CONFLUENT

Distributed Event-Level Tracing with Event Lineage Accelerator

Confluent CSID Labs

January 2023

What we will cover

Accelerator Assumptions, Prerequisites, Requirements

What does the Event Lineage Accelerator do?

How does it work?

Use case samples

Assumptions, Prerequisites, Requirements

CSID Event Lineage Accelerator Requires:

- Confluent 6.0.x or newer
- Java 8.0 or higher
- Existing or in-progress OpenTelemetry implementation

Accelerators, like Event Lineage, are not full-fledged products:

- **Requires Professional Services engagement for 8x5 support**
- **No additional support or SLA**
- Requires issuance of License and acceptance of Terms & Conditions

Code Delivery includes compiled and source libraries.

Typical implementation and guidance through testing will be estimated individually per customer use case.

Your requirements may require additional time for these scenarios:

- Clients using other languages or frameworks: e.g. Python, .NET, Node.js, C++ are not currently supported.
- Specific feature requirements that are not currently included

Only self-managed Connect cluster tracing is supported.
Tracing of ksqlDB is not yet supported.

CONFLUENT

What does the Event Lineage Accelerator do?

Customers could use this for...

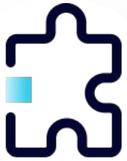
Enables understanding of event flow across the data pipeline, on an individual event level

Audit

It can provide a record of an event processing chain including event origin and a trail of touchpoints.

Root Cause Analysis

It has the ability to look up all messages leading to a given event or processing step to determine the source of the issue.
It can support log enrichment with event context data, which enables per-event log filtering



Data Analysis

The collected trace data enables event flow analytics such as event correlation, trend and performance analysis.

Data Visualization and Alerting

It supports the ability to build data visualizations and alerts, event flow visualization and outlier alerting

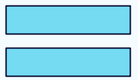
How does it work?

It is an extension to Open Telemetry and Distributed Tracing

Distributed Tracing Concepts
Distributed tracing is the capability for a tracing solution to track and observe events as they flow through distributed systems by collecting data as the events go from one service to another.

An Implementation Suite
Open Telemetry is a Distributed Tracing implementation suite that supports trace data capture from various libraries and frameworks, including Java Kafka Consumer and Producer libraries.

The Event Lineage accelerator extends this instrumentation support to Kafka Streams and Kafka Connect.



Confluent customers can understand where a Kafka message originated, what topics it went through, and which other events contributed to the event creation.

Event Lineage components overview

1. Capture

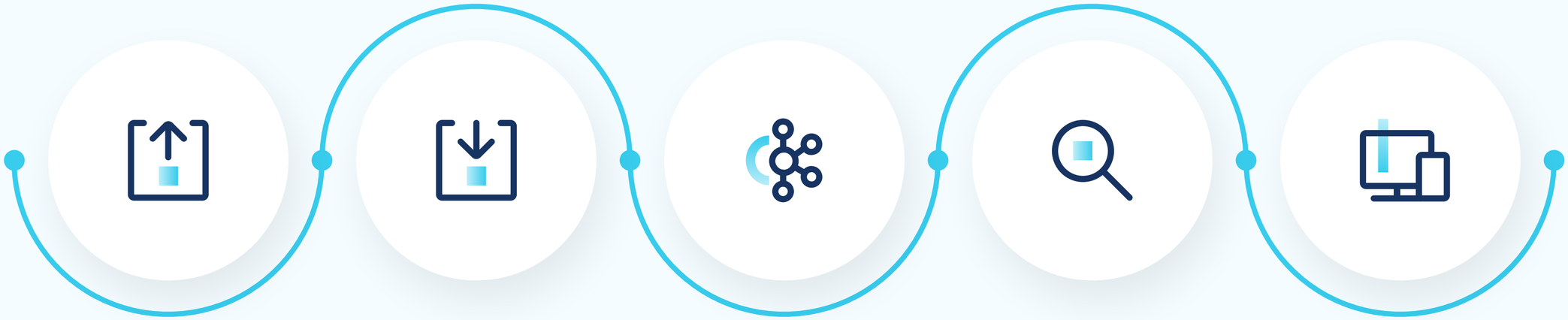
Trace propagation across components and trace metadata capture with OpenTelemetry Javaagent instrumentation and extensions.

3. Store

- Pluggable options for trace metadata storage.
- Platform to run analytics

5. Access

- UI for visualising trace data such as event flow, outliers, dashboards based on trace analysis.



2. Collect

OpenTelemetry Collector - collects data from agents, transforms and exports data to configured backends

4. Analyse

- Data Analytics can be performed on batch to identify correlations etc.
- Streaming processing of ingested data.
- Allow Field categorisation.
- Alerts based on thresholds.

Why is it different to Open Telemetry alone?

Currently vanilla OpenTelemetry auto instrumentation is not able to fully support tracing within Kafka clients such as Kafka Streams, Kafka Connect, KSQL. **This means we are unable to capture full end-to-end trace of an event.**

What does work with vanilla OpenTelemetry auto instrumentation:

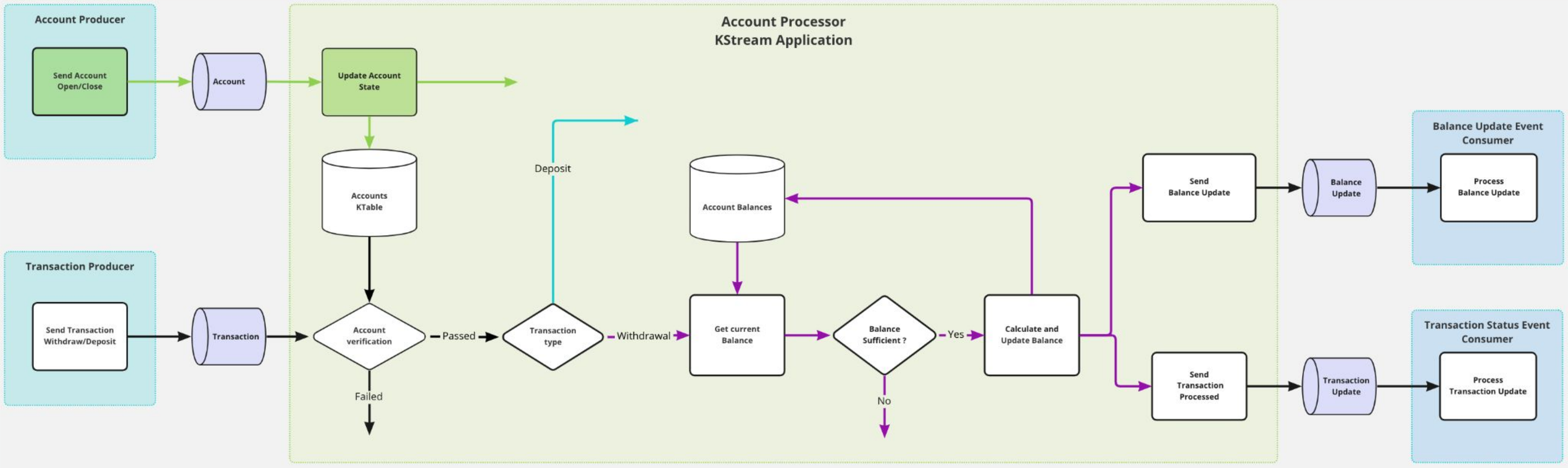
- Consumer / Producer tracing: applications developed using *kafka-clients* package.
- Stateless Kafka Streams tracing: applications developed using Kafka Streams, but not utilizing any stateful operations (i.e. no state stores are involved).

What the accelerator does to bridge this gap:

- Stateful Kafka Streams tracing
  - Context propagation: With vanilla implementation multiple disconnected traces are recorded instead of a single complete trace.
  - State store operation tracing: In vanilla implementation state store operations are not traced at all; e.g. state-store put, state-store get etc.
- Kafka Connect tracing
  - Context propagation: With vanilla instrumentation multiple disconnected traces are recorded instead of a single complete trace
  - SimpleMessageTransformation (SMT) tracing: SMTs are not traced at all.

Use case samples

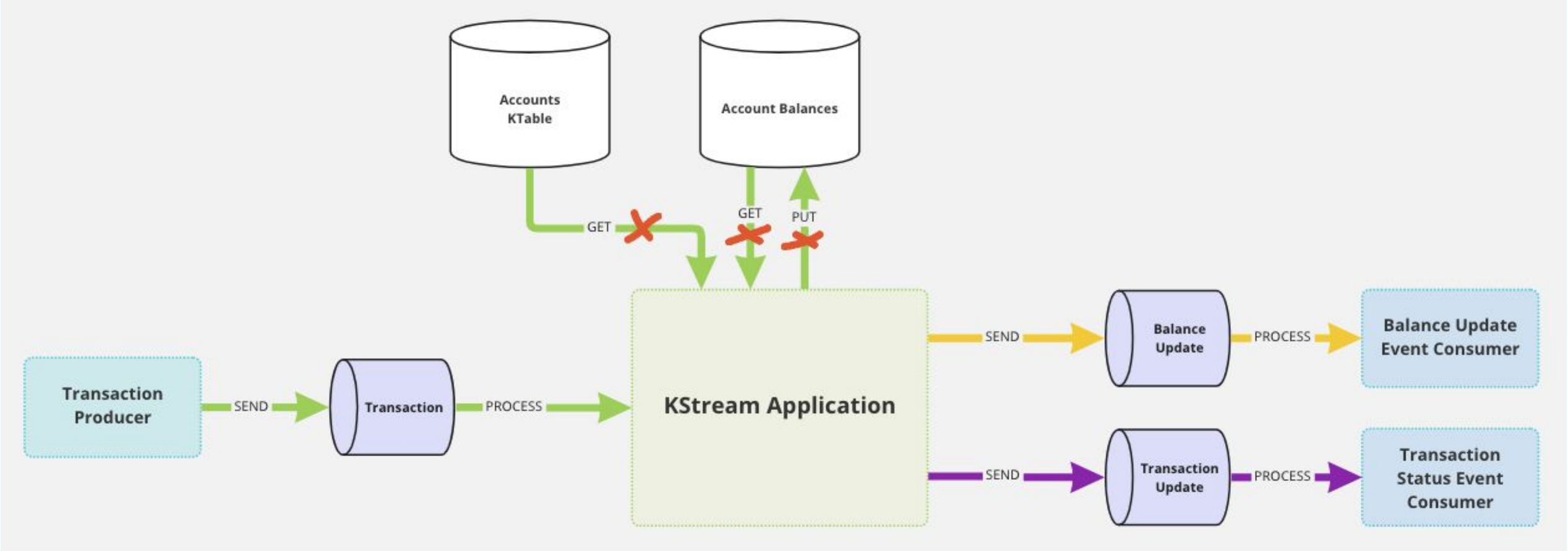
Use Case - origin tracking



This is an example end to end solution for a Kafka implementation including a Producer, Kafka Streams and Consumer.

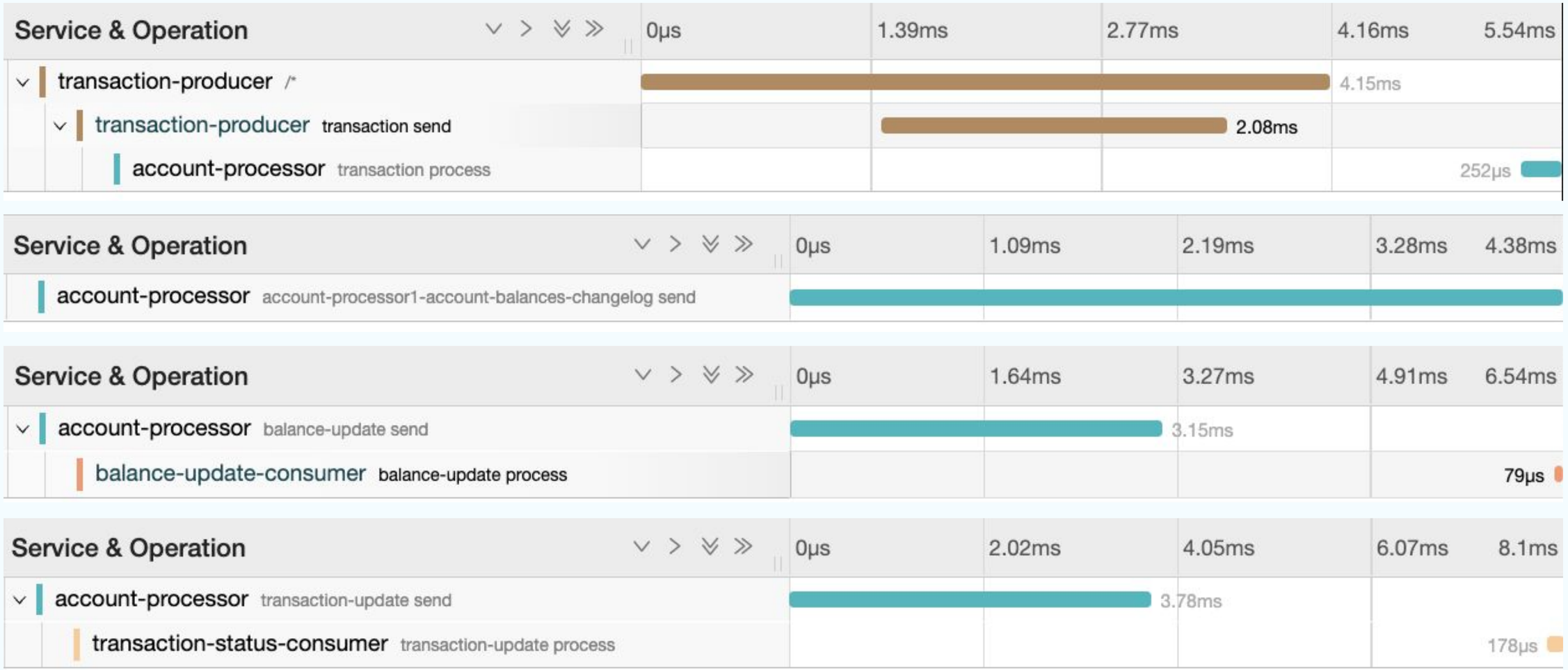
We use “*transaction withdrawal flow*” to demonstrate tracing with vanilla OpenTelemetry and the Event Lineage extension

Use Case - vanilla OpenTelemetry gaps



From the above diagram you can see that with vanilla OpenTelemetry we have three disconnected traces and state store operations are not traced at all.

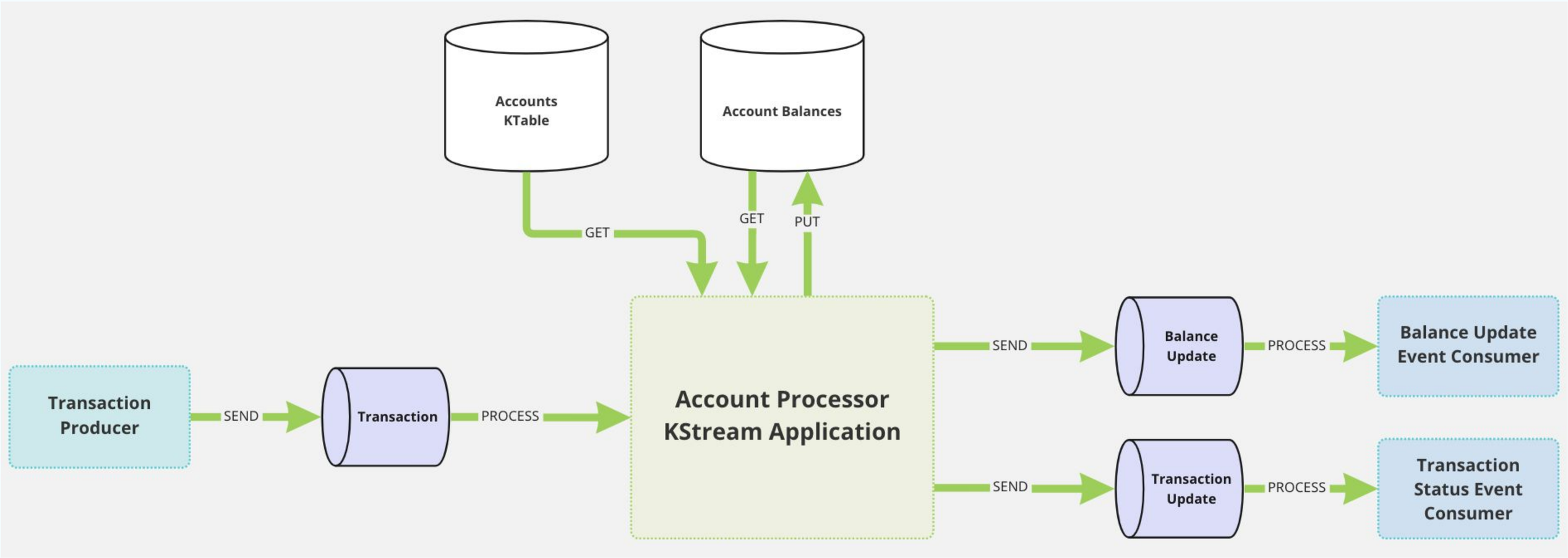
Use Case - vanilla OpenTelemetry gaps



If we visualise the trace data - we can see that we got one inbound and two outbound operation traces - that are disconnected.

For State Store operations - only change log send off is traced and even that operation's trace is disconnected from the rest.

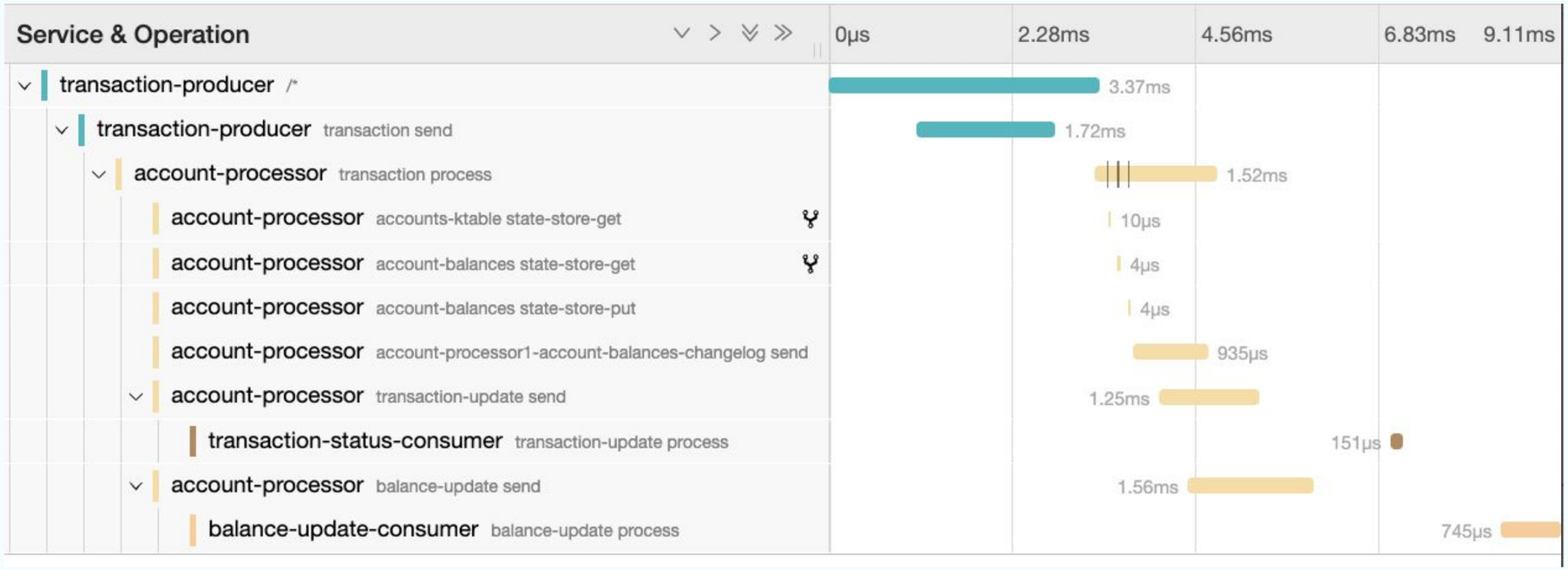
Use Case - with Event Lineage Extension



With the Event Lineage extension, the trace will now be shown as a single, complete, end-to-end flow trace which include:

- Inbound Flow
- State Store Operations
- All Outbound flows

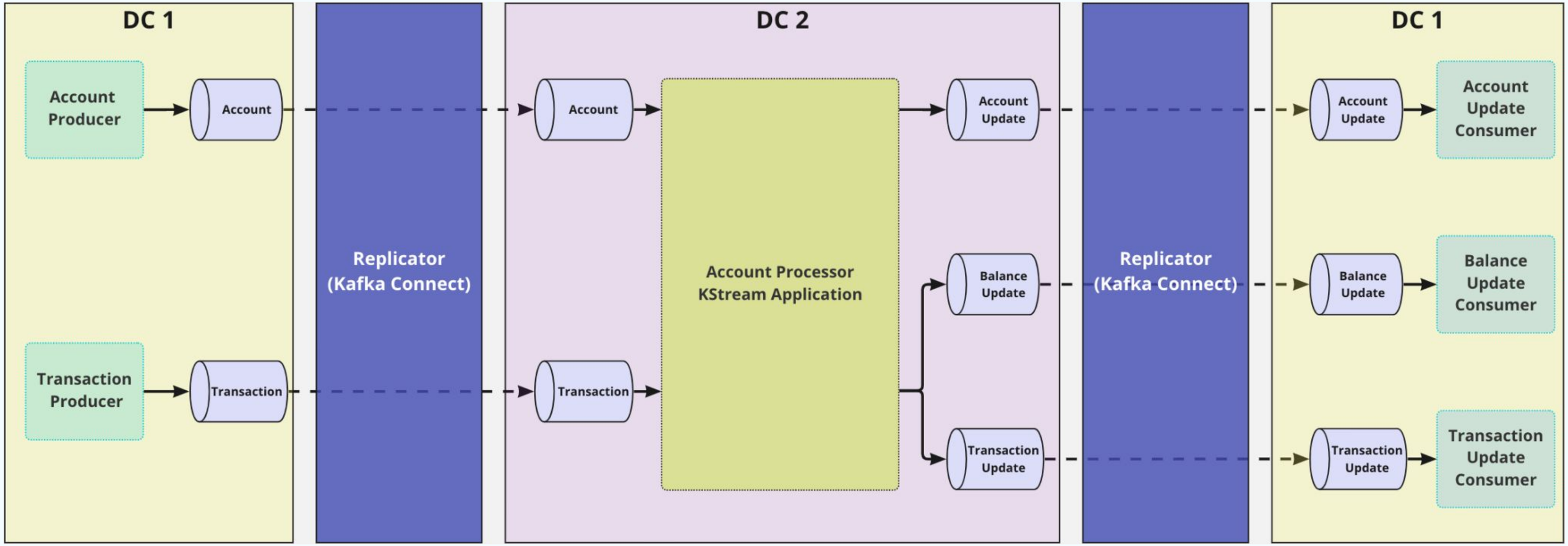
Kafka Streams: Demo trace with extension



If we visualise the trace data the following is now available:

- All the trace events are in a single connected trace
- Include all elements of the trace e.g. state store operations, inbound flow and outbound fanout flows

Use Case - Multi-Cluster



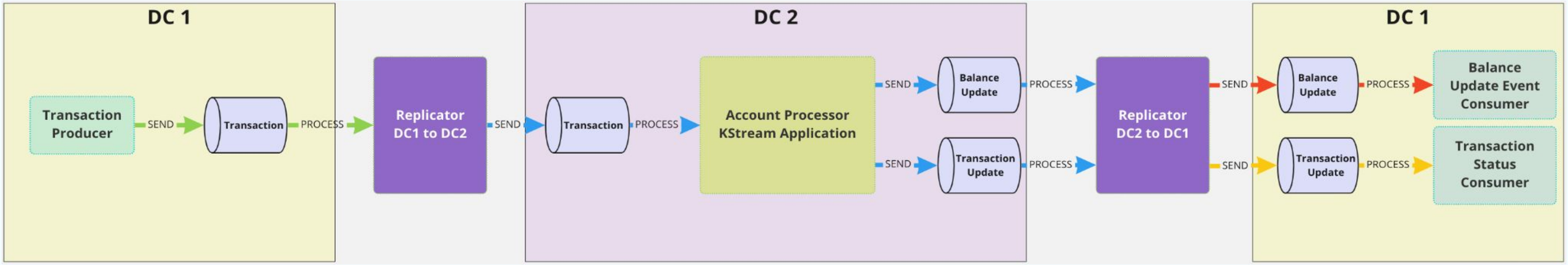
In this example, the original application has now been added to a multi-cluster environment.

The new flow now includes:

- Replication of inbound events from DC 1 to DC 2.
- Replication of outbound events from DC 2 to DC 1.

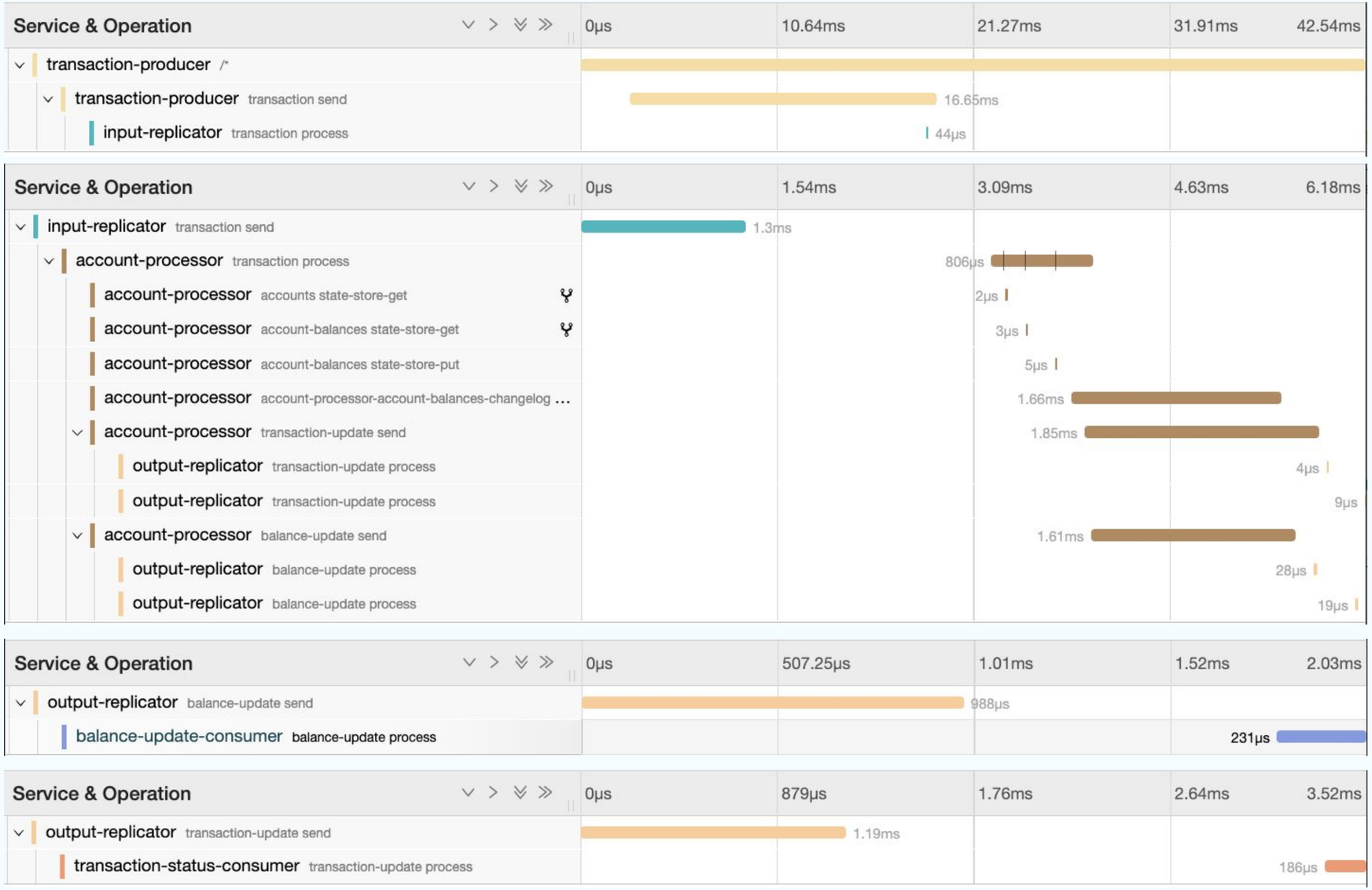
We are using Kafka Connect and Confluent Replicator for the replication.

Use Case - vanilla OpenTelemetry gaps



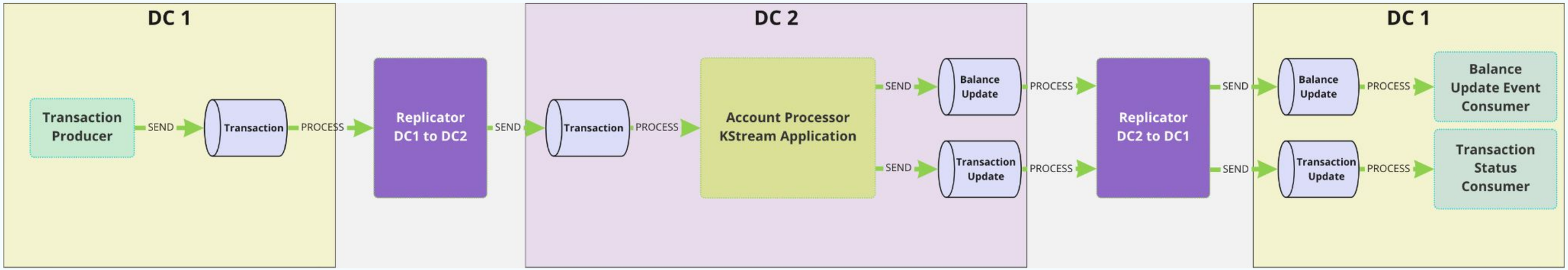
From the above diagram you can see that with vanilla OpenTelemetry we have four disconnected traces, depicted using different colours.

Use Case - vanilla OpenTelemetry gaps



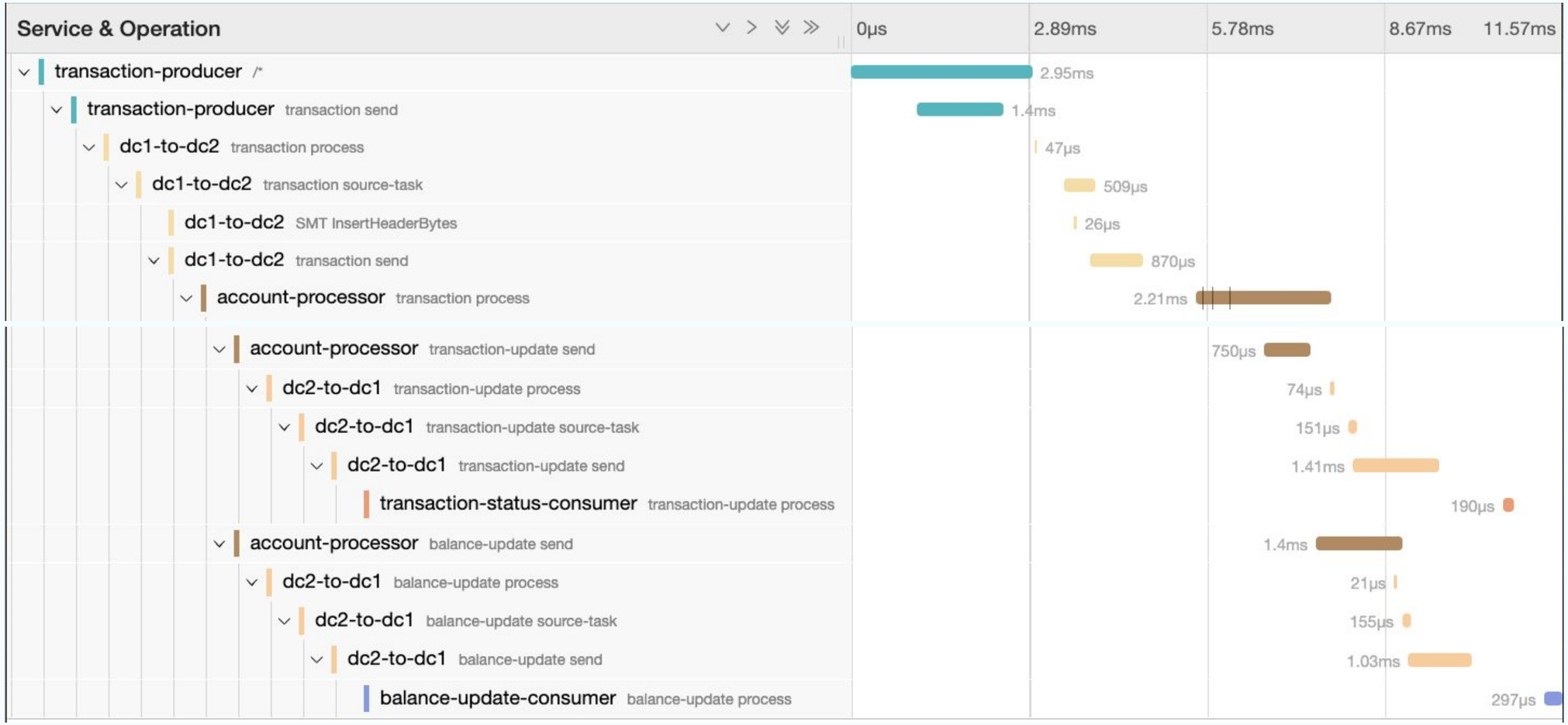
When visualising the trace data - we can see that inbound and outbound operation traces are disconnected in Connect/Replicator component - resulting in four separate traces instead of one end-to-end trace.

Use Case - vanilla OpenTelemetry gaps



From the above diagram you can see that with vanilla OpenTelemetry we have four disconnected traces, depicted using different colours.

Use Case - vanilla OpenTelemetry gaps



When visualising a single trace now includes:

- All the trace events within a single connected trace
- SimpleMessageTransform(SMT) operations are traced

CONFLUENT

Conclusion

Summary

In conclusion, achieving a complete end-to-end traceability in distributed system enables:

- **Event chain Audit** - record of: which messages were processed; where messages originated from; touch points of messages.
- **Root cause analysis** - ability to perform lookup of all messages involved in the process / event in question to determine source of bad data.
- **Data analysis** - flow and performance data analysis with contextual information granularity to individual message level.
- **Data visualization** - flow correlation, visualization of complex flows based on trace data captured.

In advance of considering deploying the accelerator, a customer must have:
An understanding of distributed tracing concepts, and an implementation suite which supports open telemetry.


CONFLUENT