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.

Application usage:

- *Event Organizers* create events with deadlines and invite participants to provide their availability and constraints.
- *Participants* submit their constraints, such as time preferences, through a user-friendly WebApp.
- 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 invitation or a shared link.
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 10,000 concurrent users with minimal latency (<1 second for responses to WebApp requests).
   - Final schedules must be computed within a reasonable time frame (e.g., <10 seconds for typical events with up to 50 constraints).
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 section.

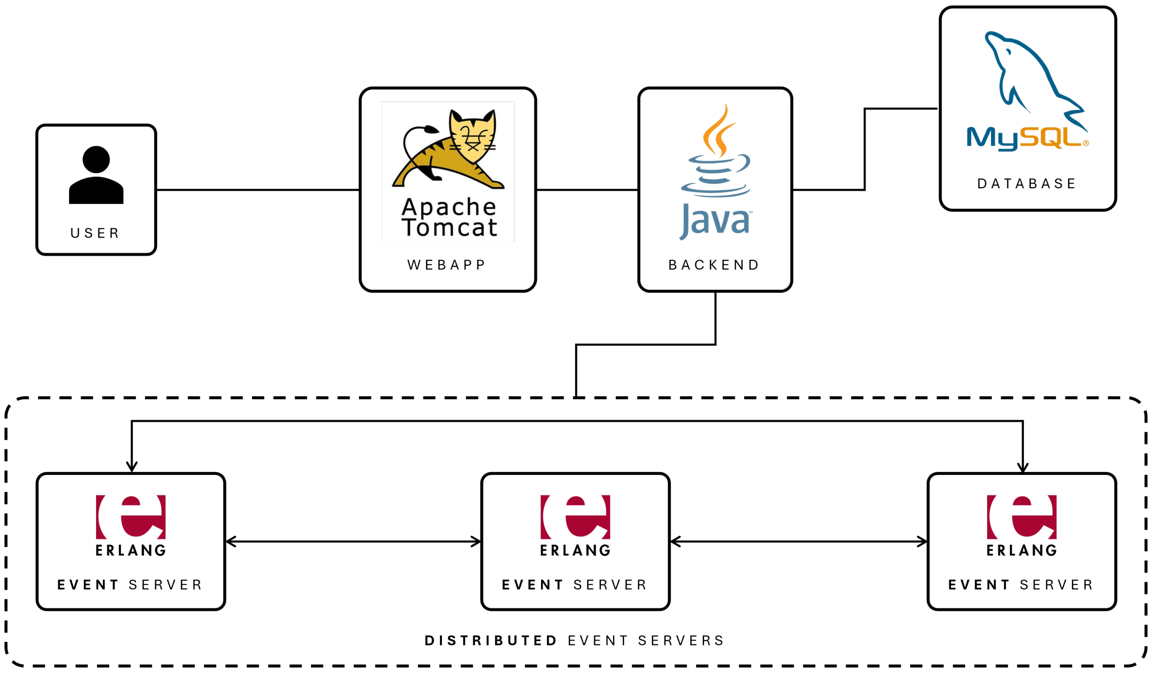

Figure 1: A high-level architecture diagram of Whenly, showcasing the modular separation of WebApp, Backend, Database, and distributed Event Server Nodes for scalable and fault-tolerant event scheduling.

4.1. Component Breakdown

1. WebApp Node
   - Hosted on *Apache Tomcat*, this node serves as the primary interface for user interaction.
   - Responsibilities:
     - Provides a graphical user interface (GUI) for event management and constraint submissions.
     - Forwards user requests to the *Backend Node* for processing.
2. Backend Node
   - Implements the core application logic using *Java*.
   - Responsibilities:

    - Handles user authentication and manages event creation.
    - Ensures persistence by storing metadata and constraints in the *Database Node*.
    - Distributes event-related tasks and constraints to *Event Server Nodes* using a circular load-balancing strategy.

3. Database Node
    - Centralized storage implemented with *MySQL*.
    - Responsibilities:
        - Stores user credentials, event metadata, and constraints.
        - Associates constraints with the *Event Server Node* responsible for processing them.
        - Stores final schedules or failure notifications for completed events.
4. Event Server Nodes
    - Distributed nodes implemented using *Erlang* with *Mnesia DB* for fault-tolerant local storage.
    - Responsibilities:
        - Maintain partial solutions for events and update them as constraints are received.
        - Coordinate with other *Event Server Nodes* to compute final schedules when event deadlines expire.
        - Store partial solutions locally in the *Mnesia DB* for consistency and fault tolerance.

4.2. System Workflow

1. User Interaction:
    - Users interact with the WebApp, which forwards their requests to the Backend Node for processing.
2. Task Distribution:
    - The Backend Node assigns constraints and events to Event Server Nodes in a circular manner, ensuring uniform load distribution.
3. Constraint Processing:
    - Event Server Nodes process constraints and compute incremental updates to partial solutions.
4. Final Scheduling:
    - The Event Server Node responsible for the event deadline initiates a distributed computation algorithm to finalize the schedule in collaboration with other Event Server Nodes.
5. Result Storage:
    - Final schedules are stored in the Database Node, and temporary data is cleaned from Event Server Nodes.

5. Synchronization, Coordination, and Communication Issues

In a distributed system like *Whenly*, ensuring synchronization, coordination, and efficient communication is critical for maintaining consistency, fault tolerance, and system responsiveness. The following sections analyze the primary challenges in these areas and propose solutions tailored to the application's architecture.

5.1. Synchronization Challenges and Solutions

Synchronization issues arise when multiple nodes simultaneously process event constraints or schedule computations, potentially leading to inconsistencies or conflicts.

5.1.1. Challenges

- **Deadline Expiry Overlap:** Events are dynamically updated with constraints, but when the deadline arrives, all updates must halt to compute the final schedule. Overlapping updates could cause inconsistencies.

5.1.2. Proposed Solutions

- **Atomic Updates via Mnesia DB:**
  - Leverage **Mnesia's transactional capabilities** to ensure atomic operations when updating partial solutions. This guarantees that updates are either fully applied or fully rolled back in case of conflict.
- **Deadline Prioritization:**
  - Assign higher priority to deadline-triggered computations. When an event reaches its deadline, all constraint updates must be paused to prevent conflicts.

  - Constraints arriving after the deadline are ignored because the event creator must be assured of receiving a final schedule on time. The system cannot afford to delay schedule computation for late constraints that may still be in the processing queue.

5.2. Coordination Challenges and Solutions

Coordination issues arise when multiple **Event Server Nodes** must collaborate to compute the final schedule for an event, particularly when failures occur.

5.2.1. Challenges

- **Deadline Execution Ownership:**
  - The node responsible for an event's deadline must coordinate with others to initiate and manage the distributed scheduling computation.
- **Failure Handling During Coordination:**
  - If the coordinating node fails during the scheduling process, the system must ensure that another node can take over without data loss or disruption.

5.2.2. Proposed Solutions

- **Predefined Deadline Ownership:**
  - The **Event Server Node** that initially received the event creation request is designated as the deadline owner. This eliminates the need for an election process during runtime.
- **Failure Recovery via Heartbeat Monitoring:**
  - The **Backend Node** continuously monitors all **Event Server Nodes** using a heartbeat mechanism. If the deadline owner node fails:
    1. The next node in the circular distribution automatically assumes leadership.
    2. The **Database Node** ensures recovery by storing backup metadata of all assigned constraints.
- **Distributed Scheduling Computation:**
  - Instead of relying on a single node for computation, the final schedule is computed in a **distributed manner** by aggregating partial solutions stored in all **Event Server Nodes** that processed constraints for the event.

5.3. Communication Challenges and Solutions

Efficient and reliable communication between **WebApp, Backend, and Event Server Nodes** is essential for performance and consistency. Communication failures or network congestion could delay event scheduling operations.

5.3.1. Challenges

- **Network Latency:** Communication between nodes might experience delays, affecting system responsiveness.
- **Message Duplication or Loss:** Messages sent between nodes may be duplicated or lost due to network issues.
- **Scalability of Messaging:** As the number of **Event Server Nodes** increases, managing communication efficiently becomes more complex.

5.3.2. Proposed Solutions

- **Reliable Messaging Protocol:**
  - Use **Erlang's built-in messaging system**, which provides reliable message delivery and handles duplication efficiently.
- **Asynchronous Communication:**
  - Implement asynchronous messaging where possible to minimize blocking operations, reducing system overhead and improving responsiveness.

- **Retry Mechanism with Exponential Backoff:**
  - For critical messages (e.g., deadline notifications and final schedules), implement automatic retries with exponential backoff to minimize network congestion and ensure message delivery.
- **Optimized Broadcast Strategy:**
  - Instead of sending messages to all nodes in a brute-force manner, use **tree-based dissemination** techniques to reduce network overhead when distributing event-related data.

5.4. Summary of Solutions

Category	Problem	Proposed Solution
Synchronization	Deadline-triggered computation conflicts with ongoing constraint updates.	Atomic updates via Mnesia DB; Deadline prioritization to ensure event completion on time.
Coordination	Failure of the event deadline owner node disrupts scheduling.	Predefined ownership; Heartbeat-based failure recovery; Distributed schedule computation.
Communication	Network latency, message loss, and scalability issues.	Reliable messaging via Erlang; Asynchronous communication; Optimized broadcast techniques.

6. 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.