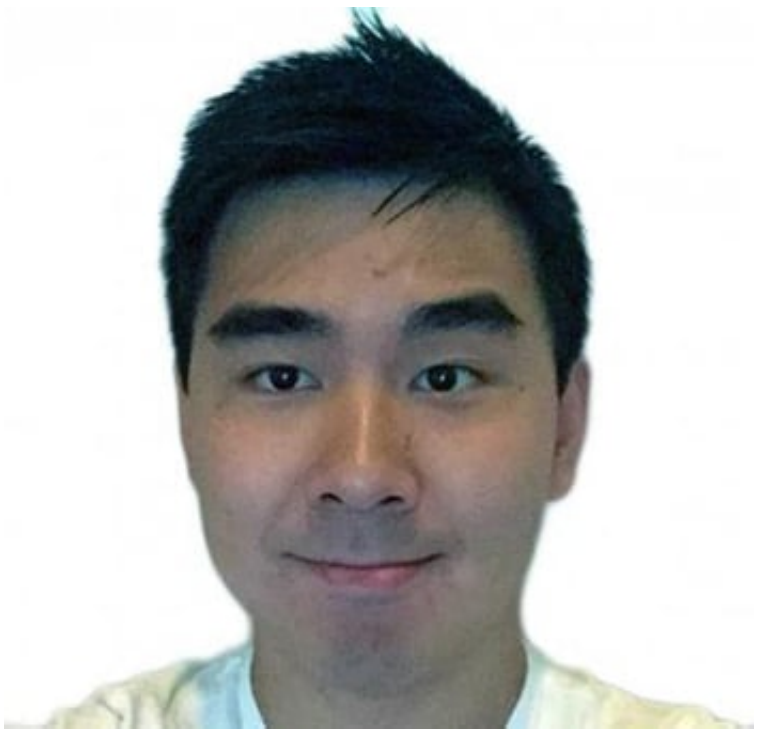
Exactly once is NOT exactly the same

October 13, 2017



Jerry Peng

Distributed event stream processing has become an increasingly hot topic in the area of Big Data. Notable Stream Processing Engines (SPEs) include Apache Storm, Apache Flink, Heron, Apache Kafka (Kafka Streams), and Apache Spark (Spark Streaming). One of the most notable and widely discussed features of SPEs is their processing semantics, with “exactly-once” being one of the most sought after and many SPEs claiming to provide “exactly-once” processing semantics.

There exists a lot of misunderstanding and ambiguity, however, surrounding what exactly “exactly-once” is, what it entails, and what it really means when individual SPEs claim to provide it. The label “exactly-once” for describing processing semantics is also very misleading. In this blog post, I’ll discuss how “exactly-once” processing semantics differ across many popular SPEs and why “exactly-once” can be better described as **effectively-once**. I’ll also explore the tradeoffs between common techniques used to achieve what is often called “exactly-once.”

Background

Stream processing, sometimes referred to as event processing, can be succinctly described as continuous processing of an unbounded series of data or events. A stream- or event-processing application can be more or less described as a directed graph and often, but not always, as a directed acyclic graph (DAG). In such a graph, each edge represents a flow of data or events and each vertex represents an operator that uses application-defined logic to process data or events from adjacent edges. There are two special types of vertices, commonly referenced as sources and sinks. Sources consume external data/events and inject them into the application while sinks typically gather results produced by the application. Figure 1 below depicts an example a streaming application.

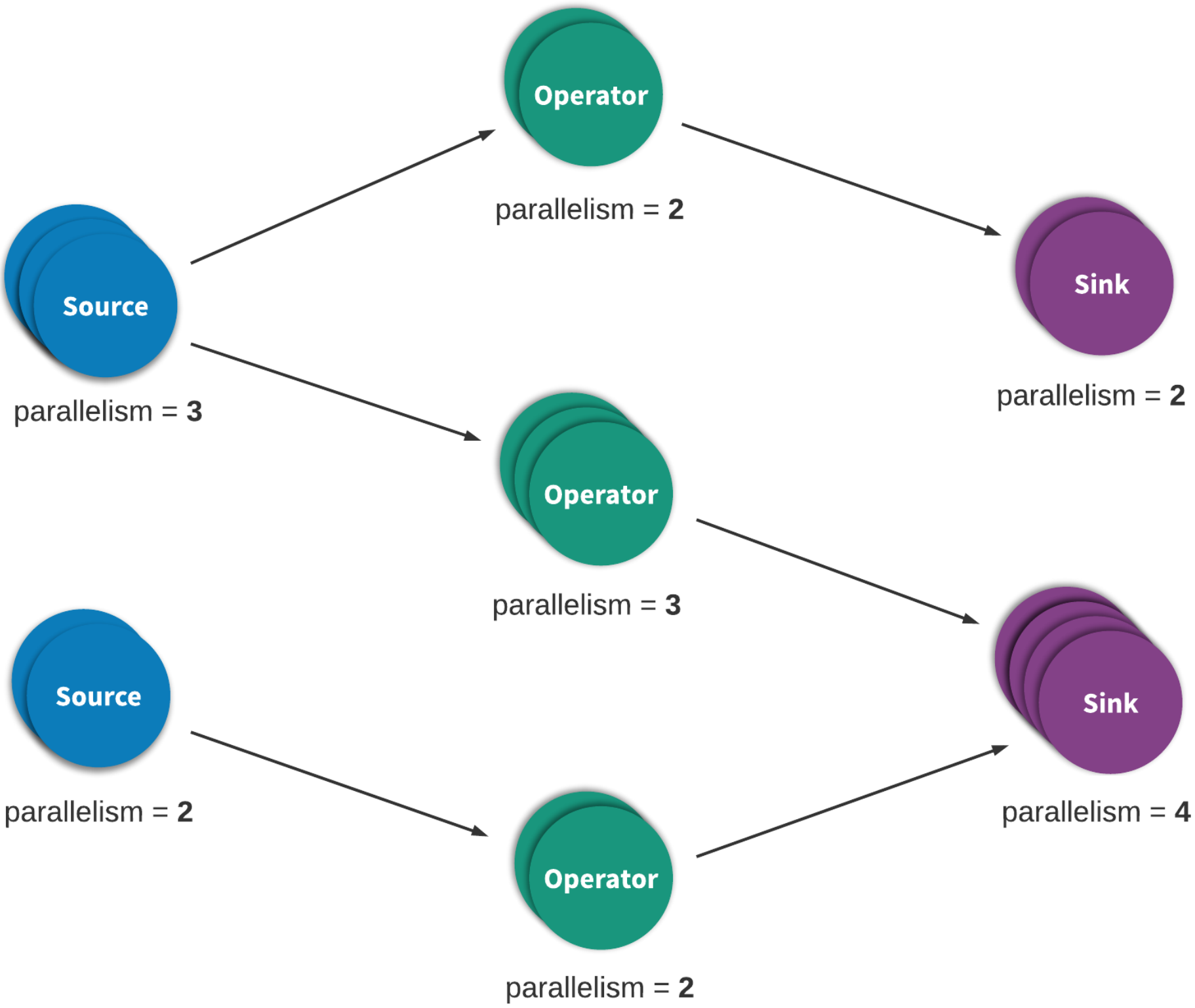


Figure 1. A typical Heron processing topology

An SPE that executes a stream/event processing application usually allows users to specify a reliability mode or processing semantics that indicates which guarantees it will provide for data processing across the entirety of the application graph. These guarantees are meaningful since you can always assume the possibility of failures via network, machines, etc. that can result in data loss. Three modes/labels, at-most-once, at-least-once, and exactly-once, are generally used to describe the data processing semantics that the SPE should provide to the application.

Here's a loose definition of those different processing semantics:

At-most-once

This is essentially a "best effort" approach. Data or events are guaranteed to be processed at most once by all operators in the application. This means that no additional attempts will be made to retry or retransmit events if it was lost before the streaming application can fully process it. Figure 2 illustrates an example of this.

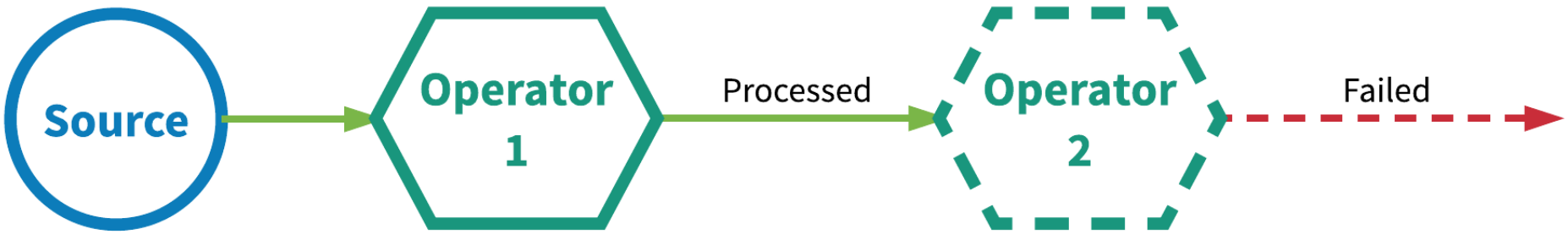


Figure 2. At-most-once processing semantics

At-least-once

Data or events are guaranteed to be processed at least once by all operators in the application graph. This usually means an event will be replayed or retransmitted from the source if the event is lost before the streaming application fully processed it. Since it can be retransmitted, however, an event can sometimes be processed more than once, thus the at-least-once term. Figure 3 illustrates an example of this. In this case, the first operator initially fails to process an event, then succeeds upon retry, then succeeds upon a second retry that turns out to have been unnecessary.

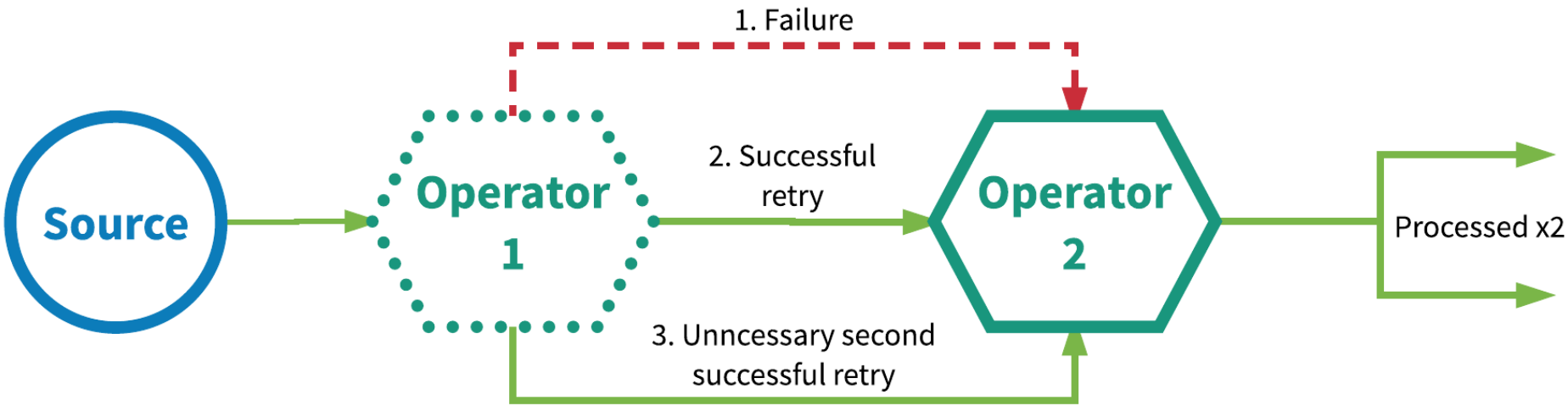


Figure 3. At-least-once processing semantics

Exactly-once

Events are guaranteed to be processed “exactly once” by all operators in the stream application, even in the event of various failures.

Two popular mechanisms are typically used to achieve “exactly-once” processing semantics.

1. Distributed snapshot/state checkpointing
2. At-least-once event delivery plus message deduplication

The distributed snapshot/state checkpointing method of achieving “exactly-once” is inspired by the Chandy-Lamport distributed snapshot algorithm.[1] With this mechanism, all the state for each operator in the streaming application is periodically checkpointed, and in the event of a failure anywhere in the system, all the state of for every operator is rolled back to the most recent globally consistent checkpoint. During the rollback, all processing will be paused. Sources are also reset to the correct offset corresponding to the most recent checkpoint. The whole streaming application is basically rewound to its most recent consistent state and processing can then restart from that state. Figure 4 below illustrates the basics of this mechanism.

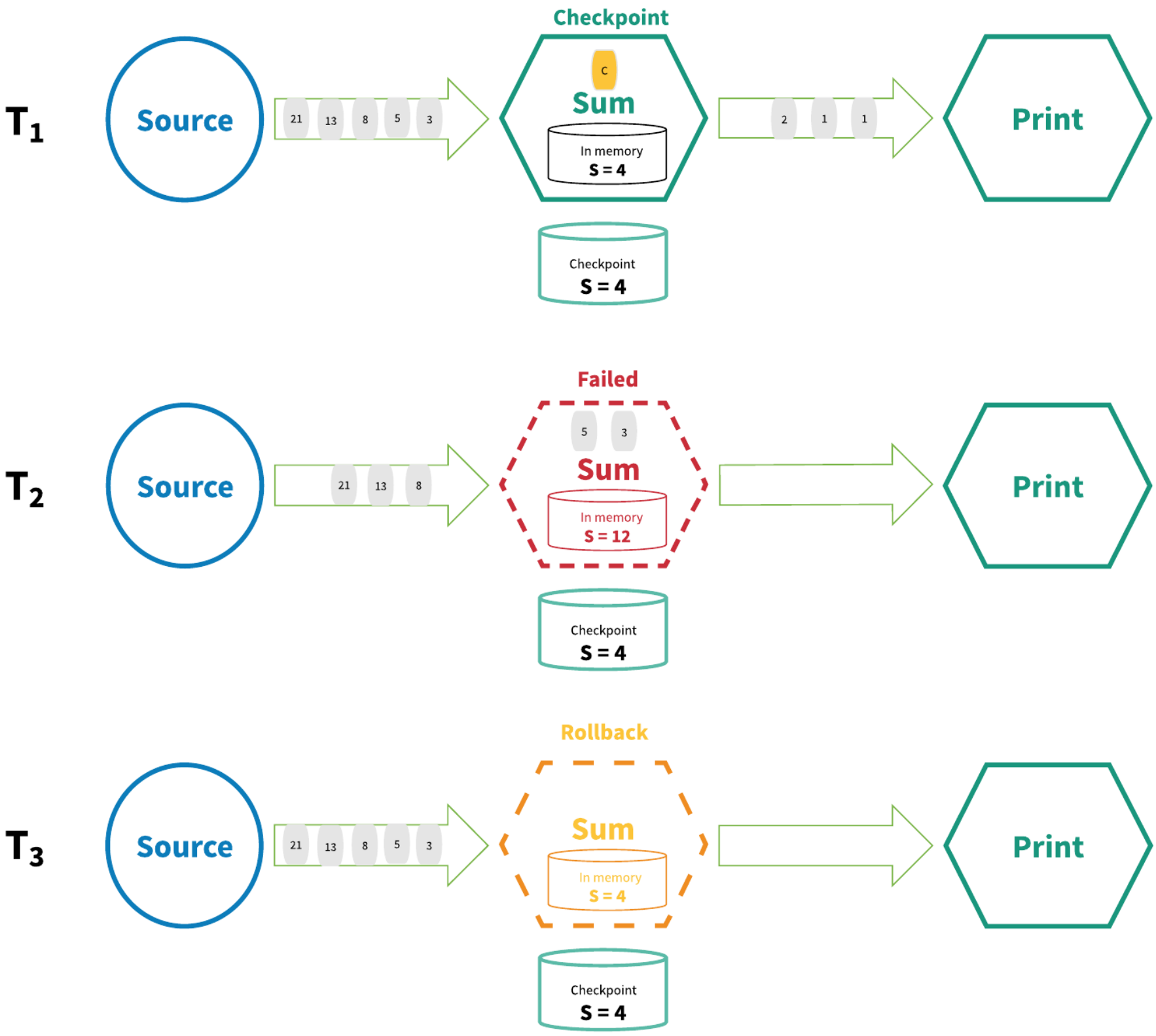


Figure 4. Distributed snapshot

In Figure 4, the streaming application is working normally at T1 and the state is checkpointed. At time T_2, however, the operator fails to process an incoming datum. At this point, the state value of S = 4 has been saved to durable storage, while the state value S = 12 is held in the operator's memory. In order to overcome this discrepancy, at time T_3 the processing graph rewinds the state to S = 4 and "replays" each successive state in the stream up to the most recent, processing each datum. The end result is that some data have been processed multiple times, but that's okay because the resulting state is the same no matter how many rollbacks have been performed.

Another method used to achieve "exactly-once" is through implementing at-least-once event delivery in conjunction with event deduplication on a per-operator basis. SPEs utilizing this approach will replay failed events for further attempts at processing and remove duplicated events for every operator prior to the events entering the user defined logic in the operator. This mechanism requires that a transaction log be maintained for every operator to track which events it has already processed. SPEs that utilize a mechanism like such are Google's MillWheel[2] and Apache Kafka Streams. Figure 5 illustrates the gist of this mechanism.

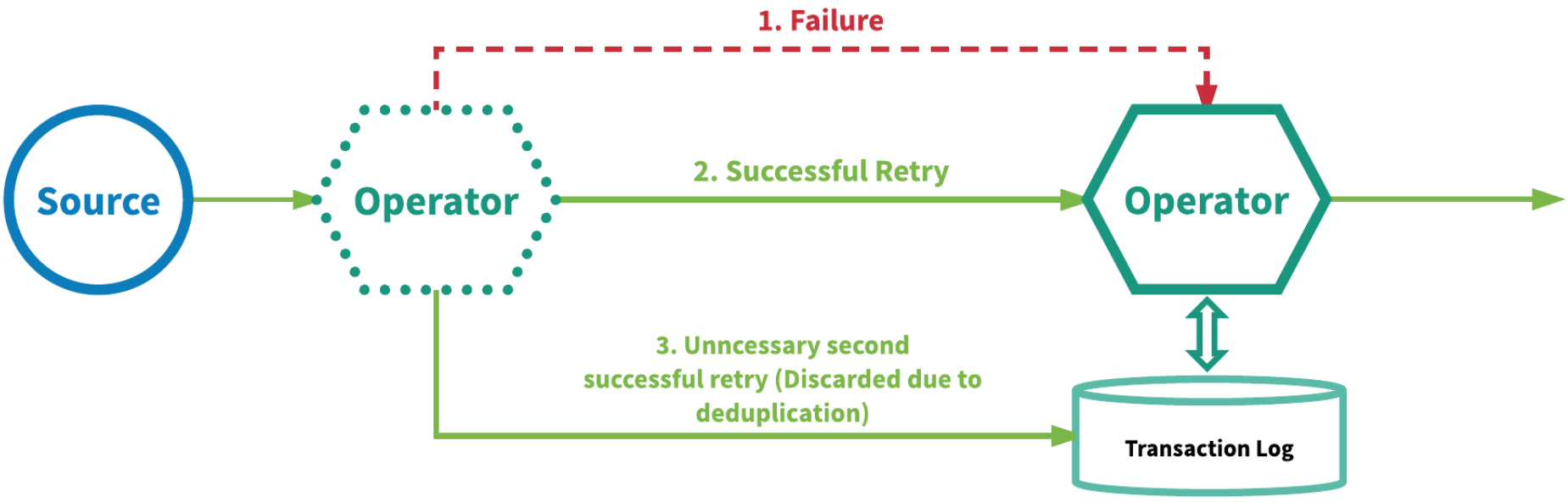


Figure 5. At-least-once delivery plus deduplication

Is exactly-once really exactly-once?

Now let's reexamine what the "exactly-once" processing semantics really guarantees to the end user. The label "exactly-once" is misleading in describing what is done exactly once.

Some might think that "exactly-once" describes the guarantee to event processing in which each event in the stream is processed only once. In reality, there is no SPE that can guarantee exactly-once processing. To guarantee that the user-defined logic in each operator only executes once per event is impossible in the face of arbitrary failures, because partial execution of user code is an ever-present possibility.

Consider the scenario where you have a stream processing operator that performs a map operation that prints the ID of an incoming event and then returns the event unchanged. The pseudocode below illustrates this operation:

```
Map (Event event) {
    Print "Event ID: " + event.getId()
    Return event
}
```

Each event has a GUID (Global Unique ID). If exactly-once execution of the user logic can be guaranteed then the event ID will be only printed out once. However, this can never be guaranteed, as failures can happen at any time and at any point in the execution of the user-defined logic. The SPE cannot determine on its own the point up to which the user-defined processing logic has been executed. Thus, arbitrary user-defined logic is not guaranteed to be executed only once. This also implies that external operations, such as database writes, implemented in user defined logic is also not guaranteed to executed exactly once. Such operations would still be need to be implemented in an idempotent fashion.

So what does SPEs guarantee when they claim "exactly-once" processing semantics? If user logic cannot be guaranteed to be executed exactly once then what is executed exactly once? When SPEs claim "exactly-once" processing semantics, what they're actually saying is that they can guarantee that updates to state managed by the SPE are committed only once to a durable backend store.

Both mechanisms described above use a durable backend store as a source of truth that can hold the state of every operator and automatically commit updates to it. For mechanism 1 (distributed snapshot/state checkpointing), this durable backend state is used to hold the globally consistent state checkpoints (checkpointed state for every operator) for the streaming application. For mechanism 2 (at-least-once event delivery plus deduplication), the durable backend state is used to store the state of every operator as well as a transaction log for every operator that tracks all the events it has already fully processed.

The committing of state or applying updates to the durable backend that is the source of truth can be described as occurring exactly-once. Computing the state update/change, i.e. processing the event that is executing arbitrary user -defined logic on the event, however, can happen more than once if failures occur, as mentioned above. In other words, the processing of an event can happen more than once but

the effect of that processing is only reflected once in the durable backend state store. Here at Streamlio, we've decided that **effectively-once** is the best term for describing these processing semantics.

Distributed snapshot versus at-least-once event delivery plus deduplication

From a semantic point of view, both the distributed snapshot and at-least-once event delivery plus deduplication mechanisms provide that same guarantee. Due to differences in implementation between the two mechanisms, however, there are significant performance differences.

The performance overhead of mechanism 1 (distributed snapshot/state checkpointing) on top of the SPE can be minimal since the SPE is essentially sending a few special events alongside regular events through all the operators in the streaming application, while state checkpointing can be performed asynchronously in the background. For large streaming applications, however, failures may happen more frequently, causing the SPE to need to pause the application and roll back the state of all operators, which will in turn impact performance. The larger the streaming application, the more likely and thus more frequently failures can occur, and in turn, the more significantly the performance of the streaming application will be impacted. However, again, this mechanism is very non-intrusive and demands minimal additional resources impact to run.

Mechanism 2 (at-least-once event delivery plus deduplication) may require a lot more resources, especially storage. With this mechanism, the SPE would need to be able to track every tuple that has been fully processed by every instance of an operator to perform deduplication as well as perform the deduplication itself for every event. This can amount to a huge amount of data to keep track of, especially if the streaming application is large or if there are many applications running. There is also performance overhead associated with every event at every operator to perform the deduplication. With this mechanism, however, the performance of the streaming application is less likely to be impacted by the size of the application. With mechanism 1, a global pause and state rollback needs to occur if any failures occur on any operator; with mechanism 2, the effects of a failure are much more localized. When a failure occurs in an operator, events that might have not been fully processed are just replayed/retransmitted from an upstream source. The performance impact is isolated to where the failure happened in the streaming application and will cause little impact to the performance of other operators in the streaming application. The pros and cons of both mechanisms from a performance standpoint are listed in the tables below.

Distributed snapshot/state checkpointing

Pros	Cons
Little performance and resource overhead	Larger impact to performance when recovering from failures
	Potential impact to performance increases as topology gets larger

At-least-once delivery plus deduplication

Pros	Cons
Performance impact of failures are localized	Potentially need large amounts of storage and infrastructure to support
Impact of failures does not necessarily increase with the size of the topology	Performance overhead for every event at every operator

Though there are differences between the distributed snapshot and at-least-once event delivery plus deduplication mechanisms from a theoretical point of view, both can be reduced to at-least-once processing plus idempotency. For both mechanisms, events will be replayed/retransmitted when failures occur (implementing at-least-once), and through state rollback or event deduplication, operators essentially become idempotent when updating internally managed state.

Conclusion

In this blog post, I hope to have convinced you that the term "exactly-once" is very misleading. Providing "exactly-once" processing semantics really means that distinct updates to the state of an operator that is managed by the stream processing engine are only reflected once. "Exactly-once" by no means guarantees that processing of an event, i.e. execution of arbitrary user-defined logic, will happen only once. Here at Streamlio, we prefer the term effectively once for this guarantee because processing is not necessarily guaranteed to occur once but the effect on the SPE-managed state is reflected once. Two popular mechanisms, distributed snapshot and dessage deduplication, are used to implement exactly/effectively-once processing semantics. Both mechanisms provide the same semantic guarantees to message processing and state updates but there are nonetheless differences in performance. This post is not meant to convince you that either mechanism is superior to the other, as each has its pros and cons.

References

1. Chandy, K. Mani and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3.1 (1985): 63-75.