Guaranteed “effectively-once” messaging semantic

Matteo Merli
What is Apache Pulsar?

- Distributed pub/sub messaging
- Backed by a scalable log store — Apache BookKeeper
- Streaming & Queuing
- Low latency
- Multi-tenant
- Geo-Replication
• Separate layers between brokers bookies

• Broker and bookies can be added independently

• Traffic can be shifted very quickly across brokers

• New bookies will ramp up on traffic quickly
Messaging model

Exclusive
- Subscription A
  - Consumer A-0

Failover
- Subscription B
  - Consumer B-0
  - Consumer B-1
  - In case of failure in Consumer B-0

Shared
- Subscription C
  - Consumer C-0
  - Consumer C-1
  - Consumer C-2
  - Consumer C-3

Streaming

Queuing
Messaging semantics

At most once

At least once

Exactly once
“Exactly once”

- There is no agreement in industry on what it really means.
- Any vendor has claimed exactly once at some point.
- Many caveats… “only if there are no crashes…”
- No formal definition of exactly once — unlike “consensus” or “atomic broadcast”
“Effectively once”

• Identify and discard duplicated messages with 100% accuracy

• In presence of any kind of failures

• Messages can be received and processed more than once

• …but *effects on the resulting state will be observed only once*
What can fail?

Producer A

Consumer 1

Broker 1

Broker 2

Broker 3
What can fail?

Producer A

Consumer 1

Broker 1

Broker 2

Broker 3
What can fail?

- Producer A
- Consumer 1
- Broker 1
- Broker 2
- Broker 3

Streamio
What can fail?

- Producer A
  - Broker 1
  - Broker 2
  - Broker 3
- Consumer 1
What can fail? — Geo-Replication
Breaking the problem

1. Store the message once — "producer idempotency"
2. Allow applications to “process data only-once”
Idempotent producer

- Pulsar broker detects and discards messages that are being retransmitted
- It works when a broker crashes and topic is reassigned
- It works when a producer application crashes
Identifying producers

- Use “sequence ids” to detect retransmissions
- Each producer on a topic has its own sequence of messages
- Use “producer-name” to identify producers
Detecting duplicates
Detecting duplicates

- Producer-A
  - A-3
  - Last sequence id pushed
    - Producer-A: 2
    - Producer-B: 1

- Producer-B
  - B-2
  - Last sequence id stored
    - <empty>

- Pulsar Broker

- BookKeeper Storage
  - A-2
  - B-1
  - A-1

- Snapshot
Detecting duplicates

Pulsar Broker

Last sequence id pushed
Producer-A: 3
Producer-B: 2

Last sequence id stored
<empty>

Snapshot

BookKeeper Storage

Producer-A

Producer-B

A-5
A-4
B-3
A-2
A-1
A-3
B-2
B-1

Detecting duplicates

- **Producer-A**
  - A-1
  - A-2
  - A-6
  - A-7

- **Producer-B**
  - B-1
  - B-4

**Pulsar Broker**
- Last sequence id pushed
  - Producer-A: 5
  - Producer-B: 3

**Last sequence id stored**
- Producer-A: 2
- Producer-B: 1

**BookKeeper Storage**
- A-4
- A-5
- B-2
- B-3
- A-3
- A-2
- B-1
- B-4

**Snapshot**
Sequence Id snapshot

Snapshot Cursor
- A=4
- B=2

A-1 A-2 B-1 A-3 B-2 A-4 B-3 A-5 A-6 B-4 A-7 ...

New messages
Sequence Id snapshot

Snapshot Cursor
A=6
B=4

A-1 A-2 B-1 A-3 B-2 A-4 B-3 A-5 A-6 B-4 A-7 ...

New messages
Sequence Id snapshot

- Snapshots are taken every N entries to limit recovery time
- Snapshot & cursor updates are atomic
- Cursor updates are stored in BookKeeper — durable & replicated
- On recovery
  - Load the snapshot from the cursor
  - Replay the entries from the cursor position
What if application producer crashes?

- Pulsar needs to identify the new producer as being the same “logical” producer as before

- In practice, this is only useful if you have a “replayable” source (e.g., file, stream, …)
Resuming a producer session

ProducerConfiguration conf = new ProducerConfiguration();
conf.setProducerName("my-producer-name");
conf.setSendTimeout(0, TimeUnit.SECONDS);
Producer producer = client.createProducer(MY_TOPIC, conf);

// Get last committed sequence id before crash
long lastSequenceId = producer.getLastSequenceId();
Using sequence Ids

// Fictitious record reader class
RecordReader source = new RecordReader("/my/file/path");

long fileOffset = producer.getLastSequenceId();
source.seekToOffset(fileOffset);

while (source.hasNext()) {
    long currentOffset = source.currentOffset();
    Message msg = MessageBuilder.create()
        .setSequenceId(currentOffset)
        .setContent(source.next()).build();

    producer.send(msg);
}
Consuming messages only once

- Pulsar Consumer API is very convenient
  
  - Managed subscription — tracking individual messages

```java
Consumer consumer = client.subscribe(MY_TOPIC, MY_SUBSCRIPTION_NAME);

while (true) {
    Message msg = consumer.receive();
    // Process the message...
    consumer.acknowledge(msg);
}
```
Effectively-once with Consumer

- Consumer is very simple but doesn’t allow a large degree of control
- Processing and acknowledge are not atomic
- To achieve “effectively once” we need to rely on an external system to deduplicate the processing results. Eg:
  - RDBMS — Keep the message id as a column with a “unique” index
  - Critical write to update the state — `compareAndSet()` or similar
Pulsar Reader

• Reader is a low level API to receive data from a Pulsar topic

• There is no managed subscription

• Application always specifies the message id where it wants to start reading from
Reader example

```java
MessageId lastMessageId = recoverLastMessageIdFromDB();
Reader reader = client.createReader(MY_TOPIC, lastMessageId,
    new ReaderConfiguration());

while (true) {
    Message msg = reader.readNext();
    byte[] msgId = msg.getMessageId().toByteArray();

    // Process the message and store msgId atomically
}
```
Example — Pulsar Functions

Source Topic

\[ f(x) \]

Sink Topic

Source Topic

Sink Topic

Source Topic

Sink Topic
Pulsar Functions

- A function gets messages from 1 or more topics
- An instance of the function is invoked to process the event
- The output of the function is published on 1 or more topics
- Super simple to use — No SDK required — Python example:

```python
def process(input):
    return input + '!!'
```
Pulsar Functions

Source Topic → Consumer → $f(x)$ → Producer → Sink Topic
Effectively once with functions

- Use the message id from source topic as sequence id for sink topic
- Works with “Consumer” API
- When consuming from multiple topics or partitions, creates 1 producer per each source topic/partition, to ensure monotonic sequence ids
Performance

- Pulsar approach guarantees deduplication in all failure scenarios
- Overhead is minimal: 2 in memory hashmap updates
- No reduction in throughput — No increased latency
- Controllable increase in recovery time
Performance — Benchmark

- OpenMessaging Benchmark
  - 1 Topic / 1 Partition
  - 1 Partition / 1 Consumer
  - 1Kb msg
## Difference with Kafka approach

<table>
<thead>
<tr>
<th></th>
<th>Kafka</th>
<th>Pulsar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Producer Idempotency</strong></td>
<td>Best-effort (in memory only)</td>
<td>Guaranteed after crash</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>2 phase commit</td>
<td>No transactions</td>
</tr>
<tr>
<td><strong>Dedup across producer sessions</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Dedup with geo-replication</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>Lower (1 in-flight message/batch for ordering)</td>
<td>Equal</td>
</tr>
</tbody>
</table>
Curious to Learn More?

• Apache Pulsar — https://pulsar.incubator.apache.org

• Follow Us — @apache_pulsar

• Streamlio blog — https://streaml.io/blog