Publish Subscribe Architecture

Group Name: Awesome Possum

Description

There are three main components to the Publish Subscribe Model: publishers, eventbus/broker and subscribers. A Publish-Subscribe Architecture is a messaging pattern where the publishers broadcast messages, with no knowledge of the subscribers. Similarly the subscribers 'listen' out for messages regarding topic/categories that they are interested in without any knowledge of who the publishers are. The event bus transfers the messages from the publishers to the subscribers.

Each subscriber only receives a subset of the messages that have been sent by the publisher; they only receive the message topics or categories they have subscribed to. There are two methods of filtering out un-required messages: topic-base filter or content-based filter.

The topic-based filtering requires the messages to be broadcasted into logical channels, the subscribers only receives messages from logic channels they care about (and have subscribed to). A content-based system allows subscribers to receive messages based on the content of the messages and the subscribers themselves must sort out junk messages from the ones they want.

Advantages

Low coupling on the publisher's end

Publishing modules can be built with little concern for subscribers. As the publisher is unaware of the number, identity or the message types of the subscribers it is only responsible for outputting data in response to the correct events through it's API's. Eg, the "onchange" DOM handler for HTML <input> elements simply broadcasts the new value of the input field when a change is detected, it is the responsibility of the developer to interpret that data correctly.

Reduced cognitive load for subscribers

From the perspective of a subscriber, the publisher exists as a black box. Subscribers need not concern themselves with the inner workings of a publisher, often times they do not even have access to the source code. Subscribers only interact with the publisher through the public API exposed by the publisher.

Separation of concerns

Due to the simplistic nature of the architecture (being that data flows one way from publishers to subscribers) developers can exercise fine grained separation of concerns by dividing up message types to serve a single simple purpose each. Eg. data with a topic “/cats” should only contain information about cats, same for “/dogs”. Cat lovers can subscribe to “/cats”, dog lovers can subscribe to “/dogs” and lovers of both can subscribe to both channels, however when a message comes from “/cats” we know for certain it will not contain information on dogs.

Improved testability

Due to the fine grained topic control that is achievable described before, it is easy to test that the various event buses are sending the correct messages.

Improved Security

The Pub/Sub architecture lends itself well to the security principle of assigning minimal privileges / information. It is easy for developers to build modules that are subscribed to the minimal set of message types they need to function.

Disadvantages

Inflexibility of data sent by publisher

The publish/subscribe model introduces **high semantic coupling** in the messages passed by the publishers to the subscribers. Once the structure of the data is established, it becomes difficult to change. In order to change the structure of the messages, all of the subscribers must be altered to accept the changed format. This can be difficult or impossible if the subscribers are external. This disadvantage is common with any versioned API.

A common solution for this problem is using a versioned messaging format so that subscribers can verify the format they are receiving. However, this still assumes that subscribers are consuming the versioning information correctly.

Another solution is to use versioned endpoints, such as /APIV0/ and /APIV1/ for backwards compatibility. A drawback to this solution is then having to support multiple versions, which is taxing on developers.

Instability of Delivery

Another drawback of the publish/subscribe pattern is that it is difficult to gauge the health of subscribers. The publisher does not have perfect knowledge of the status of the systems listening to the messages.

For instance, publish/subscribe is commonly used for logging systems. If a logger subscribing to the ‘Critical’ message type crashes or gets stuck in an error state, then the ‘Critical’ messages may be lost! Then any services depending on the error messages will be unaware of the problems with the publisher.

This problem is shared by any type of client/server system. One advantage that pub/sub offers however, is that multiple identical instances of a logger can be run concurrently with little added overhead to the system. That allows for a high level of redundancy in system designs.

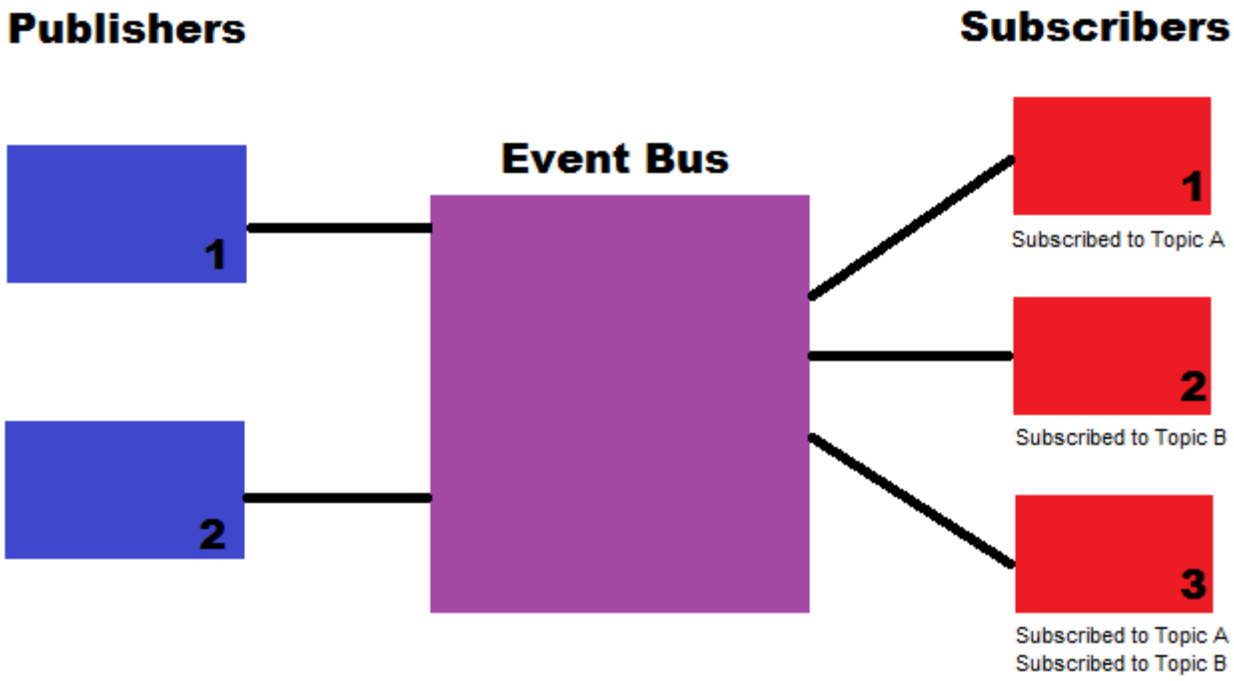
Additions to the architecture can be made to decrease this liability, such as requiring receipts of received messages. With that feature added, feedback can be given to the publisher as to the status of the subscribers.

Bottlenecks

As a pub/sub system scales, the broker often becomes a bottleneck for message flow. Load surges can slow down message sending, and subscribers get a spike in response time.

In systems that are publically accessible, brokers can be susceptible to attacks that register a large number of subscribers to overload the system.

Example



In this topic-based example the Event Bus knows what topic each subscriber is subscribed to. The event bus will filter messages based on topic and send the messages to subscribers that are subscribed to the topic of the message. The publishers are responsible for defining the topics of their messages.

In the above diagram, any message published with Topic A will be sent to Subscriber 1 and Subscriber 3. Similarly, any message published with Topic B will be sent to Subscriber 2 and Subscriber 3. If a publisher were to send a message about Topic A, but wrongfully define it as Topic B, it would be sent to the subscribers of Topic B only.