partial failures
- Consolidation of valid results

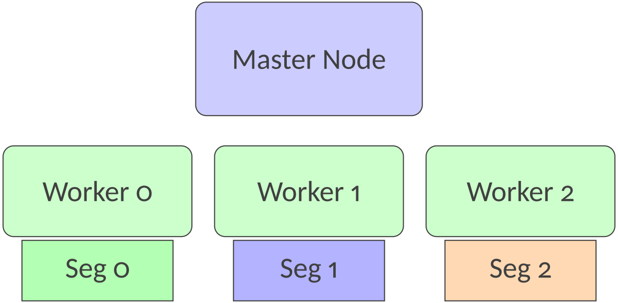
Data Replication

O



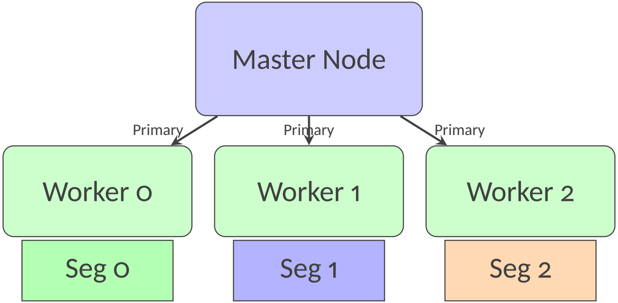
Data Replication

o



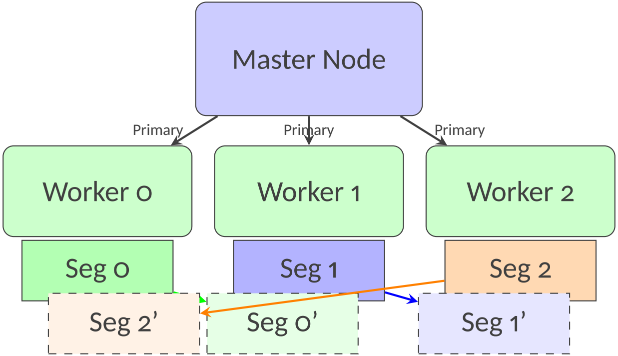
Data Replication

o



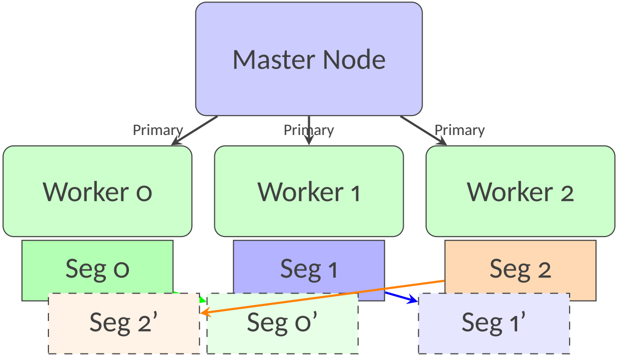
Data Replication

o



Data Replication

o



Replication factor = 2 (primary + 1 replica)

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)
2. Worker marked as down

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)
2. Worker marked as down
3. Activate recovery process

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)
2. Worker marked as down
3. Activate recovery process

Replica Promotion

1. Identify affected segments
2. Promote replicas to primary
3. Update segment mappings

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)
2. Worker marked as down
3. Activate recovery process

New Replica Creation

1. Select available workers
2. Replicate data from primary
3. Maintain replication factor

Replica Promotion

1. Identify affected segments
2. Promote replicas to primary
3. Update segment mappings

Recovery Mechanism

o

Failure Detection

1. Heartbeat timeout (10s)
2. Worker marked as down
3. Activate recovery process

New Replica Creation

1. Select available workers
2. Replicate data from primary
3. Maintain replication factor

Replica Promotion

1. Identify affected segments
2. Promote replicas to primary
3. Update segment mappings

Redistribution

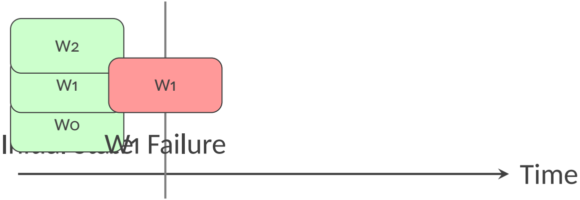
1. Balance load among workers
2. Avoid node overload
3. Optimize resource usage

Example 3: Fault Recovery

o

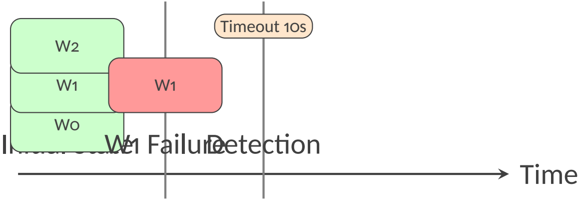
Example 3: Fault Recovery

o



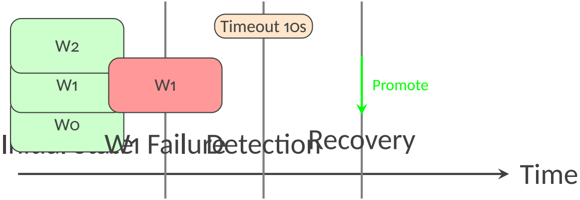
Example 3: Fault Recovery

o



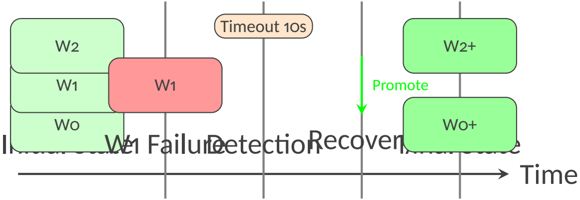
Example 3: Fault Recovery

o



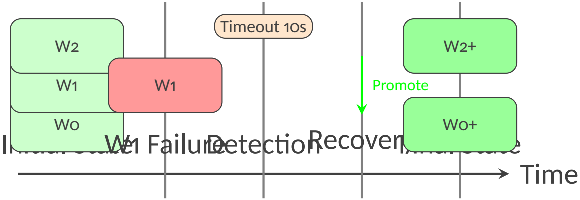
Example 3: Fault Recovery

o



Example 3: Fault Recovery

o



Automatic Process

- No data loss
- Service continuity
- Transparent to the client

Demonstration - Automatic Recovery

o

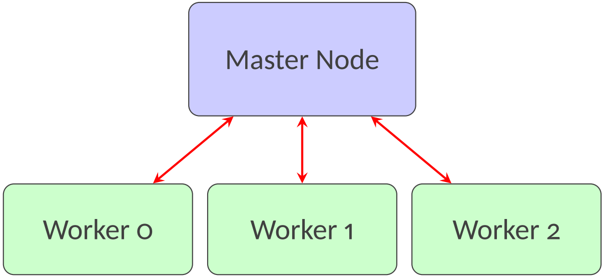
```
$ ./test-recovery.sh
=== Distributed Array Recovery Test ===
Starting Master node on port 5000
Starting Worker-1
Starting Worker-2
Starting Worker-3

=== Creating distributed array ===
Create array response: {status:created,arrayId:myArray}
INFO: Replicated segment 0 to worker-2
INFO: Replicated segment 100 to worker-3
INFO: Replicated segment 200 to worker-1

=== Simulating Worker-2 failure ===
Worker-2 has been terminated!
WARNING: Worker worker-2 failed health check
ERROR: Handling failure of worker: worker-2
```

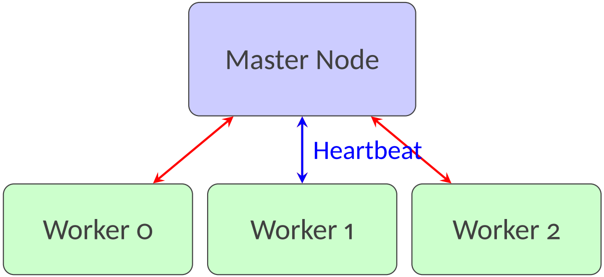
Fault Tolerance

o



Fault Tolerance

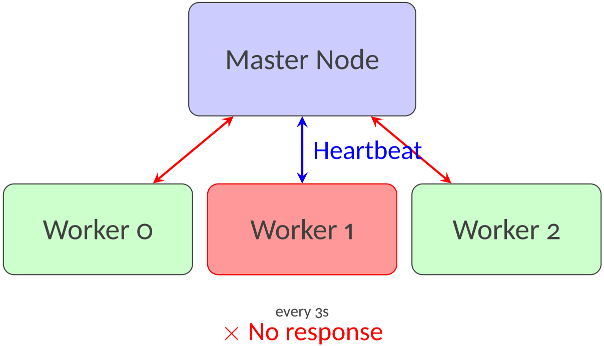
o



every 3s

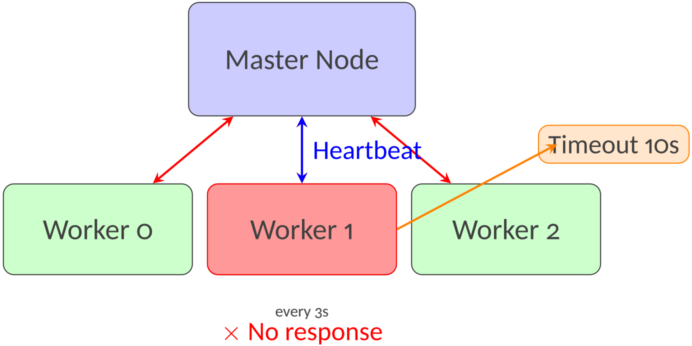
Fault Tolerance

o



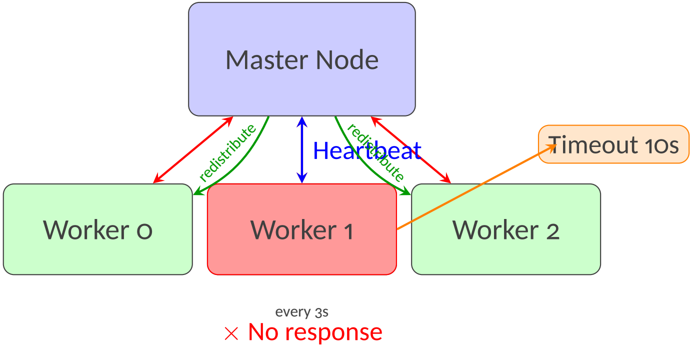
Fault Tolerance

o



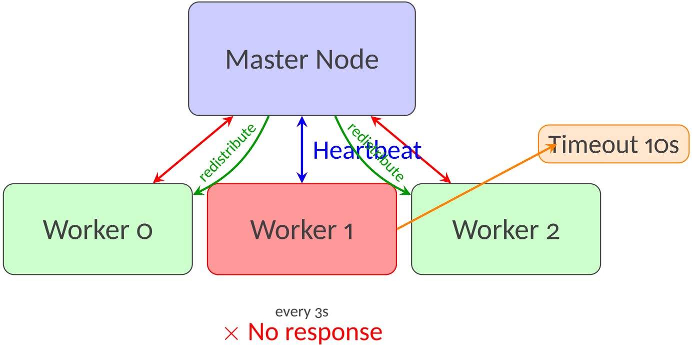
Fault Tolerance

o



Fault Tolerance

o



Fault Tolerance System

- Heartbeat: check every 3 seconds
- Detection: 10-second timeout
- Replication: factor 2 (primary + replica)

Demonstration - Cluster Start

o

```
$ ./start-java-cluster.sh
Starting Java distributed array cluster...
Starting master node on port 5000...
Master node PID: 12345
Starting worker-0...
Worker-0 PID: 12346
Starting worker-1...
Worker-1 PID: 12347
Starting worker-2...
Worker-2 PID: 12348

Java cluster started successfully!
Master node running on port 5000
3 worker nodes connected
```

TypeScript Client

o

Features

- Full client in TypeScript/Node.js

TypeScript Client

o

Features

- Full client in TypeScript/Node.js
- Compatible with Java and Python clusters

TypeScript Client

O

Features

- Full client in TypeScript/Node.js
- Compatible with Java and Python clusters
- Identical CLI interface

TypeScript Client

o

Features

- Full client in TypeScript/Node.js
- Compatible with Java and Python clusters
- Identical CLI interface
- Asynchronous communication with Promises

TypeScript Client

O

Features

- Full client in TypeScript/Node.js
- Compatible with Java and Python clusters
- Identical CLI interface
- Asynchronous communication with Promises
- Strong typing with interfaces

TypeScript Client

o

Features

- Full client in TypeScript/Node.js
- Compatible with Java and Python clusters
- Identical CLI interface
- Asynchronous communication with Promises
- Strong typing with interfaces

Usage Example

```
$ npm start -- localhost 5000
Connected to master at localhost:5000
Enter commands (type help for usage, exit to quit):
> create-double ts-array 5000
Create array response: {status:created}
> apply ts-array example1
Apply operation response: {status:processing}
```

Demonstration - Interactive Client

o

```
$ java -cp out:lib/* client.DistributedArrayClient localhost 5000
Connected to master at localhost:5000
Enter commands (type help for usage, exit to quit):
> create-double math-array 10000
Create array response: {type:OPERATION_COMPLETE,
  data:{arrayId:math-array,status:created}}

> apply math-array example1
Apply operation response: {type:OPERATION_COMPLETE,
  data:{status:processing}}

> get math-array
Get result response: {type:OPERATION_COMPLETE,
  data:{status:complete,result:Operation completed}}
```

System Logs

o

```
master.log
INFO: Master node started on port 5000
INFO: Worker registered: worker-0 from 127.0.0.1
INFO: Worker registered: worker-1 from 127.0.0.1
INFO: Worker registered: worker-2 from 127.0.0.1
INFO: Received array creation request: math-array (10000 elements)
INFO: Array segmented: 3 segments distributed
INFO: Processing operation: example1 on math-array

worker-0.log
INFO: Registered with master node
INFO: Received double array segment: math-array with 3333 elements
INFO: Processing Example 1 using 4 threads
INFO: Completed Example 1 processing for math-array
INFO: Sent result to master
```

Performance and Scalability

O

Parallelization

- Use of all cores

Performance and Scalability

o

Parallelization

- Use of all cores
- Efficient ThreadPool

Performance and Scalability

o

Parallelization

- Use of all cores
- Efficient ThreadPool
- Automatic work division

Performance and Scalability

o

Parallelization

- Use of all cores
- Efficient ThreadPool
- Automatic work division

Distribution

- Equal segmentation
- Asynchronous communication
- Independent processing

Performance and Scalability

o

Parallelization

- Use of all cores
- Efficient ThreadPool
- Automatic work division

Distribution

- Equal segmentation
- Asynchronous communication
- Independent processing

Metrics (10,000 elements)

- 1 worker: 250ms
- 2 workers: 140ms
- 3 workers: 95ms
- 4 workers: 75ms

Performance and Scalability

o

Parallelization

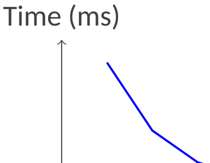
- Use of all cores
- Efficient ThreadPool
- Automatic work division

Distribution

- Equal segmentation
- Asynchronous communication
- Independent processing

Metrics (10,000 elements)

- 1 worker: 250ms
- 2 workers: 140ms
- 3 workers: 95ms
- 4 workers: 75ms



Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing
- Effective parallelization per node

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing
- Effective parallelization per node
- Complete replication and recovery system

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing
- Effective parallelization per node
- Complete replication and recovery system
- No external framework dependencies

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing
- Effective parallelization per node
- Complete replication and recovery system
- No external framework dependencies
- Interoperability between languages

Applications

- Large dataset processing
- Distributed scientific calculations
- Parallel data analysis

Conclusions

o

Achievements

- Functional library in Java, Python, and TypeScript
- Truly distributed processing
- Effective parallelization per node
- Complete replication and recovery system
- No external framework dependencies
- Interoperability between languages

Applications

- Large dataset processing
- Distributed scientific calculations
- Parallel data analysis

Advanced Implemented Features

Questions

o

Thank you for your attention

GitHub: `https://github.com/A-PachecoT/distributed-array-lib`