QWASAR

Silicon Valley

Qwasar Master's of Science in Computer Science Programs

Engineering Lab

Marry Me

Problem Statement

Congratulations, your best friend is getting married, and he had the brilliant idea to appoint you as the person responsible for setting everything up for the ceremony! Not only is he looking to save some money on the whole event, but he presented to you what could be a very lucrative business opportunity in the Marriage celebration industry.

The main drive for what you discussed with your friend is keeping stress levels, for both guests and wedding couple, at low, by optimizing how different situations are handled on the big day.

Objective

To design, develop, and test an application capable of receiving multiple events, and delivering them to the appropriate teams, according to priority and team members availability.

Solution

The entrypoint of your application is the **C**oordinator role. They are responsible for receiving events, checking if they are valid, and sending them to the next step of the application: **The Marry Me Organizer.** This second part is responsible for distributing events according to their **Type** and **Priority**.

After passing by the Organizer, Events are received by **T**eams (Waiters, Catering, Security, Officiant, Cleaning, etc.), which then checks availability of **W**orkers, under that team, to deal with each event.

If an event is due (wasn't dealt before the timeframe is up), it's discarded, and a Stress level is to be incremented. Every event (solved or unsolved, along with Stress level and any other useful information should be properly logged).

Entities

Coordinators:

- Receive events
- Validate events

Workers:

- Handle(😉) events.
- Attributes:
    - Team: [Security | Clean Up | Catering | Officiant | Waiters]
    - Current_status: [Idle | Working]

Teams

- Receive valid events from the **Organizer.**
- Check Workers availability and distribute events, organized by priority.
- Attributes:
    - Routine: [Standard | Intermittent | Concentrated]

Guests

- Generate events.
- Get **Stressed** or **Happy**, depending on how fast events are dealt with.
- Attributes:
    - Current_status: [Happy | Stressed]

Events

- Attributes:
    - Event_type:
    - Rings
    - Priority:
    - Description:

```
None
// Initial Event type by Team

Security {
       brawl,
       not_on_list,
       accident,
       }

Clean_up {
       dirty_table,
       broken_itens,
       dirty_floor
       }

Catering {
       bad_food,
       music,
       feeling_ill
       }

Officiant {
       bride,
       groom
       }

Waiters {
       broken_itens,
       accident,
       bad_food
       }
```

```
None
// Priorities timeframes

High   ->  5 minutes (5 second in the simulation)
Medium -> 10 minutes (10 second in the simulation)
Low    -> 15 minutes (15 second in the simulation)

//Routines timeframes

Standard     -> 20 seconds working, 5 seconds idle;
Intermittent -> 5 seconds working, 5 seconds idle;
Concentrated -> 60 seconds working, 60 seconds idle;

// Every Event taks 3 seconds to be handled, regardless of type, priority
```

```
None
//Team routines
    'Security': Standard,
    'Clean_up': Intermittent,
    'Catering': Concentrated
    'Officiant': Concentrated
    'Waiters': Standard
```

Workers Routines

Regardless of the Worker team, at the beginning of the simulation, the worker will behave according to its routine type. For example: Imagine in a given dataset, the Waiter team has a Routine of type **Standard** (20 seconds working, 5 second idle.). At the beginning of the Dataset simulation, every worker of this team will be unavailable to handle events for the first **20 seconds**. After that, it will have a **5 seconds** idle window, where it can handle events. When those 5 seconds are up, the worker goes back to be unavailable for more 20 seconds. This cycle repeats itself, for each team. The unavailable/idle times vary according to the team.

Wedding simulations

Each simulation will consist of a **6 minute** wedding (each minute will be equivalent to an hour, each second, equivalent to a minute). In this time frame, each

simulation will generate different payloads of events that will be delivered to the entrypoint of the application (the Coordinators).

At the beginning, all guests are with the **Happy** status. Each time an event is not dealt within its priority time frame, a guest **Stress Mark** is added to the simulation result.

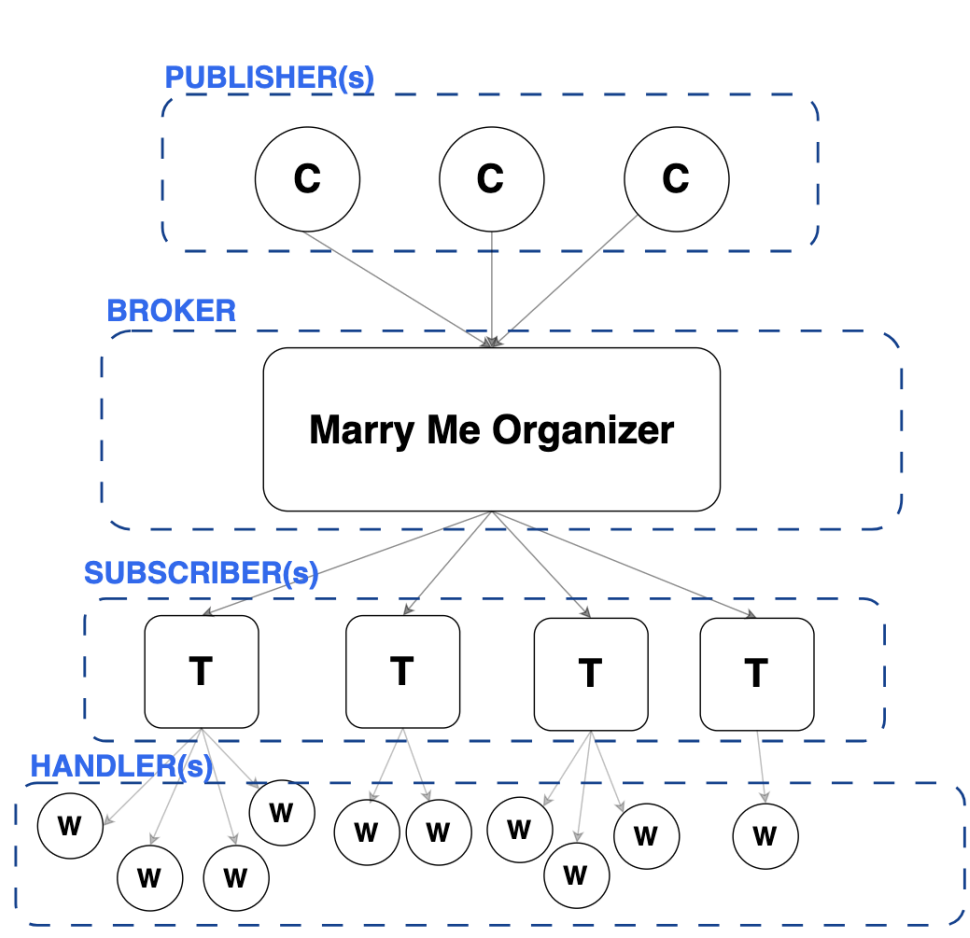
Each event has a type (which allows the coordinator to validate and address it to a team that is able to act on it), and a priority (which translates to the timeframe the event has to be dealt with, before a guest becomes stressed).

The goal is to deal with all the events without increasing the overall stress level too much.

```
None
// Event format example

event {
       id: 1
       event_type: feeling_ill,
       priority:  medium,
       description: guest has stomach ache after eating 5 pieces of cake,
       timestamp: 05:37 // 5 hours and 37 minutes after the beginning of the
weeding. 00:00 is the start of the wedding, and 06:00 is the end.
}
```

Overall Architecture and Constraints



References

- Kafka
- RabbitMQ
- EventBridge
- Flink
- Pinot (Not the Wine🥲)

Stack and Technologies Used

- Any Event Driven Platform, broker, tool, etc.
- C++, Rust, Python, Java, Go, Ruby, PHP, Scala, Clojure, OCaml, Node.js, etc.
- Any Database

Deliverables

- **README**
- **Repository with proper instructions to build locally or link to live application**
- **Final report with "stress level" for each Wedding Simulation**

Outcomes

- **Implementation of Event-Driven Architecture:** Students should demonstrate a clear understanding of event-driven architecture principles and how they apply to the project. This includes designing systems where components communicate asynchronously through events.

- **Integration with Messaging Systems:** Successful integration with messaging systems is crucial. Students should demonstrate their ability to configure, deploy, and utilize these systems effectively within their project.

- **Reliability and Fault Tolerance:** The system should be resilient to failures at different levels. Students should implement strategies such as message retries, dead letter queues, and circuit breakers to ensure reliability and fault tolerance.

- **Monitoring and Logging:** Implementing robust monitoring and logging solutions is essential for troubleshooting and maintaining the system. Students should demonstrate their ability to collect and analyze metrics, as well as handle logging and error tracking effectively.

- **Documentation and Presentation:** Clear documentation outlining the architecture, design decisions, and implementation details should be provided. Additionally, students should be able to effectively communicate their project through a presentation.