WHENLY

A Distributed Event Scheduling Application

Architecture Overview

1. **WebApp Node**
   - A graphical user interface hosted on **Apache Tomcat**.
   - Primary Responsibilities:
     - Provides the user interface for interaction with the application.
     - Forwards all user requests to the **Backend Node** for processing.
2. **Backend Node**
   - Implements the core application logic using **Java**.
   - Key Responsibilities:
     - Facilitates user authentication processes, including sign-in and sign-up.
     - Manages event creation and constraint submissions.
     - Persists event-related metadata in the **Database Node**.
     - Distributes tasks and constraints to the **Event Server Nodes** in a circular manner to ensure uniform load balancing across all nodes.
3. **Database Node**
   - A centralized storage system implemented using **MySQL**.
   - Key Functions:
     - Stores user credentials.
     - Maintains metadata and constraints for all events, ensuring fault tolerance by associating constraints with the ID of the Event Server Node to which the constraints were sent.
     - Archives the final schedules of completed events.
4. **Event Server Nodes** (Distributed Erlang Nodes)
   - Nodes implemented in **Erlang** using **Mnesia DB** for fault-tolerant, distributed local storage.
   - Responsibilities:
     - **Distributed Constraint Management:** Maintains partial solutions for events based on received constraints.
     - **Incremental Solution Computation:** Updates partial solutions dynamically with each new constraint received.
     - **Distributed Coordination:** The server responsible for an event's deadline (i.e. the one that received the event creation request) initiates and orchestrates a distributed algorithm to compute the final schedule, collaborating with all the other **Event Server Nodes**.

Key Workflow

1. **User Authentication and Event Management**

- Users interact with the **WebApp Node**, which forwards all requests to the **Backend Node**.
- The **Backend Node**:
  - Authenticates users by querying the **Database Node**.
  - Facilitates event creation and the submission of scheduling constraints.
  - Persists event metadata and constraints in the **Database Node**.
  - Assigns events and their associated constraints to the **Event Server Nodes** in a circular distribution pattern.

2. **Constraint Submission**

   - Users submit constraints via the **WebApp Node**.
   - The **Backend Node** forwards the constraints to the appropriate **Event Server Node** based on the circular distribution policy.
   - The receiving **Event Server Node**:
     - Computes or updates the partial solution for the event.
     - Stores the updated partial solution in its **Mnesia DB** for fault tolerance and consistency.

3. **Partial Solution Management**

   - Each **Event Server Node** is responsible for:
     - Storing and maintaining partial solutions for all events for which it received either the creation request with a deadline or a new constraint.
     - Incrementally updating partial solutions as new constraints are received.
     - Ensuring fault-tolerant storage of these solutions using **Mnesia DB**.

4. **Final Solution Computation**

   - When an event's deadline expires, the responsible **Event Server Node** (the one that received the event creation request):
     - Initiates the distributed scheduling algorithm.
     - Coordinates with other **Event Server Nodes** to gather their respective partial solutions for the event.
     - Computes the **final schedule** by intersecting all collected constraints.
     - Records a scheduling failure in cases where no valid solution can be derived.

5. **Result Storage and Cleanup**

   - Once the final schedule is computed:
     - The **Backend Node** stores the completed schedule in the **Database Node**.
     - All temporary data, such as partial solutions related to the event, are removed from the **Mnesia DB** on the involved **Event Server Nodes**.

Design Considerations

1. **Event Server Responsibilities**

- **Partial Solutions Storage:** Each server uses **Mnesia DB** to store partial solutions reliably.
- **Distributed Coordination:** Servers collaborate to ensure efficient final schedule computation.
- **Deadline Ownership:** The server holding the event's deadline initiates the distributed scheduling algorithm upon deadline expiration.

2. **Fault Tolerance and Recovery**
   - The **Database Node** acts as a centralized backup for all critical event and constraint data, ensuring seamless recovery in the event of server failure.
   - In the case of a failed **Event Server Node**, the **Backend Node** reassigns its tasks to an operational server, ensuring continued processing without data loss.
3. **Synchronization and Concurrency**
   - **Erlang's lightweight messaging protocols** enable efficient distributed communication among **Event Server Nodes**.
   - **Mnesia DB** ensures atomic updates and consistency for all stored partial solutions.
4. **Scalability**
   - The system's modular design allows for independent scaling of the WebApp, Backend, and Event Server Nodes based on demand.
   - New **Event Server Nodes** can be added seamlessly to handle increased workloads.

Tools and Technologies

1. **Frontend (WebApp)**
   - Developed using **Java JSP** and hosted on **Apache Tomcat** to provide a responsive user interface.
2. **Backend**
   - Implemented in **Java** for robust application logic and seamless integration with the database and event servers.
3. **Event Servers**
   - Built with **Erlang** for high concurrency and fault-tolerant distributed processing.
   - **Mnesia DB** ensures reliable local storage of constraints and partial solutions.
4. **Database**
   - A **MySQL** database provides centralized, persistent storage for user credentials, event metadata, and final schedules.

Advantages

1. **Distributed Processing:**
   - The **Event Server Nodes** handle computation locally, reducing bottlenecks and ensuring scalability.
2. **Fault Tolerance:**
   - Critical data is safeguarded in the **MySQL Database**, allowing for seamless recovery in the event of a node failure.
3. **Efficiency:**
   - Incremental updates to partial solutions minimize computational overhead and improve system responsiveness.
4. **Modular Architecture:**
   - The separation of components into WebApp, Backend, and Event Servers simplifies maintenance and enables independent scaling.