Alternative Broker Implementation based on Apache Kafka

1. Project Overview

Knative eventing "*is a system that is designed to address a common need for cloud-native development and provides composable primitives to enable late-binding event sources and event consumers*".
Knative Eventing system provides loosely coupled services called sources (or producers) and consumers that can be deployed on any platform independently.
A goal of the Knative Eventing system is to provide interoperability between systems and providers by leveraging cloud events specification and a common interface for producing, delivering, and consuming events.
The delivering aspect is represented by the **Broker** abstraction that is "*a bucket of events which can be selected by attribute. It receives events and forwards them to subscribers defined by one or more matching Triggers*".

Recent discussions of the Knative Eventing community introduced an alternate Broker proposal that addresses the needs of having more "*control over the fundamental implementations of Brokers*" by having Broker Implementations manage Triggers directly and there are no separate Trigger / Broker reconcilers.

The project idea aims to implement a Kafka controller and Kafka dispatcher, based on the alternate Broker proposal that uses Apache Kafka for delivering events.

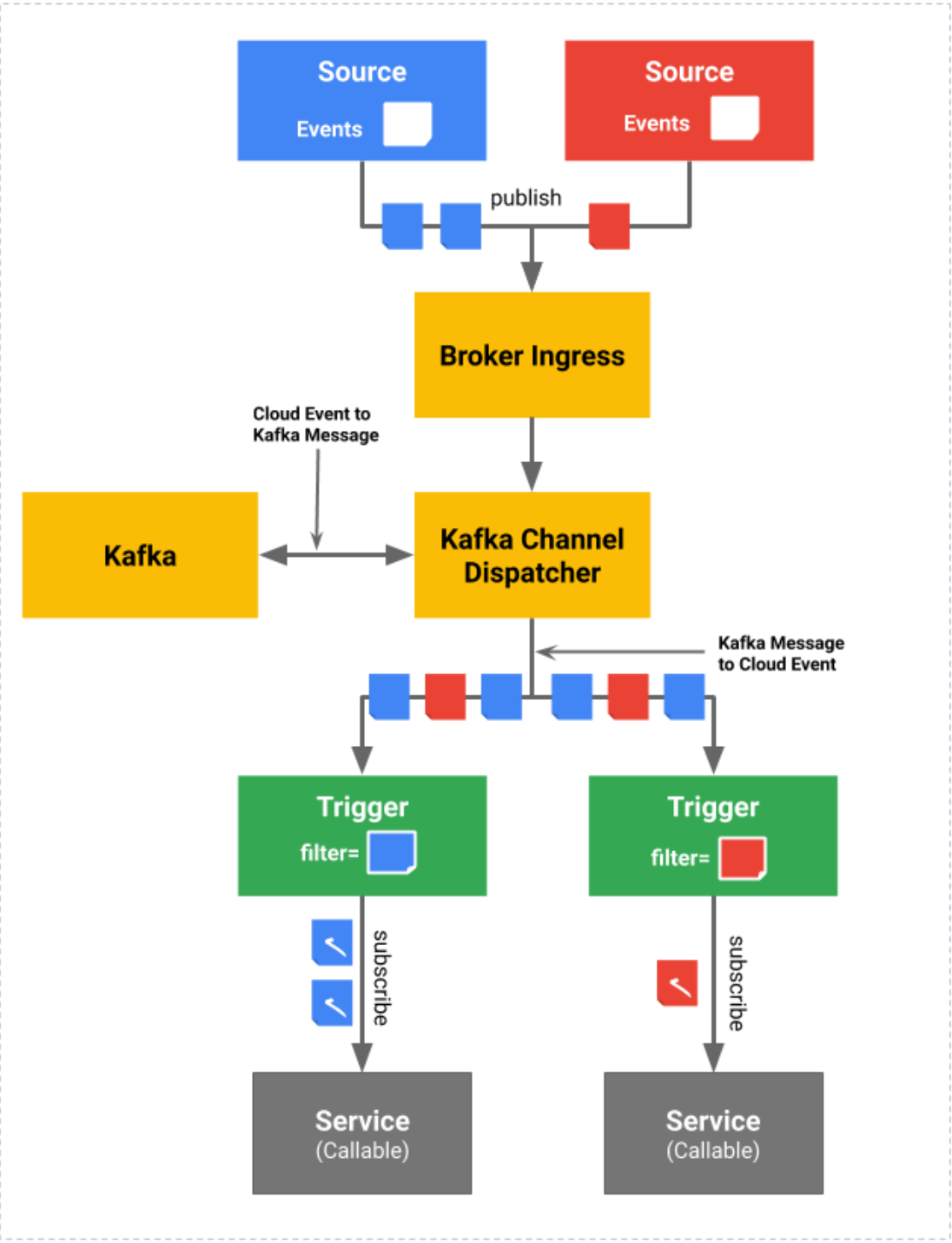
2. Deliverables

1. Implement the control-plane and the data-plane
2. Conformance tests for the Broker to match the Broker Conformance Spec
3. Filter events based on trigger filters.
4. Write documentation
5. Performance tests to compare the channel based broker implementation (eventing benchmark results) (If possible)

3. Implementation Plan

Before diving into the implementation plan, it's worth mentioning the current implementation of Kafka Channel Based Broker and how it relates to Triggers.

The following diagram shows how the Kafka channel based broker handles cloud events:



Data plane

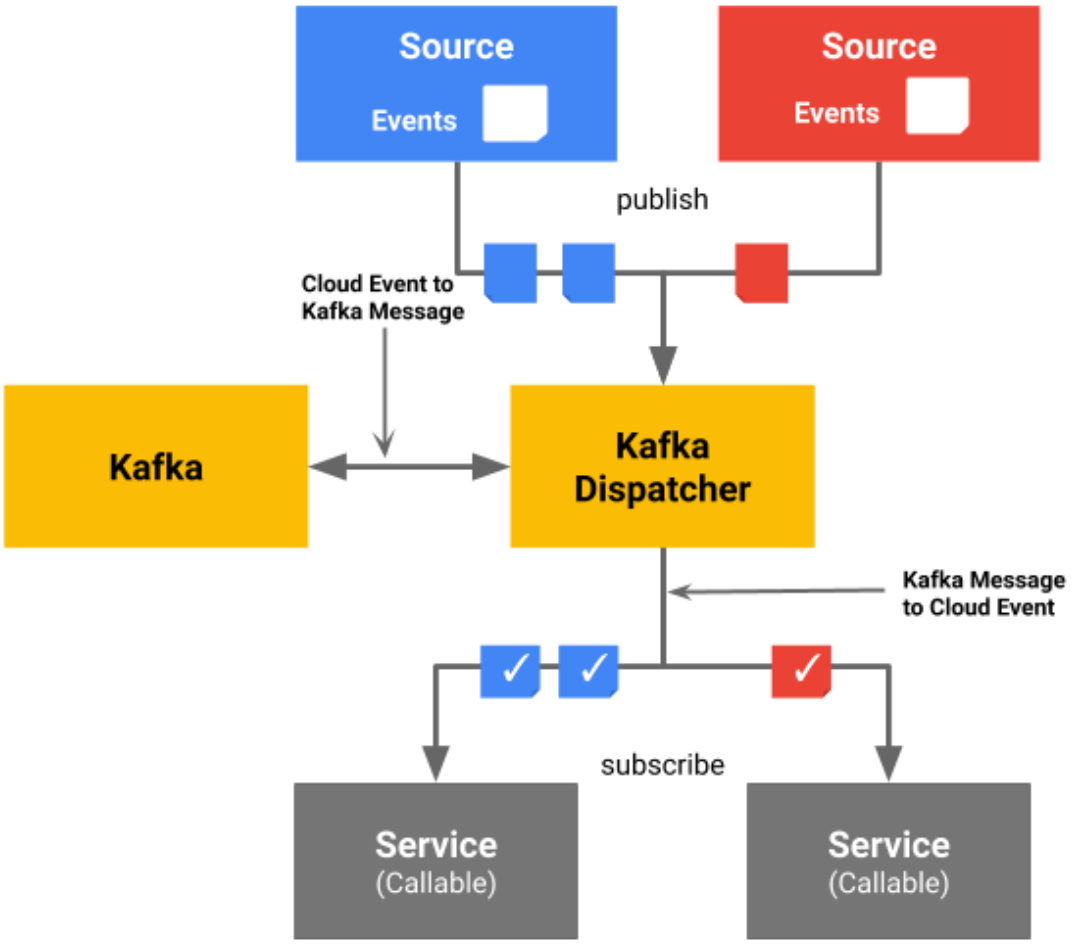
The first unnecessary network hop is $Broker\ Ingress \rightarrow Kafka\ Channel\ Dispatcher$.
This network hop is unnecessary because the *Broker Ingress* just forwards events to the *Kafka Channel Dispatcher*.
We can have only one component to handle all incoming messages, hence the component will have to:

1. Accept cloud events
2. Map cloud events structure to Kafka messages following the Kafka protocol binding specification
3. Send *Kafka* messages to *Kafka*

4. Acknowledge the event source

Other unnecessary network hops are $Kafka\ Channel\ Dispatcher \rightarrow Triggers$.
Each event read from Kafka is sent to all associated triggers and if the given event matches the user-specified filters the trigger will send the event to the *Callable* service.
A solution to this problem is to have a component that will have to:

1. Read a message from Kafka
2. Map the Kafka message to Cloud events structure following the Kafka protocol binding specification
3. Apply a function that given a set of triggers and an event returns all matching triggers
4. Send the event to all subscribers of all matching triggers
5. Acknowledge the Kafka broker (also known as "committing the position of the consumer")

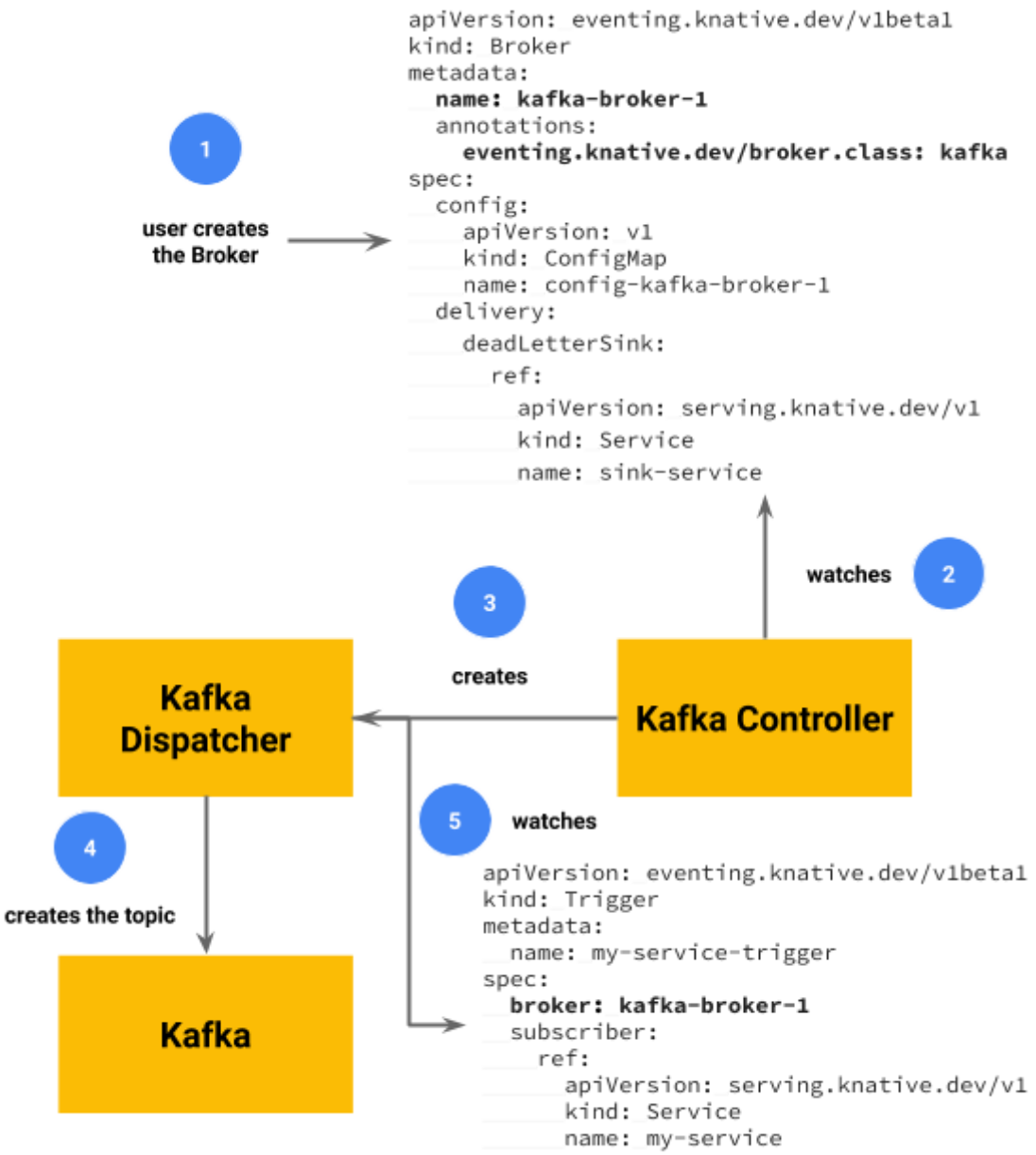


Alternative Kafka Dispatcher

An alternative Kafka Dispatcher design would be to have two components one responsible for accepting incoming events and sending them to Kafka and one responsible for reading events from Kafka and sending them to Callable services.

Having two components gives the possibility to scale the two independently from each other.

Control Plane



Delivery

The Broker Conformance Specification specifies that an event should *be delivered at least once to all subscribers of all Triggers* and the property is guaranteed because in case of a failure we won't acknowledge the Kafka broker and Kafka will resend the message to a consumer in the consumer group.

Scalability

All events will be sent to one Kafka topic, partitioned on a configurable number of partitions, per *Broker* with the annotation `*eventing.knative.dev/broker.class: kafka*`.

The *Kafka Dispatcher* component is a stateless component, hence it can scale by adding instances.

Implementation details

Data Plane

The data plane is composed of **KafkaReceiver** and **KafkaDispatcher**.

The *KafkaReceiver* will have a ***MessageReceiver*** responsible for accepting incoming events and a ***Producer*** responsible for forwarding events to Kafka.
The *KakaDispatcher* will have a **Consumer** responsible for reading events from *Kafka*, a **Filter** responsible for selecting to which *Callable* services a given event will be sent and a **Sender** responsible for sending events to services.

I'm considering implementing the data plane using Go or Java because Go has a robust integration with the Kubernetes ecosystem and Java has a stable integration with the Kafka ecosystem.
Which language to choose will be discussed with the community.

Go implementation

The alternate Broker proposal doesn't provide the *Channel* concept, therefore we have to create a slightly different version of the MessageReceiver.
A draft GitHub gist shows that most of the logic for the *MessageReceiver* can be shared between the channel based broker and the alternative broker without breaking consumers code; for this reason and also because those changes aren't related only to Kafka, I will make a PR against the `knative/eventing` repository for proposing the changes.
The *Producer* component will be implemented using an available *Kafka* client for *Go* and which library will be used is discussed in the paragraph Go Kafka Client.

The *Filter* will get a list of available *Triggers* associated with a given *Broker* through a TriggerInformer and it will have to have a method that given an event returns all matching Triggers. The *Filter* is not specific to *Kafka*, hence it can be placed in a shared package and reused in other implementations.
The *Consumer* component will be implemented using an available *Kafka* client for *Go* and which library will be used is discussed in the paragraph below.
The *Sender* component will be implemented using the MessageDispatcher.

Go Kafka Client

There are two up to date Kafka clients: Shopify/sarama and confluentinc/confluent-kafka-go.
The downside of the confluentinc/confluent-kafka-go is that it uses *Cgo*.
Now, the *knative/eventing-contrib* repository uses Shopify/sarama and also I used it in the past, therefore I would stay with Shopify/sarama. However, I plan to start a discussion with the community for the choice of the library.

Java Implementation

The *MessageReceiver* will be implemented using cloudevents/sdk-java that has a transport implementation based on Eclipse Vert.x Rx module which fits really well our use case.
The *Producer* will be implemented using the Apache Kafka Java client.

The *Filter* component will be implemented using the Kubernetes Java client using an *Informer*.
The library supports Kubernetes version 1.15 which is the minimum version required by Knative.
The *Consumer* will be implemented using the Apache Kafka Java client.

The *Sender* will be implemented using cloudevents/sdk-java.

Control Plane

The *KafkaController* will have to:

1. *Reconcile* the *Broker* (only those which have the `eventing.knative.dev/broker.class` annotation set to `kafka`)
2. *Reconcile* the data plane components:
    a. *KafkaReceiver*
    b. *KafkaDispatcher*
3. *Reconcile* the ingress service that points at the *KafkaReceiver* deployment

KafkaController won't use the existing broker reconciler because the reconciliation logic is different.

The *KafkaReceiver* is responsible for setting the status of Brokers.

The *KafkaDispatcher* is responsible for setting the status of Triggers.

4. Milestones

I plan to work for about 40 hours per week

Milestone	Tasks	Start Date	End Date
1 - Before the official start date		**23/03/2020**	**26/04/2020**
1.1	Familiarize with the code, familiarize with the community, familiarize with processes, solve open issues		
2 - Control plane		**27/04/2020**	**17/05/2020**
2.1	Broker reconciliation	27/04/2020	03/05/2020
2.2	Data plane reconciliation	04/05/2020	10/05/2020
2.3	Ingress reconciliation	11/05/2020	17/05/2020
2.4	Integration tests		
3 - Data plane		**18/05/2020**	**19/06/2020**
3.1	Implement KafkaReceiver	18/05/2020	24/05/2020
3.2	Implement KafkaDispatcher	25/05/2020	27/05/2020
3.3	Brokers and Trigger status as specified in Broker Conformance Spec	27/05/2020	07/06/2020
3.4	Documentation and integration tests	08/06/2020	14/06/2020
First evaluation		**15/06/2020**	**19/06/2020**
4 - Conformance Test and e2e tests		**19/06/2020**	**13/07/2020**
4.1	Conformance test	19/06/2020	28/06/2020
4.2	e2e tests	28/06/2020	13/07/2020
Second evaluation		**13/07/2020**	**17/07/2020**
5 - Performance tests		**17/07/2020**	**10/08/2020**
5.1	Performance tests to compare the channel based broker implementation	17/07/2020	26/07/2020
6 - Time Buffer		**27/07/2020**	**10/08/2020**
Final evaluation		**10/08/2020**	**17/08/2020**

5. Previous Open Source Pull Requests

knative/eventing-contrib

- [merged] NatssChannel Dispatcher Refactor to use CloudEvents
- [merged] Prune eventing-contrib/pkg/channel
- [merged] Port NatssChannel to cloudevents/sdk-go bindings
- [review] Migrate natss channel off reconciler.Base
- [merged] Migrate Kafka channel off reconciler.Base
- [review] Performance tests for Natss channel and Natss broker
- [merged] Remove unused handler in KafkaDispatcher

knative/eventing

- [hold] Check eventing.knative.dev/scope is not modified in InMemoryChannel and PingSource
- [merged] Update cloudevents sdk to release v2.0.0-preview7

apache/kafka

- [review] KAFKA-9088: Consolidate InternalMockProcessorContext and MockInternalProcessorContext

scylladb/gocqlx

- [merged] Add not equal comparators (!=)
- [merged] Move ne comparator below eq comparator

linkerd/linkerd2

- [merged] Fix bad request in the top routes tab on empty fields

6. Personal Details

Name	Someone
Surname	Somebody

Contacts and Professional Profiles

GitHub	https://github.com/someone
Email:	someone@example.com