UNIVERSITY OF PISA

MASTER'S DEGREE IN COMPUTER ENGINEERING

Distributed Systems and Middleware Technologies

Kairòs

Professor:

Alessio Bechini

Group members:

Matteo Halilaga

Lorenzo Mancinelli

ACADEMIC YEAR 2025/2026

Indice

Capitolo 1

Introduction

The proposed application is designed to support the collaborative organization of events within a group. Its primary goal is to provide a distributed, effective, and efficient mechanism capable of identifying an optimal solution that accounts for the constraints expressed by participants in terms of budget, time availability, and location.

The system adopts a distributed and scalable architecture, aimed at identifying the most fair solution for the entire group.

Users can interact with the system in the following ways:

- **Event creation**: define the deadline, description, visibility (public/private), and invite participants.
- **Event participation**: a user who wishes to participate (either in a public event or one they have been invited to) can specify their personal constraints. The system incrementally computes a set of partial solutions. When the deadline expires, these solutions are compared to determine the final outcome that best satisfies the collective constraints.

Capitolo 2

Functional Requirements

2.1 User level

- Authentication: users must be able to register, authenticate, and log out securely.
- Event management:
  - Organizers can create events by specifying the deadline, visibility (public/private), and - when needed - inviting additional users.
  - Participants can search for public events or those they have been invited to and submit their constraints.
- Solution computation: the system must determine the best solution based on the collected constraints.
- Result visualization: participants must be able to view the final solution.
- Error handling: users must be clearly notified in case of invalid or incomplete input.

2.2 System level

This category describes the core functionalities required for correct system operation:

- Load distribution: constraints must be distributed appropriately across nodes.
- Incremental computation of partial solutions: reduces computational pressure at the deadline.
- Final solution computation: the coordinator node notifies worker nodes when the deadline is reached and aggregates partial solutions to obtain the optimal one.

- Data persistence: data associated with events, constraints, and locations must remain persistent and recoverable even in case of system crashes.

Capitolo 3

Non-Functional Requirements

The system aims to guarantee high performance, scalability, resilience, and reliability, in order to deliver a robust and seamless user experience.

1. Performance: the system must handle a high load of users while maintaining low latency.
2. Deadline responsiveness: the final result must be available within a reasonable time after the deadline.
3. Horizontal scalability: the system must be able to scale by adding additional nodes.
4. Fault tolerance: in case of node failure, data must remain available; backup strategies must also be in place in the event of database crashes.
5. Availability: the system should guarantee at least 99% uptime.
6. Maintainability: the architecture should be modular and easy to maintain.
7. Usability: the user experience should be simple, clear, and intuitive.

Capitolo 4

Architecture

The application architecture, illustrated in Figure 4.1, is designed using a distributed system approach to ensure scalability, fault tolerance, and operational efficiency. The system combines heterogeneous technologies, each optimized for specific responsibilities.

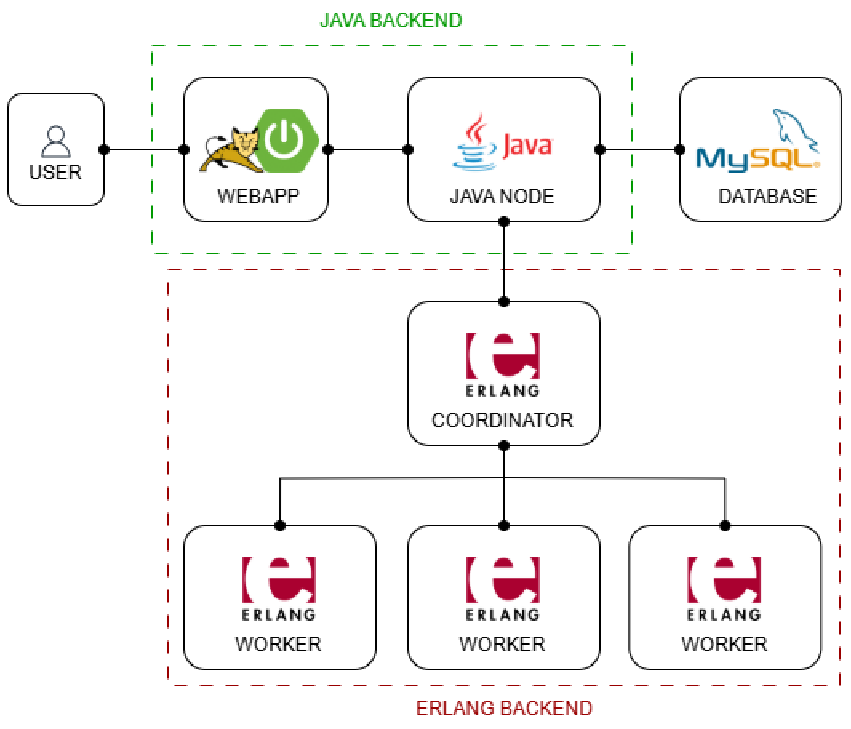

Figura 4.1: High-level logical architecture of the system. The diagram illustrates the separation between the Java Backend (in green), responsible for user management and persistence on MySQL, and the Erlang Backend (in red), dedicated to distributed computing. The Java Node acts as a bridge, sending computation requests to the Coordinator, which distributes them to the Workers.

The structure is organized into two macro-modules:

1. **Java Backend:** Based on Spring Boot and MySQL, it serves as the user entry point, managing the presentation (JSP Webapp), control logic, and structured data persistence.
2. **Erlang Backend:** A hierarchical cluster (Coordinator and Workers) that performs distributed computation and resource optimization in parallel.

4.1 Java Backend

The Java backend is implemented as a Spring Boot application. It acts as the interface between the end-user and the distributed system, handling authentication, data validation, persistence, and the presentation layer.

4.1.1 Architectural Components (MVC)

The system follows an extended MVC pattern to ensure modularity, maintainability, and scalability. Its main structural components are:

- **Model:** Domain classes that map directly to database tables. These classes leverage Lombok to automatically generate getters, setters, constructors, and other boilerplate code, improving readability and reducing manual effort.
- **View:** Implemented with JSP, managing the presentation logic, displaying data to users and capturing input through forms. JSPs interact with the application via Controllers, providing a responsive web interface.
- **Controller:** Mediates between View and Service layers, processing HTTP requests. It returns JSP views (`@Controller` annotation). Controllers process HTTP requests, invoke business logic, and orchestrate communication with the Service layer.
- **Repository:** Extends the JPA Repository interface for standard CRUD operations and additional custom queries for more complex data retrieval and persistence tasks.
- **Service:** Encapsulates core business logic, coordinating interactions between Controllers, Repositories, and Models while enforcing application rules and constraints.
- **DTO (Data Transfer Object):** Manages data exchanged between the web interface and backend, isolating internal domain models from the presentation layer. Examples include `EventRequest`, `EventResponse`, and `VincoloRequest`.

This architecture ensures a clear separation of concerns, facilitates testing and maintenance, and allows the system to scale efficiently while maintaining robust and consistent interactions with the database.

4.1.2 Functional Modules and Execution Flows

The backend logic is organized into three distinct functional modules. Below is the detailed sequential execution flow for each core operation.

4.1.2.1 User Module

The User Module handles the identity lifecycle through the following flows:

- **Authentication (`POST /auth/login`):**
    1. The `UserController` receives the login request containing credentials.
    2. It invokes the `UserService` to validate the input.
    3. The Service queries the `UserRepository` to verify the existence of the user and the correctness of the password.
    4. Upon success, the Controller initializes the HTTP Session and redirects the user to the homepage.
- **Registration (`POST /auth/register`):**
    1. The Controller accepts the registration form data (username, password and name).
    2. The Service creates a new `User` entity and persists it via the Repository.
    3. If successful, the user is redirected to the login page; otherwise, an error message is returned.

4.1.2.2 Events Module

This module provides endpoints for event management:

- **Creation (`POST /event/add`):** creates a new event with parameters such as event name, deadline, visibility (private/public), and a list of invited users.
    1. The Controller receives an `EventRequest` DTO.
    2. The `EventService` retrieves the creator's identity from the session.
    3. It creates a new `Event` entity, associating the creator and the invited participants.
    4. The entity is persisted in the database via the `EventRepository`.
- **Retrieval (`GET /event/view`):** retrieves events filtered by state (upcoming, expired) and visibility (private/public)
    1. The Controller receives a filter request.
    2. The Service queries the Repository to fetch the relevant events associated with the user.
    3. The results are mapped into `EventResponse` DTOs and rendered by the JSP view.

4.1.2.3 Constraints Module

This module enables associating constraints with a specific event:

- **Submission (`POST /constraints/add`):** adds a constraint specifying email, event ID, start and end times, location type, position, and budget range.
  1. The Controller receives a `VincoloRequest` DTO.
  2. It identifies the requesting user and forwards the data to the `VincoloService`.
  3. The `VincoloService` creates a `Vincolo` entity, associates it with the relevant User and Event, persists it using the `VincoloRepository`, and forwards the information to the Erlang coordinator node via the Java interface (JInterface).
  4. The Erlang coordinator then routes the constraints to the appropriate worker node for processing.

4.1.3 Database Schema

The application uses MySQL, interfaced via JDBC. The data model consists of the following relations:

- `User(id, email, psw, nome, cognome)`
- `Event(id, creatore_id, private, nome, descrizione, deadline)`
- `Event_partecipanti(event_id, user_id, data_iscrizione, stato)`
- `Vincolo(id, ora_inizio, ora_fine, tipo_luogo, budget_min, budget_max, posizione, event_id, user_id)`

4.1.4 Integration with the Erlang Coordinator

The Java backend also integrates a set of dedicated APIs for communication with the Erlang coordinator node:

- **Outgoing communication** (`ErlangService` Service: the Java node registers itself as `java_backend_node`, creates its mailbox, and, once the coordinator node is identified, sends messages related to event creation and constraint submission.
- **Incoming communication** (`ErlangReceiver` Thread): the results (i.e., the computed solution for a specific event) are received through a dedicated thread, which handles incoming messages asynchronously.

4.1.5 Webapp Module

User interaction is managed by the `webapp` submodule. This layer is responsible for handling HTTP requests, orchestrating calls to underlying services, and rendering the graphical user interface.

4.1.5.1 Technologies and Configuration

The core of the module consists of the Spring Boot 3.2.5 framework based on Java 21, selected for its ability to provide a robust and high-performance standalone execution environment. While Spring Boot typically favors modern template engines, JSP (JavaServer Pages) technology was adopted for this project to handle dynamic view generation. This choice necessitated a specific runtime environment configuration: since the Apache Tomcat server is integrated in embedded mode within the WAR package, the `tomcat-embed-jasper` library was included. This component is essential as it acts as a compilation engine, transforming JSP pages into Java Servlets at runtime - an operation the standard container would not perform natively.

The mapping between controllers and physical views is governed by specific configuration settings within the `application.properties` file. Spring's ViewResolver has been configured using the parameters `spring.mvc.view.prefix=/WEB-INF/jsp/` and `suffix=.jsp`. This configuration serves not only a structural purpose but also a security one: by positioning the source files within the protected `/WEB-INF` directory, direct access to the pages by web clients is prevented, thereby forcing every request to pass through the processing logic of the Spring Controllers.

4.1.5.2 Frontend Structure

The presentation logic utilizes JSTL (JavaServer Pages Standard Tag Library) and is segmented into:

- **Homepage (`index.jsp`):** A Single Page Interface for event management. It dynamically renders the Event Creation form, the Constraint Submission form, and the Results Table, which displays the optimal solution computed by the workers.
- **Authentication Views:** `login.jsp` and `registration.jsp` manage the user entry, employing reactive feedback mechanisms for error handling.

4.2 Erlang Backend

The Erlang backend constitutes the computational core of the distributed architecture. Its implementation leverages the actor model paradigms to parallelize computation and natively manage concurrency.

4.2.1 Project Structure and Supervision

The project is managed via Rebar3, adhering to OTP (Open Telecom Platform) standards. This structure allows the release to be bundled as a self-contained executable, simplifying containerization; mandates a strict project structure, thereby enhancing code maintainability; natively incorporates supervision mechanisms for fault tolerance, a critical requirement for distributed architectures.

A central design choice is the Dynamic Role Orchestration. A unified codebase is deployed to all nodes, but the `erlang_backend_sup` (Supervisor) determines the node's behavior at runtime based on the `role` environment variable:

- **Coordinator Role:** Starts the `erlang_coordinator` gen_server.
- **Worker Role:** Starts the `erlang_actor` process.

The supervision strategy is `one_for_one`, ensuring that individual process failures are recovered automatically without crashing the entire Virtual Machine.

4.2.2 Erlang Modules

4.2.2.1 Coordinator (`erlang_coordinator`)

The `erlang_coordinator` module functions as the central hub. Implemented as a `gen_server`, it acts as the intelligent gateway of the cluster, responsible for routing requests and aggregating results without performing direct computation on the data.

Dynamic Routing via Process Groups: The Coordinator relies on the `pg` (Process Groups) module for dynamic service discovery.

When a constraint message arrives (`{nuovo_vincolo, ..., Posizione}`), the Coordinator performs a Semantic Routing step:

1. **Group Resolution:** It converts the target zone string (e.g., "NORD") into a specific process group atom (e.g., `'gruppo_nord'`).
2. **Member Lookup:** It queries `pg:get_members(Group)` to retrieve the PIDs of active workers responsible for that zone.
3. **Forwarding:** The message is unicast to an available worker. If the group is empty, the system logs a warning but remains stable.

Architectural Advantages: Adopting `pg` offers substantial benefits over a static registry:

- **Elasticity:** Worker nodes can be added or removed at runtime to handle load spikes without requiring a coordinator restart or configuration reload.

- **Fault Tolerance:** The Erlang runtime automatically monitors process health. If a Worker crashes, its PID is instantly removed from the group, ensuring the Coordinator never routes messages to dead processes (prevention of Stale PIDs).
- **Decoupling:** The Coordinator logic remains purely semantic (routing to "North"), abstracting away the physical network topology and IP addresses.

Active Cluster Monitoring: Beyond semantic routing, the Coordinator ensures visibility over the physical cluster topology. Upon initialization, it explicitly activates network kernel monitoring via `net_kernel:monitor_nodes(true)`. This mechanism allows the `gen_server` to trap `nodedown` and `nodeup` system events via the `handle_info/2` callback. Consequently, the Coordinator can log high-priority alarms immediately upon any Worker disconnection, providing real-time observability of the infrastructure's health status alongside the logical group management provided by `pg`.

Aggregation Logic (Weighted Map-Reduce): The global optimization process is managed via an asynchronous Map-Reduce pattern, utilizing an internal state map (`sessioni`) to track open requests.

- **Broadcast:** Upon receiving the `calcola_ottimo_globale` trigger, the Coordinator identifies all unique nodes in the global group `'tutti_i_worker'` and broadcasts a `richiedi_parziale` signal.
- **Dynamic Quorum:** The expected number of responses is calculated at runtime based on the currently active nodes, ensuring the system can proceed even in a degraded state (Partial Availability).
- **Weighted Comparison:** As partial results arrive, the Coordinator executes a weighted comparison algorithm. It evaluates the candidate venues based on a composite metric:

  $$\text{Impact} = \text{Score} \times \text{Hits (User Count)}$$

  This ensures the algorithm privileges solutions that satisfy the largest number of users, not just those with the highest isolated quality score.

4.2.2.2 Worker (`erlang_actor`)

The `erlang_actor` module implements the computational units. Adhering to the shared-nothing model, each Worker is a plain process governed by a recursive `loop/1` function, designed for high throughput.

Self-Configuration and Data Locality: The Worker exhibits "location-aware" behavior. During the `init_internal` phase:

1. **Zone Detection:** It parses its own node name (e.g., `worker_nord@ip`) to extract the geographic zone ("nord", "centro", or "sud").
2. **Group Membership:** Based on the extracted zone, it automatically joins the relevant `pg` groups (e.g., `'gruppo_nord'` and `'tutti_i_worker'`).
3. **Selective Data Loading:** It loads into memory only the subset of venues relevant to its jurisdiction. This hardcoded partitioning ensures strict Data Locality.

Reactive Behavior and Self-Healing. The process loop handles three main responsibilities:

- **Network Self-Healing (`check_connection`):** The Worker maintains a connetcion loop. It periodically checks the connection to the Coordinator node. If the connection is lost, the Worker actively initiates a `net_adm:ping` to re-establish the mesh, ensuring automatic recovery from network partitions.
- **Real-Time Processing (`nuovo_vincolo`):** Upon receiving a constraint, the worker immediately updates the statistical model in Mnesia via `db_manager:update_stats`. It recalculates quality metrics and overlaps for all managed venues, keeping the "Best Local Candidate" constantly up-to-date.
- **Query Execution (`richiedi_parziale`):** When requested, the worker avoids heavy computation. It simply queries the pre-calculated optimum from Mnesia (`db_manager:find_best_locale_live`) and returns the result tuple to the Coordinator.

4.2.3 Data Structure and Persistence (Mnesia)

To guarantee real-time performance while ensuring data durability, the system utilizes Mnesia, Erlang's native distributed DBMS.

Within the `db_manager` module, Mnesia is configured to use `disc_copies` for all tables. This storage mode keeps data in RAM for fast read/write access during computation but synchronously logs transactions to disk, ensuring that the cluster state survives node restarts.

4.2.3.1 Database Schema

The data is structured into five specific tables, designed to store pre-aggregated statistics rather than raw constraints:

- `locale`: A static registry holding venue details (ID, name, type, average price, opening hours).

- `stats_locale`: The core table for quality aggregation. It maps the pair `{EventId, VenueId}` to the cumulative quality score and the number of votes received. This allows the system to compute the average quality in $O(1)$ without scanning historical logs.
- `slot_temporale`: Operates as a temporal histogram. It maps the triplet `{EventId, VenueId, Hour}` to a counter. This structure is used to identify the time slot with the maximum user overlap ("Hits").
- `global_state`: A key-value store used to track global counters, such as the total number of participants for a specific `EventId` in that node, which is the denominator required for the I_{tot} algorithm.
- `best_solution`: Stores the current best local candidate for each event, used for caching and rapid retrieval.

4.2.3.2 Query and Reduction Logic

The worker does not perform heavy computation upon request. Instead, the logic is split into:

1. **Write Path (Incremental Update):** When a constraint arrives, `update_stats/4` updates the running totals in `stats_locale` and increments the specific hourly buckets in `slot_temporale` inside an atomic transaction.
2. **Read Path (Local Reduction):** When the Coordinator requests a partial solution, the function `find_best_locale_live/1` is invoked. It iterates over the venues, calculates the score I_{tot} using the pre-aggregated data, and returns a tuple `{BestRecord, Hits}`. Returning the "Hits" (number of users in the best slot) alongside the record is crucial, as it allows the Coordinator to perform a weighted comparison favoring popular venues.

Capitolo 5

System Workflow

This chapter details the operational dynamics of the system, illustrating the end-to-end data flow from user interaction to the final computation of the optimal event schedule. While the previous chapter detailed the internal component interaction within the Java Backend, this section focuses on the distributed workflow across the hybrid architecture. The process is architected around two distinct, asynchronous phases: the **Submission Phase** and the **Optimization Phase**.

5.1 Phase 1: Constraint Submission and Real-Time Distribution

The first phase concerns the ingestion of user preferences. This process ensures immediate data persistence in the relational database while simultaneously populating the distributed cluster's memory.

5.1.1 Ingestion and Asynchronous Dispatch

When a user submits constraints via the Dashboard, the Java Backend executes the internal logic described in Section 4.1.2. Crucially, the workflow extends beyond the Java boundary:

1. **Persistence:** The system ensures the constraint is durably stored in MySQL for recovery purposes.

2. **Bridge to Erlang:** Immediately upon storage, the Java Node acts as a producer. It maps the constraint object into an Erlang tuple (e.g., `{nuovo_vincolo, ...}`) and dispatches it to the Coordinator's mailbox. This operation is performed in a *fire-and-forget* manner, allowing the web request to complete without waiting for the cluster's acknowledgment.

5.1.2 Routing and In-Memory Update

Upon receiving the message, the Erlang Coordinator performs a logic routing step. It inspects the geographic tag of the constraint and forwards the tuple to the competent Worker Node via the internal distribution protocol.

The Worker Node processes the message by updating its internal **Mnesia** RAM tables. It incrementally recalculates the aggregate statistics for the involved venues. This *eager aggregation* strategy ensures that the computational cost is distributed over the submission period rather than concentrated at the deadline.

5.2 Phase 2: Global Optimization and Result Feedback

The second phase is time-triggered. It represents the transition from data collection to decision-making.

5.2.1 Deadline Trigger and Map-Reduce

The `DeadlineManager` in the Java Backend acts as the temporal authority. When an event deadline expires:

1. **Trigger:** Java sends a `{calcola_ottimo_globale, EventId}` message to the Coordinator.
2. **Map (Broadcast):** The Coordinator broadcasts a request for partial solutions to all active Worker nodes via the `pg` process group.
3. **Reduce (Local):** Each Worker queries its local Mnesia instance to select the single best venue among those it manages (maximizing the I_{tot} score) and returns it to the Coordinator.
4. **Aggregation:** The Coordinator collects the responses, compares the partial local optima, and selects the absolute global winner.

5.2.2 Finalization

Once the global optimum is identified, the Coordinator sends the result back to the Java Backend using the native message format: `{risultato_finale, EventId, BestSolution}`.

A dedicated listener thread on the Java side intercepts this message and updates the `Event` entity in MySQL, changing its status to "Concluded" and storing the winning venue details. The result is immediately available for visualization in the User Dashboard.

Capitolo 6

Synchronization, Coordination, and Communication Issues

The design of a hybrid distributed system, composed of a Java backend and an Erlang computation cluster, necessitated a thorough analysis of the critical issues inherent to concurrent systems. This chapter analyzes the challenges encountered in three fundamental areas - Synchronization, Coordination, and Communication - and the architectural solutions adopted to address them.

6.1 Synchronization Problems and Solutions

Synchronization in a distributed system concerns the assurance that events and data remain consistent across time and space, despite the absence of a shared global clock and the presence of concurrent access.

6.1.1 Deadline Synchronization and Temporal Management

The system implements a hybrid distributed scheduling mechanism to ensure that the global optimum calculation occurs exactly when the voting period (Deadline) for an event ends.

The process consists of two main flows: scheduling upon creation and daily recovery via the `DeadlineManager`.

6.1.1.1 Real-Time Scheduling (Event Creation)

When a user creates a new event via the web interface, the system instantaneously calculates the time remaining until the deadline and communicates it to the Erlang node.

1. **Delay Calculation:** In the `EventController`, immediately after saving the event to the DB, the temporal difference between the current instant (`LocalDateTime.now()`) and the event's deadline is calculated.

2. **Message Dispatch to Erlang:** The `ErlangService` sends an asynchronous tuple to the Coordinator containing the event ID and the delay in milliseconds: `{deadline, EventId, DelayInMillis}`.

3. **Timer Activation (Erlang Side):** The `erlang_coordinator` process receives the message, verifies that an active timer for that event does not already exist, and starts an internal timer using the `erlang:send_after/3` primitive. Upon expiration, the process sends the message `{calcola_ottimo_globale, EventId}` to itself.

6.1.1.2 Fault Tolerance and Rescheduling (DeadlineManager)

Since timers in Erlang reside in the volatile memory of the `gen_server` process, a restart or crash of the Coordinator node would result in the loss of all pending deadlines. To mitigate this risk, the Java system acts as the persistent "source of truth."

The `DeadlineManager` component executes a scheduled task (`@Scheduled`) acting as a **recovery and synchronization** mechanism:

- **Periodic Scanning:** The system periodically analyzes (configured via CRON, e.g., hourly or at midnight) the events present in the database.

- **Identification of Daily Events:** It retrieves the list of all events whose deadline falls within the current day (between `startOfDay` and `endOfDay`).

- **Rescheduling:** For each event found:

  1. It verifies if the deadline is still in the future via `isAfter(now)`.
  2. It recalculates the updated remaining delay.
  3. It resends the scheduling message (`sendTimerRequest`) to the Erlang node.

To prevent the recovery mechanism from duplicating timers (which could create race conditions or double calculations), the `erlang_coordinator` implements idempotent logic:

- It maintains an internal map called `timers`.

- Upon receiving a `{deadline, ...}` request, it checks if the Event ID is already present in the map.

- If the timer already exists, the request is ignored and logged as such (`[IGNORATO] Timer per Evento X già presente`), ensuring that the recovery from Java does not interfere with timers that are already running correctly.

6.1.2 Concurrent Data Integrity (Mnesia)

The Challenge: Worker nodes are subject to a high level of concurrency. During peak loads, hundreds of constraints may attempt to update statistics for the same venue simultaneously. Without a concurrency control mechanism, Race Conditions would occur (e.g., Read-Modify-Write conflicts), corrupting aggregated data like user counters or score summations, thus compromising the validity of the final result.

The Solution (Atomic Transactions): Local state synchronization is entrusted to Mnesia, Erlang's real-time DBMS. The system encapsulates every update operation within a functional transaction:

$$\text{Read State} \rightarrow \text{Calculate New Average} \rightarrow \text{Write State}$$

By wrapping this logic in `mnesia:transaction`, the system guarantees ACID properties at the node level. Operations are serialized internally by the database engine, ensuring that venue state remains consistent even under stress testing, without the need to implement complex and error-prone application-level locking mechanisms.

6.2 Coordination Problems and Solutions

Coordination concerns the logical orchestration of nodes: how work is distributed, how results are aggregated, and how the cluster topology is managed.

6.2.1 Sharding Strategy and Data Locality

The Challenge: Centralizing all data (venues and constraints) on a single node would create a computational bottleneck, preventing the system from scaling. Conversely, a random distribution (e.g., Round Robin) would entail excessive network latency, as calculating a score for a "North" venue might require fetching data located on a "South" node.

The Solution (Geo-Sharding): The system implements a coordination strategy based on Geographic Partitioning. Venues are statically distributed across competent Worker nodes (e.g., North, Center, South) to maximize Data Locality.

Coherently, the Coordinator routes incoming user constraints to the specific node holding the data for that zone. This approach is effective because:

- **It eliminates contention:** Each node works on a disjoint dataset in total parallelism.
- **It eliminates I/O latency:** Score calculation occurs entirely in-memory, accessing venue data resident on the same node, without the need for remote network queries (Shared-Nothing Architecture).

6.2.2 Dynamic Topology Management and Service Discovery (pg Module)

The Challenge: In a scalable distributed architecture, cluster topology cannot be statically defined. Worker nodes must be able to join or leave dynamically to respond to load variations. Hardcoding lists of PIDs (Process IDs) or IP addresses within the Coordinator's code would render the system rigid, requiring a full service restart for every infrastructural change.

The Solution (Process Groups and Active Discovery): The system addresses this issue by adopting a Dynamic Service Discovery mechanism based on Erlang's standard `pg` (Process Groups) module. This approach decouples the Coordinator from the physical identity of the Workers, introducing an abstraction layer based on "semantic groups".

The connection and registration protocol proceeds through the following phases:

1. **Bootstrap and Mesh Connection:** Upon startup, each Worker node executes an *active polling* cycle to connect to the known Coordinator node (via `net_adm:ping`). Once the TCP handshake is established, the node enters the distributed cluster.

2. **Auto-Identification:** The Worker parses its own node name (e.g., `worker_nord@ip`) to autonomously determine its designated zone.

3. **Group Registration (Join):** Using `pg:join/2`, the Worker subscribes to two distributed groups:

   - *Geographic Group* (e.g., `'gruppo_nord'`): For routing zone-specific tasks.
   - *Global Group* (`'tutti_i_worker'`): For broadcast operations (Global Optimization).

4. **Transparent Lookup:** The Coordinator maintains no persistent state regarding active Workers. It queries the `pg` manager at runtime (e.g., `pg:get_members('gruppo_nord')`) to obtain the updated list of available processes. This ensures intrinsic load balancing and fault tolerance: if a node fails, it is automatically removed from the group by the Erlang runtime.

6.3 Communication Problems and Solutions

Communication concerns the protocols and transport mechanisms for data exchange between the system's heterogeneous components.

6.3.1 High-Performance Java-Erlang Interoperability

The Challenge: Facilitating dialogue between two different ecosystems (JVM and BEAM) often involves "bridge" protocols like HTTP/REST with JSON payloads. While standard, this solution introduces significant overhead due to text-based serialization/deserialization and the establishment of new TCP connections for each request, making it unsuitable for a high-throughput real-time system.

The Solution (JInterface & Native Messaging): The issue was resolved by implementing native binary communication via JInterface. The Java backend is configured as an Erlang Node, allowing it to:

- Register itself in the Erlang Port Mapper Daemon (EPMD).
- Encapsulate data directly into Erlang Tuples (`OtpErlangTuple`).
- Communicate via the native distribution protocol.

This results in maximum efficiency, as data is transmitted in binary format without intermediate text conversion.

6.3.2 Asynchrony and Non-Blocking I/O

The Challenge: A synchronous communication model (Blocking Request-Response) between Java and the Coordinator would risk blocking the limited Web Server threads while awaiting the completion of processing by the Erlang cluster. This would render the user interface unresponsive under high load.

The Solution (Asynchronous Message Passing): Communication was designed to be fully asynchronous. Java acts as a producer, sending messages to the Coordinator's mailbox (a *fire-and-forget* operation) and immediately freeing the thread. Within the cluster, the use of asynchronous Message Passing between Coordinator and Workers ensures the system remains reactive, preventing a single slow node from blocking the entire processing pipeline (Non-Blocking I/O).

Capitolo 7

Recovery Mechanisms

In distributed systems, failures are not exceptional events but rather expected occurrences. The architecture of this project implements a "Let It Crash" philosophy, typical of the Erlang ecosystem, supported by active self-healing mechanisms. This chapter details the strategies adopted to ensure system resilience against node failures, network partitions, and process crashes.

7.1 Network Self-Healing and Partition Recovery

A critical vulnerability in distributed clusters is the "Split-Brain" scenario or temporary network partitioning. The system implements an active recovery strategy on the Worker side to mitigate this.

7.1.1 Active Connection Polling

As implemented in the `erlang_actor` module, each Worker node maintains a dedicated connection loop (`check_connection`). Instead of passively waiting for instructions, the Worker actively verifies its inclusion in the cluster:

- **Detection:** Periodically, the worker checks if the Coordinator node is present in its connected nodes list.
- **Recovery Action:** If the connection is lost (e.g., due to a Coordinator restart or network fluctuation), the Worker invokes `net_adm:ping(?COORDINATOR_NODE)`.
- **Outcome:**
  - `pong`: The connection is re-established, and the node automatically continues its operation.
  - `pang`: The Coordinator is unreachable; the worker waits and retries in the next cycle.

This mechanism ensures that the cluster topology automatically converges to a consistent state without human intervention after a fault.

7.1.2 Coordinator-Side Monitoring

While Workers actively poll for connection, the Coordinator implements a passive monitoring strategy using `net_kernel:monitor_nodes(true)`. This ensures that any node failure is immediately detected and logged to the console ("!!! ALLARME"), allowing system administrators to promptly identify network partitions or node crashes.

7.2 Process Resilience and Dynamic Topology

At the node level, resilience is guaranteed by the combination of OTP Supervision Trees and Dynamic Service Discovery.

7.2.1 Supervisor Strategy

The system employs a `one_for_one` restart strategy. If a Worker or Coordinator process crashes (e.g., due to a software bug or an unhandled exception):

1. The Supervisor intercepts the exit signal.
2. It immediately restarts only the failed process, leaving the rest of the Erlang VM unaffected.
3. Upon restart, the new process executes its `init` function, reloads the necessary data from Mnesia, and re-registers itself with the `pg` groups.

7.2.2 Prevention of Stale PIDs

A common issue in recovery is the "Stale PID" problem, where a coordinator tries to message a process that has crashed and restarted (thus acquiring a new PID).

By utilizing the `pg` module for routing, the system eliminates this risk. The Coordinator never caches Worker PIDs. Instead, for every request (e.g., `nuovo_vincolo`), it queries the group membership at runtime. Since the Erlang runtime automatically removes dead processes from `pg` groups, the Coordinator effectively "sees" only the healthy nodes, routing messages solely to active processes.

7.3 Fault Tolerance in Global Computation

The Global Optimization phase is designed to support Degraded Service. The system prefers returning a partial result over a total failure (deadlock).

7.3.1 Dynamic Quorum

In the `erlang_coordinator`, the map-reduce algorithm does not wait for a fixed, hardcoded number of responses. Instead, it calculates the expected number of replies dynamically based on the currently active members of the `'tutti_i_worker'` group:

```
UniqueWorkers = lists:usort(pg:get_members('tutti_i_worker')),
NumExpected = length(UniqueWorkers).
```

If a Worker node is down during the deadline expiration:

1. It is excluded from the `UniqueWorkers` list.
2. The Coordinator proceeds to aggregate results from the surviving nodes.
3. The final solution is computed based on the available dataset.

This ensures that the failure of a specific geographic shard (e.g., "South" node down) does not block the event organization for users in the "North" or "Center".

7.4 Data Consistency and Persistence

Finally, data integrity during crashes is protected by Mnesia's transaction system, configured in the `db_manager` module.

7.4.1 Hybrid Storage (Disc Copies)

The tables `stats_locale`, `best_solution`, and `global_state` are configured with the `disc_copies` option.

- **Operation:** Data is kept in RAM for speed but synchronously logged to disk.
- **Recovery:** If a Worker node crashes and restarts, it reloads the statistical state from the disk. This prevents the loss of accumulated data (e.g., user votes and calculated averages) that were processed prior to the crash, ensuring the "Best Local Candidate" remains valid.

7.4.2 Atomic Transactions

The system wraps updates in `mnesia:transaction`. This ensures Atomicity: if a process crashes in the middle of an update (e.g., incrementing total users but failing to update the venue score), the entire operation is rolled back, preventing the database from reaching an inconsistent or corrupted state.

Capitolo 8

Conclusion

The project successfully achieved the objective of developing a hybrid distributed system for collaborative event organization, demonstrating how the integration of heterogeneous technologies can effectively resolve complex coordination and optimization problems.

The proposed architecture, which combines the management robustness of a **Java** backend with the concurrent computational power of an **Erlang** cluster, validated several key design choices:

1. **Efficiency of the Hybrid Model**: The separation of concerns allowed leveraging the strengths of both environments: Java for structured user management and relational persistence, and Erlang for the parallel execution of the weighted Map-Reduce optimization algorithm.

2. **Flexibility and Dynamic Discovery**: Moving away from static topologies in favor of the `pg` (Process Groups) module proved decisive. The implementation of *location-aware* logic in Workers, capable of deducing their specific zone (e.g., "North", "South") directly from the node name at startup, eliminated the need for rigid centralized configurations, significantly simplifying deployment and horizontal scalability.

3. **Resilience and Persistence**: The use of **Mnesia** with the `disc_copies` strategy ensured that the aggregated system state (votes and venue statistics) survived crashes and restarts, fulfilling fault tolerance requirements.

4. **Self-Configuration**: The *Active Connection Polling* mechanism implemented in the Workers effectively resolved temporary network partition issues, allowing the cluster to self-heal without human intervention.

In summary, the system represents a scalable and reactive solution, capable of dynamically adapting to workload variations while maintaining data consistency and high service availability.

Capitolo 9

Future Works

Although the current system fully meets both functional and non-functional requirements, the architectural analysis highlighted areas for improvement that could further enhance the cluster's degree of distribution and autonomy.

9.1 Dynamic Role Assignment and Leader Election

Currently, the distinction between a *Coordinator* node and a *Worker* node is statically determined at startup via environment variables managed by the Supervisor. While stable, this approach introduces a logical *Single Point of Failure*: if the node designated as Coordinator fails, the system requires an explicit restart of an instance with that specific role.

A future development involves implementing a **Leader Election** protocol (based on consensus algorithms like Raft or Paxos implemented in Erlang).

- **New Paradigm**: All nodes would start as generic "Peers" without predefined roles.
- **Mechanism**: Upon startup, the cluster would autonomously elect a Leader to assume the Coordinator role. The remaining nodes would operate as Workers.
- **Advantage**: In the event of a Coordinator crash, the surviving nodes would immediately elect a new Leader, ensuring real and transparent *High Availability*.

9.2 Dynamic Sharding and Consistent Hashing

The current *Geo-Sharding* strategy maps zones (North, Center, South) to Workers deterministically based on the node name. This could cause load imbalances (Hotspots) if a specific zone receives a disproportionate volume of events compared to others.

The natural evolution is the adoption of **Consistent Hashing**:

- **Distribution**: Constraints would no longer be routed solely based on geographic zone but distributed across a virtual hash ring.
- **Elasticity**: This would allow adding or removing Worker nodes on the fly without adhering to a rigid geographic topology. The system would automatically redistribute only a portion of the data (the keys involved in re-hashing) to the new nodes, optimizing hardware resource usage.

9.3 Full Decentralization (Peer-to-Peer)

To completely eliminate the central coordination bottleneck during the global aggregation phase, a **fully decentralized** architecture could be explored.

- **Gossip Protocols**: Instead of sending partial results to a single Coordinator (centralized Map-Reduce pattern), Workers could use *Gossip* protocols to disseminate and aggregate partial results among neighbors.
- **Convergence**: Each node would independently converge towards the global optimal solution, theoretically allowing the system to scale indefinitely without overloading a single aggregator node.

Capitolo 10

Appendix A

Optimization Algorithm

The logical core of the system resides in the `utils.erl` module, which implements the algorithm to determine the "suitability" of a venue. The problem is modeled as a maximization function of an aggregate index I_{tot}. The calculation occurs in two phases:

Phase 1: User Quality ($P_{quality}$) For each individual user constraint regarding a venue, an affinity score is calculated based on two weighted factors:

- **Semantic Similarity ($W_{sim} = 0.6$):** Evaluates how well the venue type (e.g., "Pub") matches the user's preference. If they coincide, the value is 1.0; if they are similar (e.g., Pub-Brewery), it is 0.9; otherwise, it drops to 0.1.
- **Budget Compatibility ($W_{budget} = 0.4$):** Checks if the venue's average price falls within the user's [Min, Max] range. If it is within the range, the score is high; if outside, it decays linearly based on the distance from the margin.

$$P_{quality} = (0.6 \cdot Similarity) + (0.4 \cdot Budget)$$

Phase 2: Total Group Index (I_{tot}) To determine the final venue score for the entire group of participants, the algorithm aggregates individual qualities and introduces the "Participation" factor:

- **Participation Index (I_{part}):** This is the percentage of users who can be present simultaneously during the venue's various time slots. The opening hours are divided into time slots, and participation is calculated for each slot. If a venue is universally liked but open when no one is available, this index will be low.
$$I_{part} = \frac{\text{Max Users in Slot}}{\text{Total Event Users}}$$
- **Average Quality ($AvgQuality$):** The average of the $P_{quality}$ scores of all users.

The final index is an equally weighted average between the aggregation capacity and the average quality:

$$I_{tot} = (0.5 \cdot I_{part}) + (0.5 \cdot AvgQuality)$$

This formula ensures that the algorithm privileges compromise solutions that maximize group presence, rather than solutions that are excellent for a few but inconvenient for many.