Event Bus - Low Level Design

What are events, and what is an event bus?

Events are objects that get sent and received by system components. Each object encapsulates input parameters for subsequent actions or logging.

A bus is a distributed data structure which accepts and transmits these events.

For example, on a `createProfile` request, the profile service creates an event and publishes it to the event bus. The email service may subscribe to the event bus to receive the `createProfile` event, and send a welcome email to the user.

Why do we need an event bus ?

As our service scales, so does it's communication need. The critical interactions become complex, and the critical path becomes broader. Components are tightly coupled with others. An event bus decouples components with a **publisher-subscriber** model. Publishers are unaware of their subscribers. Subscribers are unaware of their publishers. They speak using events, pushed and polled from a central bus (The event bus). There is **loose coupling and separation** while the services communicate.

Now let's note down the requirements for our event bus

- It must support **multiple publishers and subscribers**.
- Component can subscribe to events of a topic or multiple topics. So we need to **partition events into different queues depending upon the topic**. Additionally we each event can have multiple topics so it will be published to multiple queues.
- We want to ensure that an event is processed only once even if the event is published to queue multiple times. In short we want to achieve **idempotency**.
- It might happen that the event bus sends an event to a subscriber but does not receive any acknowledgement. We need to handle this case.
- We should be able to **retry on failures**.
- Event bus can either push events to components or the components can pull events from the bus. So we need to implement **push/pull mechanism**. We must allow components to pull events after a particular ID or timestamps.
- Some events have to be processed in order. For example event B must be processed only after event A. Our event bus must **process these events in order**.

Let's start implementing our event bus keeping the requirements in mind.

- Event is is a distributed system. It can have multiple queues. So it can support **multiple publishers and subscribers**.

- To achieve **idempotency** we can add an **unique ID** to each event. If the event bus receives an event ID that was already published then it will accept it and send acknowledgement but it will not process the event again.
- If we cannot send events to components multiple retries then it does not make sense to keep sending it forever. So we need a **dead letter queue**. If after multiple retries we cannot send the event we push it to the dead letter queue.
- We can pull events from **dead letter queue** and diagnose those events. After that we can resend those events. This takes care of the requirement **retry on failures**.
- To **process events in a particular order** we have two methods
    - **Using timestamps** When a component publishes an event it can add the timestamp to it. So now we can order the events. However there is an issue with this approach. It might happen that some components are faster/slower than others and it can mess up the ordering.
      For eg. A is faster than B. A published an event with timestamp 5 at 2.5s and B published an event with timestamp 3 at 3s. Ideally we should process A's event first and then go for B but if we go with the timestamp approach we will process B then A.
      We can sync the clocks for all the components repeatedly or we can use vector clocks but it makes our system more complicated.
    - **Using threads** An operating system divides the total time it has into multiple slices and assigns each slide to a thread. This allows for concurrent execution. We have to process two types of events, first where the ordering matters and second where it doesn't.
      So how does thread helps us do that?
      If we want to process events without any ordering then we can assign them to multiple threads. This increases concurrency. But if we want some events to execute in order then we will assign them to the same thread. This way the next event will be processed if and only if the previous one is completed.
      This process is called **causal ordering**.

Understanding the mechanism of our event bus

Here are some terms that we are going to use in the explanation below

Tr = Total Number of Reader threads Tw = Total Number of Writer threads h(x) = Hash function (Returns hash value of x)

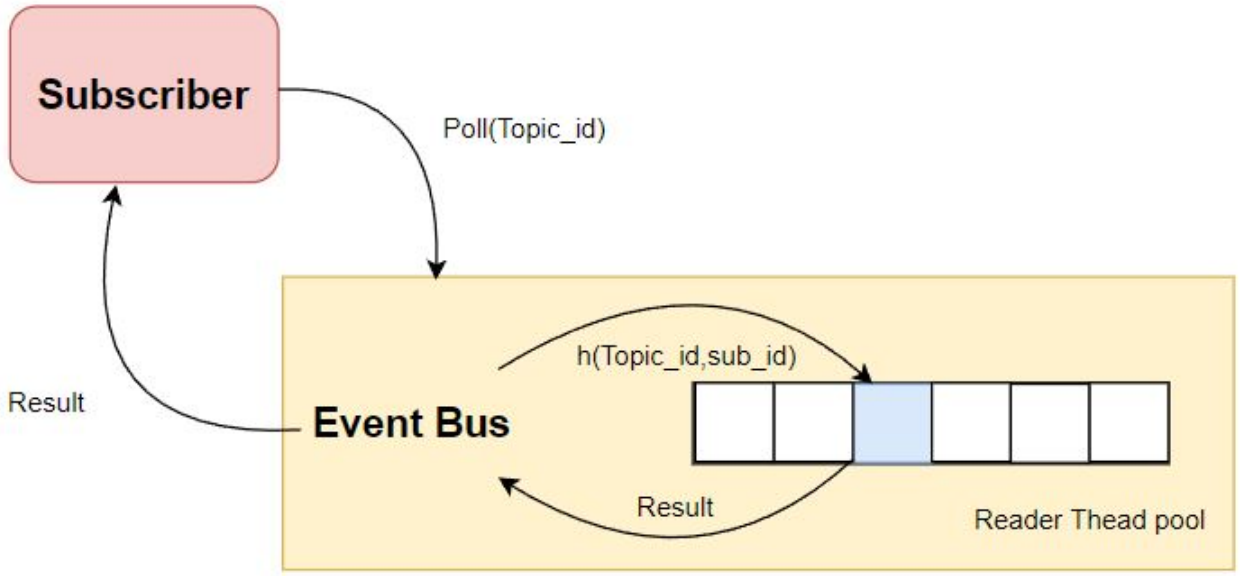
Let's first understand how a **subscriber will pull events from the event bus.**

A component subscribes to topic in the bus. Every time there is a pull request it goes to the event bus. Then the event bus calculates the **h(subscriber_id)%Tr** (*Let the value be t1*). Then it assigns the task of pulling events to thread t1. All request of a particular subscriber_id will fall into the same thread so they will be executed in order.

Now can we optimize this process further?

Yes, we can. Let's understand with an example. Suppose component A subscribes to 2 topics, `Topic1` and `Topic2` . It wants events of `Topic1` in order and events of `Topic2` in order, however it does not care if event of `Topic1` are is before or after the events of `Topic2` i.e., we only care about ordering of same topics not different topics.

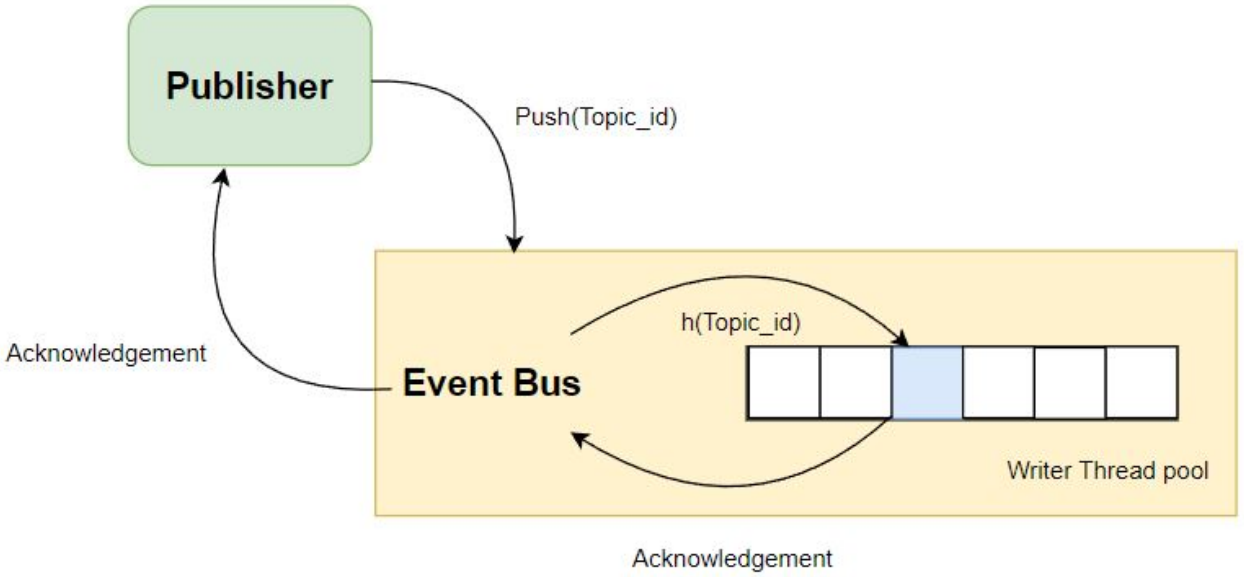
So we calculate hash value of **h(subscriber_id,topic_id)%Tr**. With this change we have multiple topics running on multiple threads possibly for the same subscriber. This **increases concurrency** and thus **reduces the response time.**



Let's discuss how will a **publisher push events to the event bus.**

All events belonging to a particular topic must be ordered. There can be multiple publisher publishing events of same topic, so we can calculate the hash value **h(topic_id)%Tw**. This ensures that events of a particular topic will always fall into same thread.

So we have **ordered reads per (subscriber,topic)** And **ordered writes per (topic)**



So which retry algorithm are we going to use?

We need a mechanism to determine how we are going to push this events and at what intervals we are going to push them. Is the interval going to be fixed or variable. Also our retry algorithm should not overload the service. Keeping all these parameters in mind we need to design a retry algorithm.

For our case we are going to implement both **Periodic retry algorithm Exponential Backoff algorithm**.

In a periodic retry algorithm we send our event after a fixed interval of time until we reach **max attempts.**

But what is **Exponential Backoff algorithm?**

In a simple retry algorithm it might happen that our requests are further adding to the overload of the busy component, then the component will be in the overloaded state longer and will take longer to recover from this state.

However in **exponential backoff instead of retrying after waiting for a fixed amount of time, we increase the waiting time between reties after each retry failure.** This gives the service some breathing time so that if the fault is due to service overload, it could get resolved faster.

We can also define the wait time for each event.

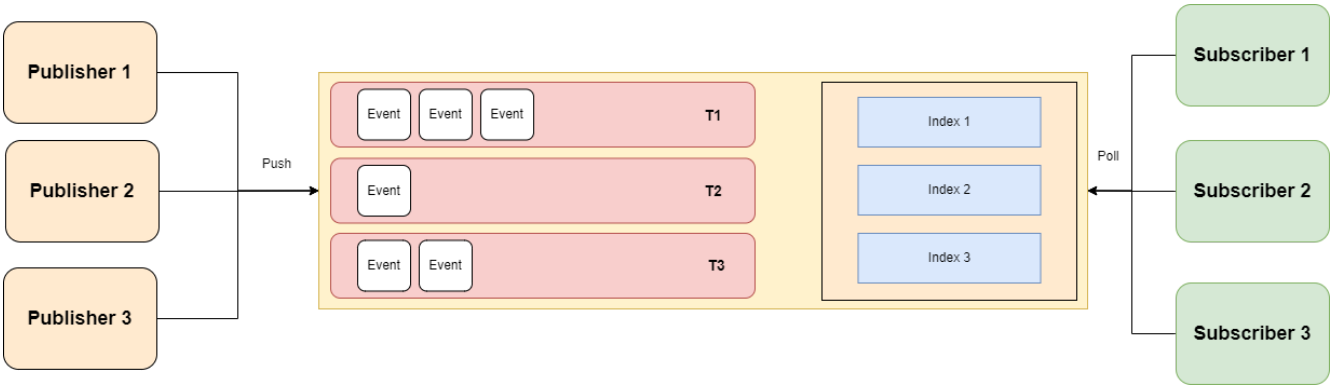
It is also important that we retry when we get an **RetryAble Exception**. If the fault is in the event then no matter how many retries we do it will always fail.

We continue this process until we reach the **maxRetries**.

Time interval can be given by the formula:

Time interval = 2^(Number of attempts-1)

Low Level design Diagram for event bus



That's it for now!

You can check out more designs on our video course at InterviewReady.