UNIVERSITY OF PISA

MASTER'S DEGREE IN COMPUTER ENGINEERING

Distributed Systems and Middleware Technologies

Whenly

Professors:

Alessio Bechini

José Luis Corcuera Bárcena

Group Members:

Alessandro Ascani

Giovanni Ligato

Giovanni Enrico Loni

ACADEMIC YEAR 2024/2025

Index

1. Introduction

Scheduling events and meetings in groups, especially when participants have conflicting availability, is a challenging task. Existing tools often lack scalability, robustness, and an effective mechanism to handle diverse constraints or ensure fault tolerance. These shortcomings become even more pronounced in scenarios where many participants or large numbers of events need to be processed simultaneously.

Whenly is a distributed event scheduling application designed to resolve these challenges. It provides a fault-tolerant, scalable system for scheduling events by collecting and reconciling constraints from participants. By leveraging a distributed architecture, the system ensures efficient computation and coordination of scheduling tasks while maintaining high availability and reliability.

The application is designed to facilitate efficient event scheduling through a user-friendly interface and robust backend processes. Below is a summary of how different users interact with the system to achieve seamless event coordination:

- *Event Organizers* create events with deadlines and invite participants to provide their availability and constraints.
- *Participants* submit their constraints, such as time preferences.

The system dynamically computes partial solutions for each event and finalizes a schedule at the event's deadline, ensuring that the best possible schedule is derived based on all submitted constraints.

2. Functional Requirements

The functional requirements define the core functionalities that the *Whenly* application must provide to achieve its objectives. These requirements are user-facing and system-focused to ensure a seamless scheduling experience.

2.1. User-Level Functionalities

User-Level Functionalities are the features that users interact with directly to manage events. In particular, the following functionalities are essential for the successful operation of the application.

1. User Authentication:
   - Users must be able to sign up, log in, and log out securely.
2. Event Management:
   - Event organizers can create events, specifying:
     - A deadline for participant responses.
     - Constraints, such as the preferred duration or other event-related conditions.
     -
   - Participants can join events via shared event IDs and add constraints to events they are part of.
3. Constraint Submission:
   - Participants must be able to submit availability constraints (e.g., preferred times).
4. Scheduling Algorithm Execution:
   - The system must compute a final schedule for events when the deadline expires by reconciling all submitted constraints.
5. Result Visualization:
   - Event organizers and participants should be able to view:
     - The finalized schedule once computed.
     - A notification if no valid schedule could be generated.
6. Error Handling:
   - Users should receive clear error messages for invalid inputs (e.g., incorrect credentials).
   - The system should provide feedback if a schedule cannot be created due to conflicts.

2.2. System-Level Functionalities

System-Level Functionalities are the core operations that the system must perform to manage events, constraints, and scheduling tasks effectively. These functionalities ensure the smooth operation of the application.

1. Distributed Task Allocation:

- The *Backend Node* must distribute event constraints to *Event Server Nodes* in a circular manner for balanced workload distribution.

2. Partial Solution Updates:
   - Each *Event Server Node* must dynamically update the partial solution for an event when a new constraint is received.
3. Distributed Final Schedule Computation:
   - The *Event Server Node* responsible for the event's deadline must initiate the distributed algorithm to finalize the schedule.
4. Persistent Data Storage:
   - The *Database Node* must store:
     - User credentials securely.
     - Event metadata and constraints.
     - Final schedules or failure notifications for completed events.

3. Non-Functional Requirements

The non-functional requirements establish performance, scalability, and reliability standards for the system to ensure user satisfaction and system robustness.

1. Performance:
   - The system must handle up to 10000 concurrent users with minimal latency (<1 second for responses to WebApp requests).
   - All final schedules must be available within 1 minute of the event deadline.
2. Scalability:
   - The system must scale horizontally by adding more *Event Server Nodes* to handle increased workloads.
3. Fault Tolerance:
   - In case of an *Event Server Node* failure:
     - No data should be lost.
     - Tasks assigned to the failed node must be reassigned to healthy nodes without disrupting ongoing processes.
   - The *Database Node* must ensure consistent backups for critical data.
4. Security:
   - User credentials must be securely stored using hashing mechanisms.
5. Availability:
   - The system should maintain 99.9% uptime, ensuring continuous availability for users.
6. Maintainability:
   - The modular architecture must allow for the independent maintenance and upgrading of components (e.g., WebApp, Backend, Event Servers).
7. Usability:
   - The WebApp interface must be intuitive and responsive, ensuring a positive user experience across devices.

4. Architecture

The architecture of the application, depicted in Figure 1, is designed with a distributed system approach to ensure scalability, fault tolerance, and efficient task handling. It leverages a combination of technologies optimized for distinct responsibilities, as detailed in the next sections.

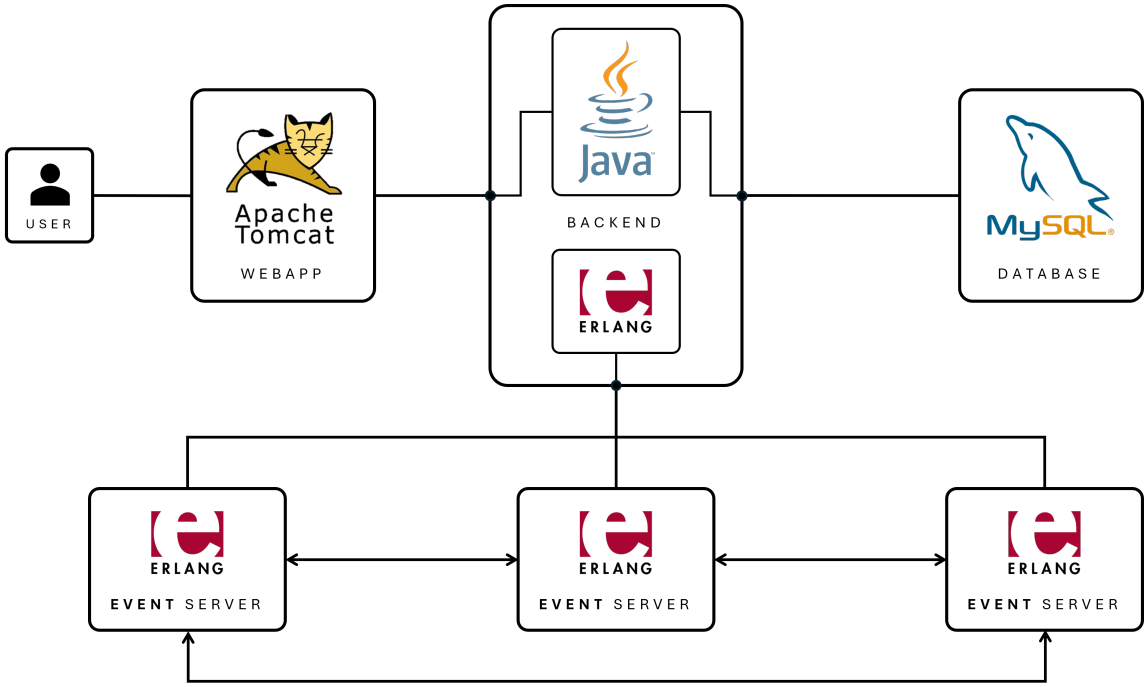

Figure 1: A high-level architecture diagram of Whenly, showcasing the modular separation of WebApp, Backend, Database, and distributed Event Server nodes for scalable and fault-tolerant event scheduling.

4.1. WebApp Node

The WebApp Node provides the user interface and is implemented using *Java Server Pages* (JSP) hosted on an *Apache Tomcat* server, implementing the Frontend of the application. Each JSP page communicates with the Backend Node through dedicated *servlets* that send HTTP requests to the REST API provided by the Backend. The primary functionalities of the WebApp are as follows:

- **Homepage**: Displays the login page by default. The login (and registration) process is managed via forms linked to specific servlets. When a user submits credentials via a POST request, the servlet forwards this to the Backend. Based on the response, the user is either redirected to their User Area or shown an error message.
- **User Area**: Accessible after successful authentication, this page retrieves event data (created, participated, or pending events) via a GET request through a servlet. Event data is displayed in a tabular format. Users can initiate event creation or add constraints directly from this

page.

- **Event Management**: For event creation, the corresponding servlet sends a POST request to the Backend. Once an event is created, the Backend returns an event ID which is displayed to the user. This ID can be used later for adding constraints. Users can add constraints to events using a specified format (each constraint is provided as a pair of timestamps in the format `YYYY-MM-DD HH:mm` for lower and upper bounds). After an event's deadline, the final event result is displayed using the same time interval format.

4.2. Backend Node

The Backend Node orchestrates core application logic by bridging user interactions with persistent storage and distributed event scheduling. It is composed of two main components: the Java Backend Component and the Erlang Backend Component.

4.2.1. Java Backend Component

The Java Backend Component is implemented as a *Spring Boot* application that exposes REST APIs for the WebApp. It handles core business logic, processes user requests, and coordinates event scheduling operations. The application is modular, featuring several dedicated modules.

- **Users Module:** Manages user registration and authentication:
  - `POST /users/signup`: Registers a new user with `username` and `password`.
  - `POST /users/login`: Authenticates a user with the provided credentials.
- **Events Module:** Provides endpoints for event management:
  - `POST /events/create`: Creates a new event, requiring parameters such as `username`, `deadline`, and `eventName`.
  - `GET /events/{id}`: Retrieves a specific event by its ID.
  - `GET /events/user/{username}`: Retrieves all events associated with a user (created, participated in, or pending).
- **Constraints Module:** Handles the submission of scheduling constraints:
  - `POST /constraints/add`: Adds a constraint to an event, with parameters `username`, `eventId`, and `constraint` (formatted as a pair of timestamps `YYYY-MM-DD HH:mm` representing lower and upper bounds).
- **Supporting Modules:** Additional modules include:
  - `sharedStringList`: Implements mutex access to shared lists.
  - `ErlangMessageHandler`: Processes incoming messages from the Erlang Backend.

Furthermore, the Java Backend Component incorporates the **ErlangBackendAPI** service—a dedicated Java service responsible for managing communication with the Erlang Backend Component. This service handles:

- **Outgoing Communication:** Sending event creation and constraint requests to a designated Erlang node, along with details about which node should handle final solution computation.
- **Incoming Communication:** Receiving final scheduling results and node status notifications from the Erlang Backend, then forwarding these to the appropriate Spring Boot modules (e.g., updating the `final_result` in the event table).

Communication between Java and Erlang is achieved using JInterface, which creates an Erlang node and mailbox within the Java environment, ensuring robust bidirectional message passing.

4.2.2. Erlang Backend Component

The Erlang Backend Component runs on the same node as the Java Backend, serving as the intermediary between Java and the distributed Erlang Event Server Nodes. Developed as a rebar3 OTP application, this component provides robust, bidirectional communication and continuous system monitoring. The Erlang Backend is responsible for the following tasks:

- **API Exposure:** Provides APIs for the Java Backend to send event creation and constraint requests directly to the appropriate Event Server Node.
- **Node Monitoring:** Continuously monitors the availability of all Event Server Nodes, notifying the Java Backend of any changes (e.g., node failures or new node additions) to support dynamic load balancing and fault recovery.
- **Final Schedule Handling:** Receives final scheduling results or failure notifications from the Coordinator Module within the Event Server Nodes, then forwards these results to the Java Backend for storage and user notification.

Implementation Details:

- The Erlang Backend Component is structured with a supervisor that starts the `erlang_backend_api` process (implemented as a gen_server) using a one_for_one strategy to ensure automatic recovery from failures.
- It employs a configuration reader module to manage settings via JSON.
- Interaction with the Java Backend is facilitated by JInterface, which allows seamless message passing through an Erlang node and mailbox integrated into the Java environment.

4.3. Database Node

The Database Node provides centralized, persistent storage for critical system data and is implemented using MySQL on a separate physical node. It ensures data integrity and supports recovery in case of failures. The information stored in the database includes:

- **User Credentials:** The `users` table stores user credentials, maintaining fields for `username` and `password`.
- **Event Metadata:** The `event` table captures event metadata, including:
  - `id`: Primary key for the event.
  - `creator_username`: Username of the event creator.
  - `deadline`: Timestamp indicating when final schedule computation should occur.
  - `erlang_node_ip`: IP address of the Erlang node designated to compute the final schedule (also used in the recovery process).
  - `event_name`: Name of the event.
  - `final_result`: The final schedule result, which can be:
    - `null` if the event is still in progress,
    - A pair of timestamp strings if solved,
    - `undefined` if no constraints were added, or

    * `no_solution` if no valid schedule can be derived.

- **Constraint Records:** The `constraints` table records user-submitted constraints with the following fields:
  - `id`: Primary key for the constraint.
  - `event_id`: Identifier of the associated event.
  - `username`: The user who submitted the constraint.
  - `assigned_erlang_node`: IP address of the Erlang node that processed the constraint, which is critical for the recovery algorithm.
  - `constraints_list`: A list of strings representing the submitted constraint intervals.

Access Mechanism:
The Spring Boot application interacts with the MySQL Database using the `JdbcTemplate` class, ensuring efficient and secure data operations.

4.4. Event Server Nodes

Three distinct Event Server Nodes run concurrently. They all share the same distributed cookie (`whenly`) to enable secure inter-node communication.

Internal Structure and Key Modules

Each Event Server Node is built using OTP and rebar3, structured under a supervision tree (with a `one_for_one` strategy) that spawns the following processes:

- **Storage Module:** The Storage Module uses *Mnesia DB* for local, fault-tolerant storage. It stores records such as `event_record`, which contains `eventid` and the current `partialsolution` (i.e., the cumulative intersection of constraints), and `event_deadline`, which contains `eventid` and the corresponding `deadline` (used solely by the coordinator). Implemented as a `gen_server`, it provides APIs for getting, storing, and deleting event-related data. Mnesia is configured with `local_content` and `disc_copies` to ensure data persistence in both RAM and on disk.
- **Base Module:** The Base Module acts as the entry point for all requests received from the Erlang Backend Application. For event creation requests, it forwards them to the Coordinator Module. For constraint addition requests, it retrieves the current partial solution from the Storage Module, calculates the new intersection using the Calculator module, and updates the stored value. It handles special cases such as storing the new constraint as the initial solution if no partial solution exists, and ignoring further constraints if a solution is marked as `no_solution` or if the deadline has expired. This module is realized as a `gen_server`.
- **Coordinator Module:** The Coordinator Module manages event creation deadlines and coordinates the final schedule computation. Upon receiving an event creation request, it schedules a deadline expiration. When the deadline is reached, it gathers partial solutions from all Event Server Nodes via RPC calls and computes the final schedule using `final_intersection/1` from the Calculator module. It then sends the computed final schedule to the Erlang Backend Application and marks the event as expired by deleting its deadline from storage. This module is implemented as a `gen_server`.
- **Utility Modules:**

- **Calculator Module:** Provides functions such as `intersect/2` (to compute the intersection between two lists of time intervals) and `final_intersection/1` (to consolidate multiple partial solutions into a final schedule).
- **Config_Reader Module:** Reads configuration parameters from a JSON file (using jsx) and exposes utility functions to retrieve configuration values.

4.5. Additional Distributed Considerations

- **Shared Distributed Cookie:**
  All Erlang nodes (Event Server Nodes and Erlang Backend Application) share the same cookie (`whenly`) to ensure secure and consistent distributed communication.
- **Fault Tolerance & Recovery:**
  OTP supervision trees (with a `one_for_one` strategy) ensure that individual module failures are automatically recovered. Persistent storage in both Mnesia and MySQL ensures that critical event data is never lost.
- **Inter-Node Communication:**
  Erlang's native messaging and RPC mechanisms enable efficient asynchronous communication among nodes, while the Erlang Backend Application bridges communication with the Java Backend via JInterface.

5. System Workflow

This section describes the end-to-end process by which Whenly handles user requests and schedules events. It explains how user interactions are translated into backend processes, how event-related data is distributed and processed across the system, and ultimately how final event schedules are computed and made available to users.

1. **User Interaction**
   Users begin their journey by accessing the WebApp, which serves as the graphical interface of Whenly. Through this interface, users can log in, create new events, or submit scheduling constraints. The WebApp collects user inputs via dedicated JSP pages and associated servlets. These servlets package the information into HTTP requests (using POST for submissions and GET for data retrieval) and forward them to the Backend Node, ensuring that every user action is efficiently relayed into the system.

2. **Backend Processing**
   Upon receiving user requests, the Backend Node processes the data by authenticating users, managing event creation, and handling constraint submissions. It assigns events and their corresponding constraints to the distributed Event Server Nodes using a circular distribution strategy, which ensures uniform load balancing across all nodes. Additionally, the Backend Node persists event information in the MySQL Database to support fault recovery, ensuring that no data is lost even if an event server fails.

3. **Constraint Processing**
   Once the constraints are distributed to the Event Server Nodes, each node processes its assigned constraints. The nodes update partial solutions incrementally by computing intersections of the new constraints with any existing partial solution for the event.

4. **Final Scheduling**
   When an event's deadline is reached, the Event Server Node responsible for that event (the node that originally received the event creation request and hence its deadline) initiates the final scheduling process. This node coordinates with other Event Server Nodes to collect their respective partial solutions. By applying the final intersection algorithm, it computes the final schedule for the event. In cases where no valid solution can be found, the system records a scheduling failure.

5. **Result Storage**
   After the final schedule has been determined, the resulting event schedule is sent back to the Backend Node. The Backend then stores the final schedule in the MySQL Database, ensuring that all participants can access the outcome when they revisit their user area. Whether a user created the event or simply added a constraint, the final schedule becomes visible in their personalized dashboard, providing a clear and timely resolution to the scheduling process.

6. Synchronization, Coordination, and Communication Issues

In a distributed system like *Whenly*, ensuring proper synchronization, coordination, and efficient communication is critical for maintaining consistency, fault tolerance, and overall system responsiveness. This section outlines the primary challenges in these areas and presents tailored solutions.

6.1. Synchronization Challenges and Solutions

Synchronization issues arise when multiple nodes concurrently process event constraints or perform schedule computations, potentially causing inconsistencies or race conditions.

6.1.1. Challenges

- **Deadline Expiry Overlap:**
  As events are dynamically updated with constraints, the moment an event deadline is reached, all ongoing updates must be halted so that the final schedule can be computed. Any overlap between deadline-triggered operations and incoming constraint updates can lead to inconsistencies.
- **Concurrent Access to Shared Erlang Node List:**
  The shared list of strings containing the addresses of the Erlang nodes is used concurrently by the `eventService` and `constraintService` modules to select the target node for sending events and constraints. Although both modules access the list in read-only mode, the `eventService` module may also add or remove elements from the list. This simultaneous access can result in race conditions and data inconsistencies in the Java Backend.

6.1.2. Proposed Solutions

- **Atomic Updates via Mnesia DB:**
  Leverage Mnesia's transactional capabilities to ensure that operations on partial solutions are atomic. This guarantees that updates are either fully applied or entirely rolled back in the event of conflicts.
- **Deadline Expiration Mechanism:**
  When an event reaches its deadline, deadline-triggered computations are initiated through RPC calls within each Event Server Node's storage process. Mnesia ensures that the read operations required for final schedule computation are atomic. Once the final schedule is computed, a subsequent RPC call from the coordinator writes an `expired` marker across all nodes' Mnesia databases via their storage processes. This prevents any further constraint updates from affecting an event that has already been finalized. Late constraints, arriving after the deadline, are simply ignored to guarantee timely schedule finalization.
- **Shared List Access Control via `sharedStringList` Module:**
  To manage concurrent access to the shared list of Erlang node addresses, the Java Backend employs the `sharedStringList` module. This module implements mutex-protected access using a `ReentrantLock` to ensure that only one thread can modify the list at any given time. While both the `eventService` and `constraintService` modules can read from the list concurrently, any modifications—such as additions or removals by the `eventService`—are

serialized. This mutex mechanism is crucial for preventing race conditions and maintaining data consistency across the application.

6.2. Coordination Challenges and Solutions

Coordination challenges emerge when multiple Event Server Nodes must work together to compute the final schedule, particularly under failure conditions.

6.2.1. Challenges

- **Deadline Execution Ownership:**
  The node responsible for an event's deadline must orchestrate distributed scheduling operations and coordinate with other nodes to gather their partial solutions.
- **Failure Handling During Coordination:**
  If the coordinating node (deadline owner) fails during the scheduling process, the system must promptly reassign its responsibilities without data loss or operational disruption.

6.2.2. Proposed Solutions

- **Predefined Deadline Ownership:**
  The Event Server Node that initially receives an event creation request is designated as the deadline owner. This predefined role eliminates the need for runtime leader election.
- **Failure Recovery via Monitoring:**
  The Erlang Backend continuously monitors all Event Server Nodes using mechanisms such as `net_kernel:monitor_nodes(true)`. In addition, during initialization, each node's Base Module attempts to connect with the Erlang Backend to signal its presence. If the deadline owner fails, the recovery algorithm (detailed in the next section) is triggered automatically.
- **Distributed Scheduling Computation:**
  The final schedule is computed in a distributed manner by aggregating partial solutions from all Event Server Nodes that processed constraints for the event. This approach avoids single points of failure and ensures robustness in final schedule computation.

6.3. Communication Challenges and Solutions

Efficient and reliable communication between the WebApp, Backend, and Event Server Nodes is essential for system performance. Challenges include network latency, message duplication or loss, and increased messaging overhead as the system scales.

6.3.1. Challenges

- **Network Latency:**
  Delays in communication between nodes can affect system responsiveness and user experience.
- **Message Duplication or Loss:**
  Network issues might result in duplicated or lost messages, impacting the reliability of data exchanges.

- **Scalability of Messaging:**
  As the number of Event Server Nodes increases, managing and coordinating communication becomes more complex.

6.3.2. Proposed Solutions

- **Reliable Messaging Protocol:**
  Utilize Erlang's built-in messaging system, which inherently provides reliable message delivery and effectively handles duplication issues.
- **Asynchronous Communication:**
  Implement asynchronous messaging to minimize blocking operations. All communications between the Java Backend and the Erlang Backend are handled asynchronously using JInterface, thereby reducing system overhead and improving responsiveness.
- **Communication Protocols:**
  - The WebApp and the Backend communicate via HTTP using RESTful APIs.
  - The Java Backend and the Erlang Backend interact via JInterface, which leverages Erlang's asynchronous messaging capabilities to handle inter-node communications robustly.
  - Communication between the Erlang Backend and the Event Server Nodes is performed through RPC calls, ensuring efficient and reliable data transfer across the distributed system.

7. Recovery Mechanisms

Whenly incorporates robust recovery mechanisms to ensure system continuity in the face of failures. There are two primary recovery approaches: one addressing the failure of an entire Erlang node, and another handling process crashes within individual Event Server Nodes.

7.1. Erlang Node Failure Recovery

The recovery algorithm is designed to handle the failure of an Erlang node by reassigning its tasks—specifically, the processing of constraints and the responsibility for final solution computation—to other active nodes. This mechanism operates as follows:

1. **Node Failure Detection:**
   The `ErlangBackendAPI` module detects the failure of an Erlang node and relays a failure notification to the `EventService` module via the `ErlangMessageHandler`. Simultaneously, the failed node is removed from the list of available nodes maintained by the `sharedStringList` module.

2. **Constraint Recovery:**
   The `EventService` retrieves all constraints assigned to the failed node (using methods such as `constraintRepository.findConstraintsByErlangNode(failedNode)`) that are related to events without a final result. For each such constraint, a new active Erlang node is selected using the `selectErlangNode()` method. The constraint is then reassigned to this node by sending it via the `constraintService.sendConstraintsToErlang` method, and the corresponding `assigned_erlang_node` field in the `constraints` table is updated accordingly.

3. **Event Recovery:**
   The `EventService` also retrieves events managed by the failed node (using methods like `eventRepository.findByErlangNodeIpAndFinalResultIsNull(failedNode)`) that have not yet been finalized. For each event, a new active Erlang node is selected to assume responsibility, and the event's `erlang_node_ip` field in the `event` table is updated. The `erlangBackendAPI.createEvent` method is then invoked to re-establish the event on the new node.

4. **Error Handling:**
   If no active Erlang nodes are available during constraint or event recovery, the system throws an `IllegalStateException`. All exceptions during the reassignment or messaging processes are caught and logged to ensure that issues are promptly addressed.

5. **Scalability and Transparency:**
   This recovery algorithm ensures that the failure of an Erlang node does not result in data loss or disruption of service. The process is transparent to the user and to the remaining Erlang Event Server Nodes, as the Java Backend manages the recovery seamlessly.

7.2. Intra-Node Process Recovery

Within each Event Server Node, fault tolerance is achieved through OTP supervision. A supervision tree, configured with a `one_for_one` strategy, oversees critical processes and automatically restarts

any process that fails. This simple restart is sufficient for all modules except the *Coordinator Module*, which requires additional considerations as outlined below.

Coordinator Process Recovery

The Coordinator is responsible for scheduling deadline expiration events and managing final solution computation for the events it oversees. Since it maintains state regarding pending deadlines, its recovery is critical. In the Coordinator's `init` function, a recovery mechanism retrieves all stored deadlines from the Mnesia database via the Storage Module. For each retrieved deadline, the Coordinator reschedules the corresponding expiration event using a `send_after` function.

- If a deadline is found at initialization, it indicates that the node has restarted. If the Backend has already reassigned the event to another event server node and the solution has been computed, the Coordinator will detect this by retrieving `expired` markers from other event server nodes, signaling that the event has been finalized. In this case, the Coordinator will not send anything to the Backend, as the final schedule is already computed.
- If only the Coordinator process restarted (not the entire node), and the Backend recovery mechanism did not take place, the Coordinator will compute the final schedule when the deadline expires, send it to the Backend, broadcast the final result to all Event Server Nodes (marking the event as finalized), and then remove the event deadline from storage.

This intra-node recovery ensures that even if the Coordinator process crashes, the state of ongoing events is recovered automatically, maintaining system consistency and minimizing disruption.

Together, these recovery mechanisms enable Whenly to maintain high availability and fault tolerance, ensuring that both node-level and process-level failures are handled transparently and efficiently.

8. Conclusion

The *Whenly* distributed event scheduling application addresses the complex challenges of efficient, fault-tolerant, and scalable event coordination in modern systems. By leveraging a modular architecture, the system ensures seamless collaboration between its WebApp, Backend, and distributed Event Server nodes. Key technologies like *Java*, *Erlang*, *Mnesia DB*, and *MySQL* are strategically utilized to guarantee robust synchronization, coordination, and communication. By employing deadline prioritization, distributed constraint management, and reliable inter-node messaging, *Whenly* offers a user-friendly platform that balances performance with fault tolerance.