Making sense of exactly-once semantics

Flavio Junqueira
About me

- **Confluent**
  - Infrastructure Engineer
  - Kafka Core
- **Apache Software Foundation (ASF)**
  - Apache ZooKeeper, BookKeeper, Kafka
  - Apache Incubator
- **Previously**
  - Yahoo! Research and Microsoft Research
Why this talk...
Apache Flink

The problem of providing exactly once guarantees really boils down to determining what state the streaming computation currently is in … rewinding the stream source (for example with help of Apache Kafka) to the point when the snapshot was taken and hitting the play button again.


Apache Spark

Exactly-once semantics: … we use simple Kafka API … each record is received by Spark Streaming effectively exactly once despite failures.

Apache Kafka guarantees at least once
Towards exactly-once in Apache Kafka
Context
Send £100 check to pay bill by mail.

Acme Ltd.
Post Office on Fire

The main post office was set on fire yesterday by an unhappy customer who hasn't had his letters delivered correctly for the past 5 years. Witnesses said that the customer has started the fire while shouting "No more paper oppression!". The customer escaped and was identified by multiple other customers as a Strata London speaker. Unfortunately, multiple letters and parcels have been completely destroyed. The postal service apologizes for the inconvenience, but asks its customers to resend when possible.
Hmm, should I send it again?

Acme Ltd.
Send another £100 check to pay bill by mail.

Acme Ltd.
• Expect to pay only 100 and not 200
• Check name or customer id
Make actions idempotent
# Remote Reference/Remote Operation Model

<table>
<thead>
<tr>
<th>Protocol Class</th>
<th>No Failures</th>
<th>Lost Packets</th>
<th>Lost Packets &amp; Slave Failure</th>
<th>Lost Packets &amp; Master Failure</th>
<th>Lost Packets, Master &amp; Slave Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maybe</td>
<td>op performed: 1</td>
<td>op performed: 0,1</td>
<td>op performed: 0,1</td>
<td>op performed: 0,1</td>
<td>op performed: 0,1</td>
</tr>
<tr>
<td></td>
<td>result-commit: 1</td>
<td>result-commit: 0,1</td>
<td>result-commit: 0,1</td>
<td>result-commit: 0</td>
<td>result-commit: 0</td>
</tr>
<tr>
<td>At-Least-Once</td>
<td>op performed: 1</td>
<td>op performed: ≥ 1</td>
<td>op performed: ≥ 0</td>
<td>op performed: ≥ 0</td>
<td>op performed: ≥ 0</td>
</tr>
<tr>
<td></td>
<td>result-commit: 1</td>
<td>result-commit: ≥ 1</td>
<td>result-commit: ≥ 0</td>
<td>result-commit: 0</td>
<td>result-commit: 0</td>
</tr>
<tr>
<td>Only-Once-Type-1</td>
<td>op performed: 1</td>
<td>op performed: 1</td>
<td>op performed: 0,1</td>
<td>op performed: 0,1</td>
<td>op performed: 0,1</td>
</tr>
<tr>
<td></td>
<td>result-commit: 1</td>
<td>result-commit: 1</td>
<td>result-commit: 0,1</td>
<td>result-commit: 0</td>
<td>result-commit: 0</td>
</tr>
<tr>
<td>Only-Once-Type-2</td>
<td>op performed: 1</td>
<td>op performed: 1</td>
<td>op performed: 1</td>
<td>regular master process</td>
<td>regular master process</td>
</tr>
<tr>
<td></td>
<td>result-commit: 1</td>
<td>result-commit: 1</td>
<td>result-commit: 0</td>
<td>op performed: 1</td>
<td>op performed: 1</td>
</tr>
</tbody>
</table>

Remote Reference/Remote Operation Model

Master

Worker

inc(x)

inc(x)

x = 10

x = 11

x = 11
Remote Reference/Remote Operation Model

Master

\text{inc}(x)

Worker

\text{inc}(x)

x = 10

\textbf{Crash}

Worker has crashed

x = 10
Remote Reference/Remote Operation Model

Master

Worker

Network error

\( \text{inc}(x) \to x = 10 \)

\( x = 10 \)
Remote Reference/Remote Operation Model

Master

inc(x)

Worker

x = 10
↓
x = 11

11

Retry!
Remote Reference/Remote Operation Model

Master

inc(x)

Worker

x = 10

\[ \downarrow \]

x = 11

Crash

Oh no! Executed before crashing

x = 11
Remote Reference/Remote Operation Model

Oh no! Network error on the response
Remote Reference/Remote Operation Model

Retrying gives us the wrong behavior for only-once.
Remote Reference/Remote Operation Model

Give each operation a name, an identifier
Remote Reference/Remote Operation Model

Give each operation a name, an identifier

- Master
  - $bob$, inc($x$)
  - name, inc($x$)
  - name, 11

- Worker
  - $x = 10$
  - $x = 11$

- Result
  - $bob$, result 11
    - $x = 11$
Remote Reference/Remote Operation Model

Master

bill, inc(x)
bob, inc(x)

Worker

x = 10
x = 11

Give each operation a name, an identifier

bob, result 11
x = 11
Remote Reference/Remote Operation Model

Master

\[ \text{name, inc(x)} \]

Worker

\[ \text{x = 10} \]
\[ \downarrow \]
\[ \text{x = 11} \]

Crash

\[ \text{name, result 11} \]
\[ \text{x = 11} \]

What if I retry now?
Remote Reference/Remote Operation Model

A retry doesn't increment it again
Remote Reference/Remote Operation Model

Name makes it possible to spot duplicates
Remote Reference/Remote Operation Model

Master

\(name, \text{inc}(x)\)

Worker

\(x = 10\)
\(\downarrow\)
\(x = 11\)

Add ack response to release resources

\(name, \text{result} \ 11\)
\(x = 11\)
Names

• Remote reference/Remote operation model

• Two levels
  • **Session**: logical connection between master and worker
  • **Sequence number**: order of request
What about large-scale?

• Multiple servers and clients
• Servers can crash
• Objects can migrate

• Challenges
  • System-wide mechanism for assigning unique client identifiers
  • Completion record durability
  • Match retries
  • Garbage collection

• *Reusable Infrastructure For Linearizability* [Lee et al., SOSP’15]
  • Can't guarantee linearizability in the presence of retries
Consensus and Exactly once
Agreement on the sequence of requests to be executed
Agreement

Replicas have a consistent state

Execute requests in the sequence agreed upon
Agreement

Replicas have a consistent state

Replicas deduplicate by processing the sequence of requests
Using agreement in practice

Replicated Service

Client

RPC

Replica 1

Replica 2

Replica 3
Using agreement in practice

Naturally deduplicated:
- create persistent and ephemeral
- conditional setData and delete
Using agreement in practice

Apache ZooKeeper

Client

Server 1
Server 2
Server 3

create
setData
delete

... but not create sequential!
Using agreement in practice

First attempt creates /test-1

Apache ZooKeeper

Client

create
/test-, sequential

Server 1
/test-1

Server 2
/test-1

Server 3
/test-1
Using agreement in practice

Second attempt creates /test-2, which is undesirable
Retries

• ZooKeeper client currently does not retry
  • Error upon connection loss
• Possible by checking operation numbers
How does it help with building real-time message processing?
Exactly-once for Messaging
Messaging simplified

Producer  msgs  $m_3 \rightarrow m_2 \rightarrow m_1$  Consumer
  msgs  Consumer
Messaging simplified

Messages are persisted and appended to a log

(see it as a replicated state machine)

Apache Kafka
Messaging simplified

Idempotent produce calls

Deduplicate while consuming

Producer → msgs → $m_1, m_2, m_3$ → Consumer

msgs → Consumer

msgs → Consumer
Messaging simplified

Producer → msgs → \[ m_3 \quad m_2 \quad m_1 \] → Consumer

- If done by consumer only
- Difficult to resume from arbitrary position
- If broker checks for duplicates
- Better off discarding when produced
Messaging simplified

Focus on making produce calls idempotent
Messaging

Producer: `produce(msg)`

Broker: `msg`

OK

msg

...
Messaging

Producer
produce(msg)

Broker

msg
msg

OK

msg

...
Messaging

Producer

produce(msg)

OK

msg

Broker

msg msg

msg msg

msg ...

Retrying produces a duplicate
Messaging - Avoiding the duplicate

Identifiers prevent duplicates

Producer:
produce(n, msg)

OK

Broker:

n, msg

Check if id exists

...
Looking more closely

• Id guarantees no more than once

• **At least once**
  ✓ Retry upon network error
  ✓ Retry forever? How long is enough?

• **At most once - Id verification**
  ✓ Use separate data structure?
  ✓ Store all identifiers ever seen?
At least once

Retry an unbounded number of times, until done!
Safe, but not live...
At least once

User events
Search queries
Ad clicks
etc.

Producing events...

Buffer

Producer

Retrying indefinitely causes the buffer to fill up.
At least once

Eventually, we have to start dropping events

User events
Search queries
Ad clicks
etc.

Producer

Buffer

Produce($n_1, m_1$)
At least once

User events
Search queries
Ad clicks

etc.

Producer

Buffer

\[ m_n \quad \ldots \quad m_2 \quad m_1 \]

Produce\((n_1, m_1)\)

... and the at-least-once guarantee is violated
At least once

User events
Search queries
Ad clicks
*etc.*

... unless you can apply back pressure
At least once

User events
Search queries
Ad clicks
etc.

Producer

Buffer

\[ \text{Produce}(n_1, m_1) \]

\[ m_n \cdots m_2 m_1 \]

... or leak to disk, but that's also bounded
At least once

• To enforce it
  ✓ Might need to stall
  ✓ Not live for some unknown period
  ✓ Pressure back to source

• When progress is strictly necessary, drop
Id verification

Produce($n_1, m_1$) → Have I seen this id before? → no, append
→ yes, duplicate
Have I seen this Id before?

• Choices
  • Sequence number?
  • Arbitrary byte vectors or strings?
• Critical for efficient processing
Sequence numbers

Produce\((2, m_2)\)
Produce\((1, m_1)\)

Next seq. \# = 1

append

✓ Expected sequence number is 1
Sequence numbers

Produce(2, \(m_2\)) \[\rightarrow\] \text{Next seq. \# = 2} \[\xrightarrow{\text{append}}\] \(m_1\)

✓ Message is successfully produced
✓ Advance expected sequence number to 2
Sequence numbers

Next seq. # = 3

append

✓ Advance expected sequence number to 3
Sequence numbers

Produce\((1, m_1)\) \rightarrow \text{Next seq. } \# = 3 \rightarrow \text{append} \rightarrow m_2 \ m_1

\checkmark \text{Produce request fails}
\checkmark \text{Expected is 3}
\checkmark \text{Produce request is 1}
Where?

- Remote Reference/Remote Operation Model
- Kafka - Idempotent producer (proposal)
  - Unique identifiers for produced messages
  - \( ID = \text{Producer Id} + \text{Epoch} + \text{Seq. #} \)
- RIFL - SOSP’15
  - Linearizability for RPCs
Caveat - Sequence number assignment

App

Send \( m_2 \)
Send \( m_1 \)

Producer

\( m_2, 2 \)
\( m_1, 1 \)

Next seq. # = 3

Produce \( m_2, 2 \)
Produce \( m_1, 1 \)
Caveat - Sequence number assignment

App

Send $m_2$
Send $m_1$

Producer

$\text{Produce } m_2, 2$
$\text{Produce } m_1, 1$

Next seq. # = 3

Crash
Caveat - Sequence number assignment

App

Send $m_2$
Send $m_1$

Producer

$m_2$, 2
$m_1$, 1

Next seq. # = 3

Produce $m_2$, 2
Produce $m_1$, 1

Crash, and upon recovery, must be able to assign the same sequence numbers
Caveat - Sequence number assignment

• Two options
  ✓ Start from scratch (first message)
  ✓ Persist \( \langle \text{source offset, seq. #} \rangle \) pairs
Arbitrary Identifiers

Produce\((n_2, m_2)\)
Produce\((n_1, m_1)\)

isDuplicate(id)

append

Relies on a separate data structure to spot duplicates
Arbitrary Identifiers

Produce($n_2, m_2$) → isDuplicate(id) → append

If id is unknown, add to list and produce
Arbitrary Identifiers

If id is unknown, add to list and produce
Arbitrary Identifiers

isDuplicate(id)

append

List can grow without bounds

List can grow without bounds
Arbitrary Identifiers

Use Bloom Filters!

Bloom filters!
Arbitrary Identifiers

Use Bloom Filters!
Reduce the footprint of ids

Bloom filters!
Arbitrary Identifiers

isDuplicate(id)

append

Piggyback acks to free resources

\[
\begin{align*}
&n_1 \\
&n_2 \\
&\ldots
\end{align*}
\]
Where?

- Google Millwheel  
  ✓ Low-latency data processing  
  ✓ Unique ids and Bloom filters of record fingerprints  
  ✓ Bigtable as a backend  

- Google Photon  
  ✓ Event id: server IP + process id + timestamp  
  ✓ Multi-DC Paxos DB for deduplication  

[Tyler Akidau et al., VLDB 2013]  
[R. Ananthanarayanan et al., SIGMOD 2013]
Let’s assume sequence numbers for the next part
Exactly-once for Streams
Abstracting streams

Function $f$ transforms the messages of the input stream (e.g., filter)
Simple streams example

Say we are simply filtering
Simple streams example

- Message $m_1$ passes the filter
- Produced to the output stream
Simple streams example

- Message $m_2$ passes the filter
- Request fails, retry
Simple streams example

- Message $m_2$ passes the filter
- Request fails, retry
Simple streams example

- Message $m_2$ passes the filter
- Request fails, retry
Simple streams example

- Message $m_2$ passes the filter
- Requires deduplication!
Deduplication isn’t always sufficient
...because of non-determinism
Two input sources

- Two input streams
- Two input Kafka partitions
Two input sources

Output order depends on how the input is scanned.
Two input sources

Even if scan order is deterministic, timing is important
Two input sources

Even if scan order is deterministic, timing is important
Recommendations

Weather forecast

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Sunny</td>
</tr>
<tr>
<td>y</td>
<td>Rain!</td>
</tr>
</tbody>
</table>

Output stream

Join

Location updates

$l_1$: Device $d$, Location $x$

Outdoors recommendation + Forecast update
Recommendations

Weather forecast
Location updates

Join

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Sunny</td>
</tr>
<tr>
<td>y</td>
<td>Rain!</td>
</tr>
</tbody>
</table>

$O_1$: great weather for outdoors
Recommendations

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Sunny</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

Weather forecast

Location updates

\( l_1 \): Device \( d \), Location \( x \)

Join

Output stream

\( o_1 \): great weather for outdoors
Recommendations

Weather forecast

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

Location updates

\( l_1 \): Device \( d \), Location \( x \)

\( u_1 \): Location \( x \), Tornadoes

Output stream

\( o_1' \): great weather for outdoors

\( o_1 \): warning!
Recommendations

Weather forecast

Location updates

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

Join

Output stream

Duplicate

Outdoors recommendation + Forecast update

$O_1$: great weather for outdoors

$O'_1$: warning!

$l_1$: Device $d$, Location $x$

$u_1$: Location $x$, Tornadoes
Recommendations

Weather forecast

Location updates

Join

Output stream

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

$u_1$: Location $x$, Tornadoes

$u_1$: Device $d$, Location $x$

$u_1$: Location $x$, Tornadoes

$u_1$: tornadoes

$o_1$: great weather for outdoors

Outdoors recommendation

Forecast update
Recommendations

Weather forecast

Location updates

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

Outdoors recommendation

Inconsistent

Device $d$, Location $x$

Location $x$, Tornadoes

Output stream

$u_1$, $o_1$

Forecast update

$o_1$: great weather for outdoors

$u_1$: tornadoes
Recommendations

Weather forecast

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

Join

Output stream

Outdoors recommendation + Forecast update

Discard!

$l_1$: Device $d$, Location $x$

$u_1$: Location $x$, Tornadoes
Recommendations

**Weather forecast**

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>$y$</td>
<td>Rain</td>
</tr>
</tbody>
</table>

**Location updates**

$l_1$: Device $d$, Location $x$

$u_1$: Location $x$, Tornadoes

**Output stream**

$u_1$ $o'_1$

**Outdoors recommendation**

$o'_1$: warning!

$u_1$: tornadoes

**Consistent**
Discarding messages

• Downstream
  • Separate tentative from stable
  • Use special marker messages

• Markers
  • Stable: messages are ready to be consumed
  • Discard: discard the last batch of messages
Two input sources
Two output partitions

- Multiple output partitions
- Output is sharded by key
Recommendations

Weather forecast

Location updates

Forecast update

Join

<table>
<thead>
<tr>
<th>Location</th>
<th>Forecast</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tornadoes</td>
</tr>
<tr>
<td>y</td>
<td>Rain</td>
</tr>
</tbody>
</table>

\( u_1 \): Location \( x \), Tornadoes

\( l_1 \): Device \( d \), Location \( x \)

\( u_1 \): Location \( x \), Tornadoes

\( o_1 \): great weather for outdoors

\( u_1 \): tornadoes
Transactional

- Batch of output messages
- All partitions are made stable or none
- Use a commit protocol to guarantee atomicity
- **Atomicity:** Eventually all tentative messages are made either stable or discarded.
Two input sources

Input stream

Input stream

Output stream

Output stream

filter

Prepare

flush

Produce prepare

Commit log
Two input sources

Input stream

\[ m_1 \]

Input stream

\[ m_2 \]

Commit

\[ S \quad m_1 \]

Output stream

\[ S \quad m_2 \]

Output stream

\[ C \quad P \]

Commit log

filter

Produce commit
Wrap-up
Consuming

Exactly-onced Kafka Topics

Apache Flink
Apache Spark
Kafka Connectors
Consuming

m_1

m_2

Apache Flink
Apache Spark
Kafka Connectors

Exactly-onced
Kafka Topics

Application involvement for
end-to-end guarantee
Summing up

- Exactly-once semantics for RPCs
- State-Machine replication
- Exactly-once for messaging
  - Retries and deduplication
- Exactly-once for streams
  - Not sufficient to deduplicate messages
  - Require the ability of discarding partial state
Future

Deduplicated topics

Exactly-once for Kafka Connectors

Apache Kafka

Exactly-once for Kafka Streams
Acknowledgments

Thanks to my Confluent colleagues:
✓ Eno Thereska
✓ Neha Narkhede
✓ Ismael Juma
✓ Ben Stopford
✓ Jun Rao
✓ Jay Kreps
✓ Guozhang Wang
✓ Ewen Cheslack-Postava
✓ Sriram Subramanian
Questions?

e-mail: fpj@apache.org
twitter: @fpjunqueira
web site: http://fpj.me

Visit us in the Confluent Booth (#217)
- Kafka: The Definitive Guide = Book Giveaway and Signing
- Making Sense of Stream Processing = Book Giveaway

Kafka Training with Confluent University
- Kafka Developer and Operations Courses
- Visit www.confluent.io/training

Want more Kafka?
- Apache Kafka 0.10 upgrade documentation at http://docs.confluent.io/3.0.0/upgrade.html
- Kafka Summit recordings now available at http://kafka-summit.org/schedule/