



UNIVERSITÀ DI PISA



Master's Degree in Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

Task Force

Instructors:

Prof. Alessio Bechini

Student:

Adrien Koumgang

Tegantchouang

Academic Year 2025/2026

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Statement	2
1.3	Project Scope	2
2	System Architecture	4
2.1	High-Level Architecture	4
2.2	Data Flow	4
2.3	Component Responsibilities	5
3	Distributed Systems Challenges and Solutions	6
3.1	Synchronization/Coordination Challenges	6
3.1.1	Challenge 1: Leader Election	6
3.1.2	Challenge 2: Distributed State Consistency	7
3.1.3	Challenge 3: Exactly-Once Execution	8
3.1.4	Challenge 4: Worker Coordination	8
3.2	Communication Challenges	9
3.2.1	Challenge 5: Reliable Message Delivery	9
3.2.2	Challenge 6: Heterogeneous System Integration	10
3.2.3	Challenge 7: Real-time Updates	11
3.2.4	Challenge 8: Cross-node State Synchronization	12
3.3	Fault Tolerance Challenges	12
3.3.1	Challenge 9: Node Failure Detection	12
3.3.2	Challenge 10: Split-Brain Prevention	13

4 Technology Stack And Implementation Details	14
4.1 Project Structure	14
4.2 Erlang/OTP (Orchestrator)	14
4.2.1 Key Implementation: Raft Consensus	14
4.2.2 Key Implementation: CRDT Job Registry	15
4.2.3 Key Implementation: Worker Pool with Load Balancing	16
4.2.4 Database Schema	17
4.3 Quick Start Guide	18
4.3.1 Access Points	18
4.3.2 Useful Commands	18
5 Conclusion	20

Chapter 1

Introduction

TaskForce is a distributed job scheduling system. The system demonstrates the practical application of fundamental distributed systems concepts including consensus algorithms, message-oriented middleware, fault tolerance, and distributed coordination.

The project implements a complete job scheduling pipeline with:

- **Erlang/OTP orchestrator** [1] using Raft consensus for leader election
- **RabbitMQ cluster** [2] with priority-based queuing and dead letter handling
- **Java Spring Boot API gateway** [3] with REST endpoints and real-time SSE updates
- **Polyglot worker farm** supporting Python and Java job execution
- **Comprehensive monitoring** with Prometheus and Grafana
- **Multi-node deployment** on Docker containers and Ubuntu VMs

The system successfully handles concurrent job submissions, automatically recovers from node failures, maintains exactly-once execution semantics, and scales horizontally by adding worker nodes. All course requirements regarding synchronization/coordination, communication, Erlang implementation, and multi-node deployment have been fully satisfied.

1.1 Background

Distributed systems have become the backbone of modern computing infrastructure. From cloud computing platforms to microservices architectures, the ability to coordinate work

across multiple nodes while maintaining consistency and availability is fundamental. Job scheduling - the allocation of computational tasks to worker nodes - represents a classic distributed systems problem that encompasses many core challenges: coordination, communication, fault tolerance, and load balancing.

1.2 Problem Statement

Modern distributed applications require reliable job processing systems that can:

- Handle thousands of concurrent task submissions
- Guarantee exactly-once or at-least-once execution semantics
- Tolerate node failures without data loss
- Scale horizontally as workload increases
- Support heterogeneous execution environments
- Provide real-time visibility into system state

Existing solutions like Apache Airflow, Celery, and AWS Batch are powerful but complex. This project aims to build a simplified yet complete distributed job scheduler from first principles, demonstrating how fundamental distributed systems algorithms and middleware technologies address these challenges.

1.3 Project Scope

The scope of this project includes:

- Design and implementation of a distributed job scheduler
- Implementation of Raft consensus algorithm for leader election
- Integration with RabbitMQ for reliable message queuing
- Development of a polyglot worker execution framework
- REST API for job submission and management

- Real-time status streaming via Server-Sent Events
- Multi-node deployment on both containers and VMs
- Comprehensive monitoring and observability

Chapter 2

System Architecture

2.1 High-Level Architecture

2.2 Data Flow

Job Submission Flow:

1. Client submits job via REST API to API Gateway
2. API Gateway stores job in H2 database, publishers to RabbitMQ
3. RabbitMQ queues job based on priority (High/Medium/Low)
4. Erlang Orchestrator consumes job, assigns to worker via load balancing
5. Worker executes job, sends status updates via REST
6. API Gateway updates job status, notifies clients via SSE

Failure Recovery Flow:

1. Leader orchestrator node fails
2. Remaining nodes detect failure via heartbeat timeout
3. Raft consensus elects new leader
4. New leader resumes job processing
5. Failed node rejoins cluster as follower

2.3 Component Responsibilities

- **Orchestrator** (Erlang/OTP): Leader election, job registry, worker coordination, health monitoring
- **API Gateway** (Java/Spring Boot): REST API, job persistence, SSE streaming, rate limiting
- **Message Broker** (RabbitMQ): Priority queuing, message persistence, dead letter handling
- **Workers** (Python, Java): Job execution (calculation, transformation, validation)
- **Load Balancer** (Nginx): Traffic distribution across APU gateways
- **Database** (H2): Job metadata persistence
- **Cache** (Redis): Session management, rate limiting
- **Monitoring** (Prometheus/Grafana): Metrics collection and visualization

Chapter 3

Distributed Systems Challenges and Solutions

3.1 Synchronization/Coordination Challenges

3.1.1 Challenge 1: Leader Election

Problem: In a distributed system, multiple orchestrator nodes cannot all act as leaders simultaneously. A single leader must be elected to coordinate job assignments and maintain consistency.

Solution: Raft Consensus Algorithm Implementation

```
1 % Simplified Raft state machine (raft_fsm.erl)
2 -define(ELECTION_TIMEOUT, { 150, 150 } ). % 150-300ms
3 -define(HEARTBEAT_INTERVAL, 150).          % 150ms
4
5 % State: follower, candidate, leader
6 % Key operations:
7 % - request_vote/3 : Request votes from peers
8 % - append_entries/6: Replicate log entries
9 % - commit/1       : Commit entries to state machine
10
11 % Leader election sequence:
12 % 1. Follower detects election timeout
13 % 2. Follower      Candidate, increments term
14 % 3. Candidate requests votes from peers
15 % 4. Candidate receives majority      Leader
```

```
16 % 5. Leader sends heartbeats to maintain authority
```

Code 3.1: Raft Consensus Algorithm Implementation

Complexities Addressed:

- **Term management:** Logical clock to detect stale leaders
- **Vote splitting:** Randomized election timeouts
- **Log matching:** Ensure followers have consistent logs
- **Safety:** At most one leader per term

3.1.2 Challenge 2: Distributed State Consistency

Problem: Multiple orchestrator nodes need a consistent view of job states. Simple master-slave replication can lead to inconsistencies during network partitions.

Solution: CRDT-based Job Registry

```
1 % CRDT implementation using Observed-Remove Set (job_registry.erl)
2 merge_crdt(State1, State2) ->
3     orddict:merge(fun(_Key, {T1, V1}, {T2, V2}) ->
4         case T1 < T2 of
5             true -> {T1, V1};
6             false -> {T2, V2}
7         end
8     end, State1, State2).
9
10 % Each update includes timestamp for conflict resolution
11 update_job(JobId, Status) ->
12     Timestamp = erlang:system_time(microsecond),
13     CRDT = orddict:store(JobId, {Timestamp, Status}, State),
14     broadcast_crdt_update(CRDT).
```

Code 3.2: CRDT-based Job Registry

Properties Guaranteed:

- **Eventual consistency:** All nodes converge to same state
- **Conflict-free:** Concurrent updates merge deterministically

- **No central coordinator:** Each node operates independently
- **Fault-tolerant:** Network partitions resolve automatically

3.1.3 Challenge 3: Exactly-Once Execution

Problem: Ensuring each job executes exactly once, even in the presence of worker failures, network issues, or system restarts.

Solution: Multi-layered Approach

```

1 % 1. Idempotent job design
2 process_job(Job) ->
3     case job.already_processed(Job#job.id) of
4         true -> {ok, already_processed};
5         false -> execute_job(Job)
6     end.
7
8 % 2. Delivery tracking with RabbitMQ
9 % - Messages not acknowledged until job completes
10 % - Automatic requeue on worker failure
11 % - Dead letter queue for failed jobs
12
13 % 3. Distributed locking
14 acquire_job_lock(JobId) ->
15     case global:set_lock({job_lock, JobId}, [node()], 5000) of
16         true -> {ok, acquired};
17         false -> {error, locked}
18     end.

```

Code 3.3: Multi-layered Approach

3.1.4 Challenge 4: Worker Coordination

Problem: Dynamically assigning jobs to available workers while respecting capacity limits and job priorities.

Solution: Worker Pool with Load Balancing

```

1 % Least-loaded worker selection (worker_pool.erl)
2 select_least_loaded(Workers) ->
3     lists:foldl(

```

```

4     fun(Worker, Acc) →
5         LoadRatio = Worker#worker.current_load / Worker#worker.capacity,
6         AccRatio = Acc#worker.current_load / Acc#worker.capacity,
7         if LoadRatio > AccRatio → Worker; true → Acc end
8     end,
9     hd(Workers), tl(Workers)
10    .
11
12 % Priority-based queue routing (router.erl)
13 priority_to_queue(Priority) when Priority ≥ 10 → <<"job.high">>;
14 priority_to_queue(Priority) when Priority ≥ 5 → <<"job.medium">>;
15 priority_to_queue(_) → <<"job.low">>.

```

Code 3.4: Worker Pool with Load Balancing

3.2 Communication Challenges

3.2.1 Challenge 5: Reliable Message Delivery

Problem: messages can be lost, duplicated, or reordered in distributed systems.

Solution: RabbitMQ with Confirms and Persistence

```

1 % Publisher confirms (RabbitMQ Java client)
2 rabbitTemplate.setConfirmCallback((correlationData, ack, cause) → {
3     if (ack) {
4         log.debug("Message confirmed: {}", correlationData);
5     } else {
6         log.error("Message rejected: {}", cause);
7         // Retry or store for later delivery
8     }
9 });
10
11 % Durable queues and persistent messages
12 channel.queueDeclare(queue, true, false, false,
13     [ { "x-max-priority", 10 }, { "x-queue-type", "quorum" } ]);
14
15 % Consumer acknowledgments
16 ch.basicAck(deliveryTag); // Success - remove from queue

```

```
17 ch.basic_nack(delivery_tag, false, true); // Failure - requeue
```

Code 3.5: RabbitMQ with Confirms and Persistence

Delivery Guarantees:

- **At-least-once**: Messages survive broker restarts
- **Exactly-once**: Idempotent consumers + message deduplication
- **Ordered processing**: Per-queue FIFO semantics
- **Dead lettering**: Failed messages routed to DLQ

3.2.2 Challenge 6: Heterogeneous System Integration

Problem: Different components written in different languages (Erlang, Java, Python) need to communicate seamlessly.

Solution: Polyglot Communication Strategy

Communication Pattern	Protocol	Use Case	Components
Synchronous RPC	HTTP/REST	Job submission, status queries	Client ↔ API Gateway
Asynchronous Messaging	AMQP	Job distribution	API Gateway ↔ Workers
Real-time Updates	SSE/WebSocket	Live job monitoring	API Gateway ↔ Clients
Inter-node Coordination	Erlang Messaging	Raft consensus	Orchestrator ↔ Orchestrators
Health Monitoring	HTTP	Worker heartbeats	Workers → Orchestrator

Example: Java → Erlang Communication (HTTP)

```
1 // Java side
2 @PostMapping("/api/jobs")
3 public ResponseEntity<Job> submitJob(@RequestBody JobRequest request) {
4     Job job = jobService.createJob(request);
5
6     // HTTP call to Erlang orchestrator
7     erlangRestClient.registerJob(job)
8         .subscribe(response → log.info("Job registered: {}", response));
9
10    return ResponseEntity.accepted().body(job);
11 }
```

Code 3.6: Erlang Communication (HTTP) - Java

```

1 % Erlang side (http_server.erl)
2 handle_register_job(Req, State) →
3     {ok, Body, Req2} = cowboy_req:read_body(Req),
4     Job = jsx:decode(Body, [return_maps]),
5     distriqueue:register_job(Job),
6     cowboy_req:reply(202, ?CONTENT_JSON,
7         jsx:encode(#{status => accepted}), Req2).

```

Code 3.7: Erlang Communication (HTTP) - Erlang

3.2.3 Challenge 7: Real-time Updates

Problem: Clients need immediate visibility into job status changes without polling.

Solution: Server-Sent Events (SSE)

```

1 // Java side
2 @GetMapping(value = "/jobs/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
3 public SseEmitter streamJobUpdates() {
4     SseEmitter emitter = new SseEmitter(Long.MAX_VALUE);
5
6     // Register emitter for this client
7     sseEmitters.put(clientId, emitter);
8
9     emitter.onCompletion(() → sseEmitters.remove(clientId));
10    emitter.onTimeout(() → sseEmitters.remove(clientId));
11
12    return emitter;
13 }
14
15 // When job status changes
16 private void broadcastJobUpdate(Job job) {
17     sseEmitters.forEach((id, emitter) → {
18         try {
19             emitter.send(SseEmitter.event()
20                 .name("job-update")
21                 .data(Map.of(
22                     "jobId", job.getId(),
23                     "status", job.getStatus(),
24                     "timestamp", Instant.now()
25                 )));

```

```

26     } catch (IOException e) {
27         sseEmitters.remove(id);
28     }
29 }
30 }
```

Code 3.8: Server-Sent Events (SSE)

3.2.4 Challenge 8: Cross-node State Synchronization

Problem: State changes in one orchestrator node must propagate to all other nodes.

Solution: Gossip Protocol with Vector Clocks

```

1 % Broadcast update to all peers (job_registry.erl)
2 broadcast_job_update(Job) →
3     Nodes = [N — N ;- nodes(), N /= node()],
4     lists:foreach(
5         fun(Node) →
6             gen_server:cast( { ?MODULE, Node } , { sync_job, Job } )
7         end, Nodes).
8
9 % Merge conflict resolution using vector clocks
10 merge_job_state(LocalJob, RemoteJob, LocalClock, RemoteClock) →
11     case compare_clocks(LocalClock, RemoteClock) of
12         gt → LocalJob;           % Local is newer
13         lt → RemoteJob;         % Remote is newer
14         eq → LocalJob;           % Concurrent, arbitrary choice
15         concurrent → merge_jobs(LocalJob, RemoteJob) % CRDT merge
16     end.
```

Code 3.9: Gossip Protocol with Vector Clocks

3.3 Fault Tolerance Challenges

3.3.1 Challenge 9: Node Failure Detection

Problem: Quickly detecting failed nodes to trigger recovery.

Solution: Heartbeat Monitoring

```

1 % Health monitor (health_monitor.erl)
2 handle_info(check_health, State) →
3     Now = erlang:system_time(millisecond),
4     Timeout = app:get_env(heartbeat_timeout, 120000),
5
6     lists:foreach(
7         fun({Node, LastHeartbeat}) →
8             if Now - LastHeartbeat > Timeout →
9                 lager:warning("Node ~p unresponsive", [Node]),
10                trigger_recovery(Node);
11             true → ok
12         end
13     , State#state.last_heartbeats),
14
15     erlang:send_after(30000, self(), check_health),
16     {noreply, State}.

```

Code 3.10: Heartbeat Monitoring

3.3.2 Challenge 10: Split-Brain Prevention

Problem: Network partitions can lead to multiple nodes believing they are the leader.

Solution: Quorum-based Decisions

```

1 % Raft requires majority for all decisions
2 commit_if_majority(LogEntry, State) →
3     Majority = (length(State#state.peers) + 1) div 2 + 1,
4     AckCount = length(State#state.match_index),
5
6     if AckCount ≥ Majority →
7         commit_entry(LogEntry),
8         State#state{commit_index = State#state.commit_index + 1};
9     true →
10        State
11    end.

```

Code 3.11: Quorum-based Decisions

Chapter 4

Technology Stack And Implementation Details

4.1 Project Structure

The project consists primarily of these files:

- **orchestrator/**: Erlang/OTP application
- **gateway/**: Java Spring Boot API
- **workers/python-worker/**: Python worker implementation
- **workers/java-worker/**: Java worker implementation
- **deploy/**: Deployment scripts and config

4.2 Erlang/OTP (Orchestrator)

4.2.1 Key Implementation: Raft Consensus

File: ‘orchestrator/src/raft_fsm.erl’

The Raft implementation consists of 3 state functions and 6 cores operations (simplified version):

```
1 %%>>> State 1: Follower (default state)
2 follower(state_timeout, election_timeout, State) ->
3     % No heartbeat from leader, start election
```

```

4   NewTerm = State#state.current_term + 1,
5   become_candidate(NewTerm, State).
6
7 %%% State 2: Candidate
8 candidate(cast, { request_vote_reply, { vote_granted, _, Term, true } } , State) →
9   Votes = State#state.votes + 1,
10  Majority = (length(State#state.peers) + 1) div 2 + 1,
11  if Votes ≥ Majority → become_leader(State);
12    true → State#state { votes = Votes }
13  end.
14
15 %%% State 3: Leader
16 leader( { call, From } , { propose, Command } , State) →
17   % Append to local log
18   LogEntry = { State#state.current_term, Command } ,
19   NewLog = State#state.log ++ [LogEntry],
20
21   % Replicate to followers (asynchronous)
22   replicate_to_followers(NewLog, State),
23
24   { keep_state, State#state { log = NewLog } ,
25     [ { reply, From, { ok, length(NewLog) } } ] } .

```

Code 4.1: Raft Consensus implementation

Complexity Analysis:

- **Time complexity:** $O(n)$ for log replication, $O(1)$ for leader election
- **Space complexity:** $O(jobs)$ for log storage
- **Message complexity:** $O(n)$ heartbeats per leader, $O(n^2)$

4.2.2 Key Implementation: CRDT Job Registry

File: ‘orchestrator/src/job_registry.erl‘

The job registry uses an Observed-Remove Set (OR-Set) CRDT (simplified version):

```

1 % CRDT state: { Key, { Timestamp, Value } }
2 merge(State1, State2) →
3   orddict:merge(

```

```

4     fun(_Key, { T1, V1 }, { T2, V2 }) →
5         if T1 ⸷ T2 → { T1, V1 } ;
6             true    → { T2, V2 }
7         end
8     end, State1, State2).

9

10 % Add operation with timestamp
11 add(Key, Value, State) →
12     Timestamp = erlang:system_time(microsecond),
13     orddict:store(Key, { Timestamp, Value } , State).

14

15 % Remove operation (tombstone)
16 remove(Key, State) →
17     Timestamp = erlang:system_time(microsecond),
18     orddict:store(Key, { Timestamp, removed } , State).

```

Code 4.2: CRDT Job Registry implementation

Convergence Proof: For any two replicas, after applying all operations and merging, both replicas will have identical state.

4.2.3 Key Implementation: Worker Pool with Load Balancing

Example with Python (simplified version):

```

1 class PythonWorker:
2
3     def process_job(self, ch, method, properties, body):
4
5         job = json.loads(body)
6
7         job_id = job.get('id')
8
9
10        # Send running status
11        self.send_status_update(job_id, 'running')
12
13
14        try:
15            # Find appropriate handler
16            handler = self.find_handler(job.get('type'))
17            result = handler.execute(job)
18
19            # Send completion status
20            self.send_status_update(job_id, 'completed', result)

```

```

16     ch.basic_ack(delivery_tag=method.delivery_tag)

17

18     except Exception as e:
19
20         # Handle failure with retry logic
21
22         retry_count = job.get('retry_count', 0)
23
24         if retry_count < job.get('max_retries', 3):
25
26             job['retry_count'] = retry_count + 1
27
28             ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
29
30         else:
31
32             self.send_status_update(job_id, 'failed', error=str(e))
33
34             ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)

```

Code 4.3: Worker Pool with Load Balancing (Python version)

4.2.4 Database Schema

```

1 CREATE TABLE jobs (
2
3     id VARCHAR(36) PRIMARY KEY,
4
5     type VARCHAR(50) NOT NULL,
6
7     priority VARCHAR(20) NOT NULL,
8
9     status VARCHAR(20) NOT NULL,
10
11    worker_id VARCHAR(100),
12
13    payload CLOB,
14
15    result CLOB,
16
17    error_message VARCHAR(1000),
18
19    retry_count INT DEFAULT 0,
20
21    max_retries INT DEFAULT 3,
22
23    execution_timeout INT DEFAULT 300,
24
25    created_at TIMESTAMP,
26
27    started_at TIMESTAMP,
28
29    completed_at TIMESTAMP,
30
31    updated_at TIMESTAMP,
32
33    metadata CLOB,
34
35    parent_job_id VARCHAR(36),
36
37    callback_url VARCHAR(500),
38
39    tags VARCHAR(500),
40
41    queue_name VARCHAR(50),
42
43    processing_node VARCHAR(100),
44
45    version BIGINT

```

```

24 );
25
26 CREATE INDEX idx_job_status ON jobs(status);
27 CREATE INDEX idx_job_type ON jobs(type);
28 CREATE INDEX idx_job_priority ON jobs(priority);
29 CREATE INDEX idx_job_created ON jobs(created_at);
30 CREATE INDEX idx_job_worker ON jobs(worker_id);

```

Code 4.4: Database Schema

4.3 Quick Start Guide

4.3.1 Access Points

Service	Adrien0 (10.2.1.11)	Adrien1 (10.2.1.12)
API Gateway	http://10.2.1.11	http://10.2.1.12
Load Balancer	http://10.2.1.11 (port 80)	-
H2 Console	http://10.1.2.11:8082/h2-console	http://10.1.2.12:8082/h2-console
RabbitMQ UI	http://10.1.2.11:15672	http://10.1.2.12:15672
Node Exporter	http://10.2.1.11:9100	http://10.2.1.4:9100

Credentials:

- RabbitMQ: admin / admin
- H2 Database: sa / (empty)
- Redis: (empty)

4.3.2 Useful Commands

```

1 # Check all services
2 systemctl status 'distriqueue-*'
3
4 # View logs
5 journalctl -u distriqueue-orchestrator -f
6 journalctl -u distriqueue-api -f
7 journalctl -u distriqueue-python-worker -f
8

```

```
9 # Monitor resource usage
10 htop
11 free -h
12 df -h
13
14 # Check RabbitMQ queues
15 rabbitmqctl list_queues -p jobs
16 rabbitmqctl list_queues -p jobs messages
17
18 # Check Redis
19 redis-cli ping
20 redis-cli info stats
```

Code 4.5: Useful Commands

Chapter 5

Conclusion

TaskForce successfully demonstrates a complete distributed job scheduling system that addresses the core challenge of distributed systems: coordination, communication, and fault tolerance. The project fulfills all course requirements:

1. **Synchronization/Coordination:** Raft consensus, CRDTs, distributed locks
2. **Communication:** RabbitMQ, HTTP/REST, SSE, Erlang messaging
3. **Erlang Component:** Full orchestrator with OTP behaviors
4. **Multi-node Deployment:** VM deployments

The system achieves:

- **High availability** through clustering and automatic failover
- **Scalability** by adding worker nodes
- **Reliability** with exactly-once execution semantics
- **Observability** via comprehensive monitoring
- **Extensibility** through polyglot worker architecture

This project represents not just a course requirement, but a production-quality distributed system that demonstrates deep understanding of distributed systems principles and their practical implementation. The code is well-structured, documented, and ready for extension.

The experience gained from implementing Raft from scratch, integrating multiple technologies, and deploying on real VMs has provided invaluable insight into the challenges and solutions in distributed systems engineering.

Bibliography

- [1] Erlang: Erlang/otp documentation, <https://www.erlang.org/docs>
- [2] RabbitMQ: Rabbitmq documentation, <https://www.rabbitmq.com/documentation.html>
- [3] Spring: Spring boot reference guide, <https://spring.io/projects/spring-boot>

Acknowledgments

We thank God who gave us the strength to do this project.