DataStax

Building resilient and scalable API backends with Apache Pulsar and Spring Reactive

Presentation at ApacheCon@Home 2021
September 21, 2021

Lari Hotari @lhotari
Senior Software Engineer, DataStax, Inc

Apache Pulsar committer

Who

Lari Hotari   @lhotari

Software Engineer @DataStax, working on Luna Streaming powered by Apache Pulsar

Open-source contributor, Apache Pulsar committer

Agenda

01

Pulsar & Reactive Messaging

02

Scaling performance of message processing

03

Sample use case & Live coding

Agenda

01

Pulsar & Reactive Messaging

02

Scaling performance of message processing

03

Sample use case & Live coding

Apache Pulsar – Favourite properties

01 Streaming, pub-sub and queuing come together	**03** Cloud Native Architecture	Layered storage architecture
02 Many options for scaling - not just about partitions	First class support for Kubernetes	Stateless Brokers

Reactive Message handling – why?

Fault tolerance with **fallbacks, retries and timeouts**

Compatibility with Reactive Streams to achieve **reactive from end-to-end**

Performance by **parallelism and pipelining in processing**

Resilience with flow control (**backpressure**)

Simplification by **data oriented** & functional approach

Efficiency by optimal resource usage

Reactive Streams adapter library for Apache Pulsar

- https://github.com/lhotari/reactive-pulsar , License: ASL 2.0
- Wraps the Apache Pulsar Java Client async API with a simple and intuitive reactive API
- Goes beyond a wrapper
  - Back-pressure support for provided interfaces
  - Pulsar client resource lifecycle management
  - Message producer caching
  - In-order parallel processing at the application level
  - Spring Boot starter for auto configuring a Pulsar Java client and a ReactivePulsarClient

Reactive Messaging Application building blocks

provided by the Reactive Pulsar Adapter library

Message Sender

Message Reader

Message Consumer

Message Listener Container

- **Sender -** for sending messages with a specific configuration to a specific destination topic.
- **Reader** - The application decides the position where to start reading messages. Position can be earliest, latest, or an absolute or time based position. Suitable also for short-time message polling.
- **Consumer** - for guaranteed message delivery and handling. The broker redelivers unacknowledged messages. Used for both "always on" use cases and batch processing or short-time message consuming or polling use cases.
- **Message Listener Container -** integrates a consumer with the Spring application lifecycle.

*) The abstractions of the Reactive Pulsar Adapter library are slightly different from the abstractions in Pulsar Java Client. The sender is one example of a difference. The lifecycles of the abstractions are completely different. The ReactiveMessageSender, ReactiveMessageReader and ReactiveMessageConsumer instances themselves are stateless.
"Nothing happens until you subscribe" - the key difference in the Reactive programming model.

Basics of Reactive APIs

- “Nothing happens until you subscribe”
  - Assembly-time vs Runtime
- API calls return a reactive publisher type
  - Flux<T> or Mono<T> when using Project Reactor
  - Mono<Void> for calls that don’t return data

Reactive Pulsar Adapter

```java
public interface ReactivePulsarClient {

    <T> ReactiveMessageSenderBuilder<T> messageSender(Schema<T> schema);

    <T> ReactiveMessageReaderBuilder<T> messageReader(Schema<T> schema);

    <T> ReactiveMessageConsumerBuilder<T> messageConsumer(Schema<T> schema);

}
```

```java
public interface ReactiveMessageSender<T> {

    Mono<MessageId> sendMessage(Mono<MessageSpec<T>> messageSpec);

    Flux<MessageId> sendMessages(Flux<MessageSpec<T>> messageSpecs);

}
```

```java
public interface ReactiveMessageReader<T> {

    Mono<Message<T>> readMessage();

    Flux<Message<T>> readMessages();

}
```

```java
public interface ReactiveMessageConsumer<T> {

  <R> Mono<R> consumeMessage(

    Function<Mono<Message<T>>, Mono<MessageResult<R>>> messageHandler);

  <R> Flux<R> consumeMessages(

    Function<Flux<Message<T>>, Flux<MessageResult<R>>> messageHandler);

}
```

Agenda

01

Pulsar & Reactive Messaging

02

Scaling performance of message processing

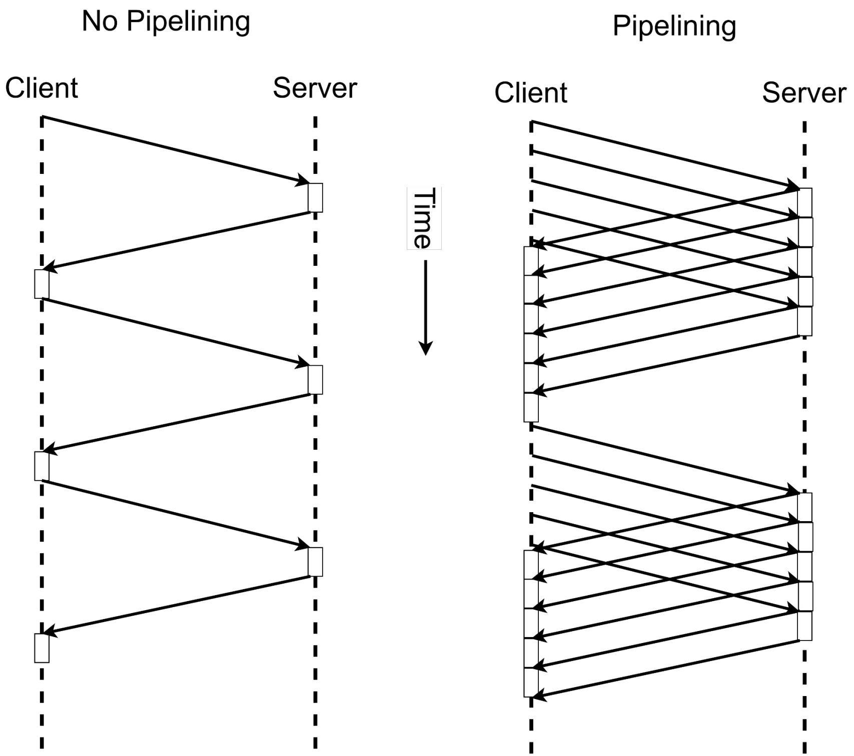
03

Sample use case & Live coding

All significant improvements in system scalability come from parallelism.

Pipelining

HTTP/ 1.1 protocol pipelining illustration



"This idea is an extension of the idea of parallelism. It is essentially handling the activities involved in instruction execution as an assembly line. As soon as the first activity of an instruction is done you move it to the second activity and start the first activity of a new instruction. This results in executing more instructions per unit time compared to waiting for all activities of the first instruction to complete before starting the second instruction."

https://www.d.umn.edu/~gshute/arch/great-ideas.html

In-order parallel processing strategies

- System level: Multiple topic partitions
  - Message is mapped to a specific partition by hashing the key

    ```
    partition = hash(message key) % number_of_partitions
    ```

- Application level: Message consumer parallelism
  - Micro-batching
    - Messages are consumed in batches which is limited by time and number of entries.
    - The processing can sort and group the entries for parallel processing.
  - Stream splitting / routing / grouping
    - Message is mapped to a specific processing group based on the hash of the message key

      ```
      processing group = hash(message key) % number_of_processing_groups
      ```

    - This is well suited for splitting the stream with Project Reactor's .groupBy
    - Benefit over micro-batching: no extra latency caused by batching

Parallel processing with ReactiveMessageConsumer

- An enabler is the the special signature of the consumeMessages method on the ReactiveMessageConsumer interface.
  - The method accepts a function that takes `Flux<Message<T>>` input and produces `Flux<MessageResult<R>>>` output.
  - MessageResult combines acknowledgement and the result. The implementation consumes the `Flux<MessageResult<R>>>` , handles the acknowledgements, and returns a `Flux<R>` .
  - This enables guaranteed message delivery combined with stream transformations.

```
public interface ReactiveMessageConsumer<T> {

  <R> Flux<R> consumeMessages(
    Function<Flux<Message<T>>, Flux<MessageResult<R>>>
    messageHandler);

}
```

```
public interface MessageResult<T> {
  static <T> MessageResult<T> acknowledge(MessageId messageId, T value) {}

  static <T> MessageResult<T> negativeAcknowledge(MessageId messageId, T
value) {}

  static MessageResult<Void> acknowledge(MessageId messageId) {}

  static MessageResult<Void> negativeAcknowledge(MessageId messageId) {}

  static <V> MessageResult<Message<V>> acknowledgeAndReturn(Message<V>
message) {}

  boolean isAcknowledgeMessage();
  MessageId getMessageId();
  T getValue();
}
```

Agenda

01

Pulsar & Reactive Messaging

02

Scaling performance of message processing

03

Sample use case & Live coding

Sample use cases – simplified IoT backend

https://github.com/lhotari/reactive-pulsar-showcase for complete sample (work in progress)

Telemetry ingestion
from IoT devices

Telemetry processing

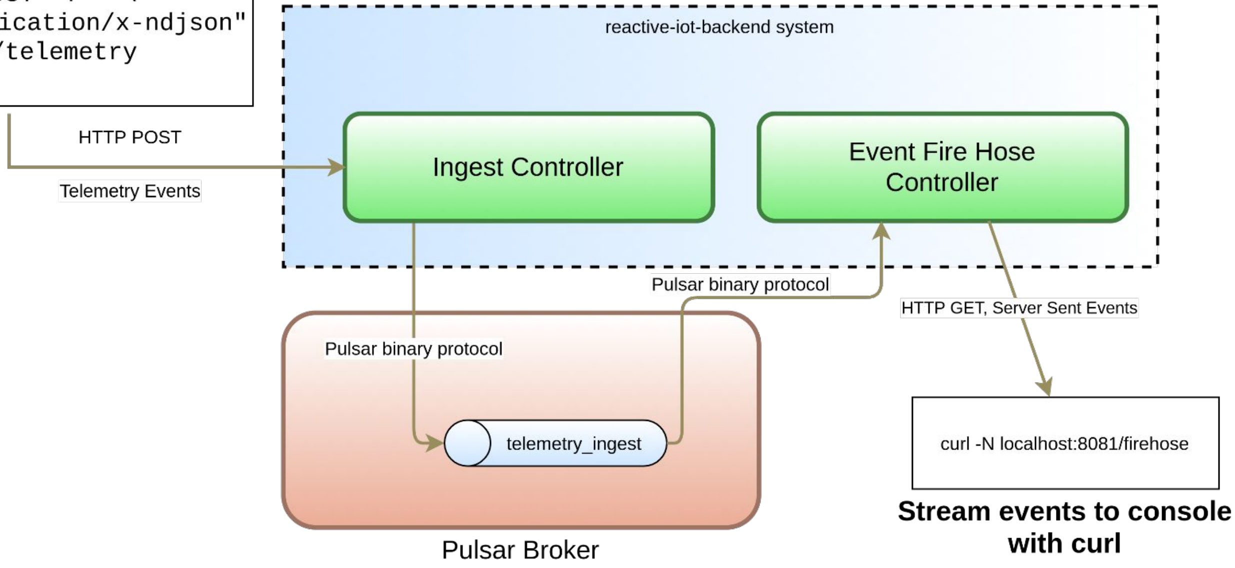
Sending alerts to
external application - webhook
interface

Streaming of telemetry values
to other applications

Our goal for live coding

Generate 1 million telemetry events with a shell script and curl

```
{ for i in {1..1000000}; do echo '{"n":
"device'$i'/sensor1", "v": '$i'.123}'; done;
} \
| curl -X POST -T - \
-H "Content-Type: application/x-ndjson"
localhost:8081/telemetry
```



Stream events to console with curl

Source code: https://github.com/lhotari/reactive-iot-backend-ApacheCon2021

```java
@RestController
public class IngestController {
   private final ReactiveMessageSender<TelemetryEvent> reactiveMessageSender;

   public IngestController(ReactivePulsarClient reactivePulsarClient,
                           ReactiveProducerCache reactiveProducerCache) {
       reactiveMessageSender = reactivePulsarClient
               .messageSender(Schema.JSON(TelemetryEvent.class))
               .topic("telemetry ingest")
               .cache(reactiveProducerCache)
               .maxInflight(100)
               .build();
   }

   @PostMapping("/telemetry")
   public Mono<Void> ingestTelemetry(@RequestBody Flux<TelemetryEvent> telemetryEventFlux) {
       return reactiveMessageSender
               .sendMessages(telemetryEventFlux.map(MessageSpec::of))
               .then();
   }
}
```

```java
@RestController
public class EventFireHoseController {
   private final ReactiveMessageReader<TelemetryEvent> reactiveMessageReader;

   public EventFireHoseController(ReactivePulsarClient reactivePulsarClient) {
       reactiveMessageReader = reactivePulsarClient
               .messageReader(Schema.JSON(TelemetryEvent.class))
               .topic("telemetry ingest")
               .startAtSpec(StartAtSpec.ofEarliest())
               .endOfStreamAction(EndOfStreamAction.POLL)
               .build();
   }

   @GetMapping("/firehose")
   public Flux<ServerSentEvent<TelemetryEvent>> firehose() {
       return reactiveMessageReader
               .readMessages()
               .map(message -> ServerSentEvent.builder(message.getValue())
                       .id(message.getMessageId().toString())
                       .build());
   }
}
```

References

Apache Pulsar: https://pulsar.apache.org/

Spring Reactive: https://spring.io/reactive

Reactive Pulsar adapter: https://github.com/lhotari/reactive-pulsar

Status: *experimental* version 0.1.0 available for use, API is subject to change until 1.0 is released.

Reactive Pulsar showcase application: https://github.com/lhotari/reactive-pulsar-showcase

Status: work in progress, demonstrates the usage of the Reactive Pulsar Adapter.

Live coding source code: https://github.com/lhotari/reactive-iot-backend-ApacheCon2021

For usage questions, please use apache-pulsar and reactive-pulsar tags on Stackoverflow.

Thank you !

We are hiring: https://www.datastax.com/company/careers