Flink (//ci.apache.org/projects/flink/flink-docs-release-1.3) v1.3

Search Docs

Go

Pick Docs Version

This documentation is for an out-of-date version of Apache Flink. We recommend you use the latest stable version (https://ci.apache.org/projects/flink/flink-docs-stable/).

Data Streaming Fault Tolerance

This document describes Flink's fault tolerance mechanism for streaming data flows.

Introduction

Apache Flink offers a fault tolerance mechanism to consistently recover the state of data streaming applications. The mechanism ensures that even in the presence of failures, the program's state will eventually reflect every record from the data stream **exactly once**. Note that there is a switch to **downgrade** the guarantees to **at least once** (described below).

The fault tolerance mechanism continuously draws snapshots of the distributed streaming data flow. For streaming applications with small state, these snapshots are very light-weight and can be drawn frequently without much impact on performance. The state of the streaming applications is stored at a configurable place (such as the master node, or HDFS).

In case of a program failure (due to machine-, network-, or software failure), Flink stops the distributed streaming dataflow. The system then restarts the operators and resets them to the latest successful checkpoint. The input streams are reset to the point of the state snapshot. Any records that are processed as part of the restarted parallel dataflow are guaranteed to not have been part of the previously checkpointed state.

Note: By default, checkpointing is disabled. See Checkpointing (//ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/checkpointing.html) for details on how to enable and configure checkpointing.

Note: For this mechanism to realize its full guarantees, the data stream source (such as message queue or broker) needs to be able to rewind the stream to a defined recent point. Apache Kafka (http://kafka.apache.org) has this ability and Flink's connector to Kafka exploits this ability. See Fault Tolerance Guarantees of Data Sources and

Sinks (//ci.apache.org/projects/flink/flink-docs-release-1.3/dev/connectors/guarantees.html) for more information about the guarantees provided by Flink's connectors.

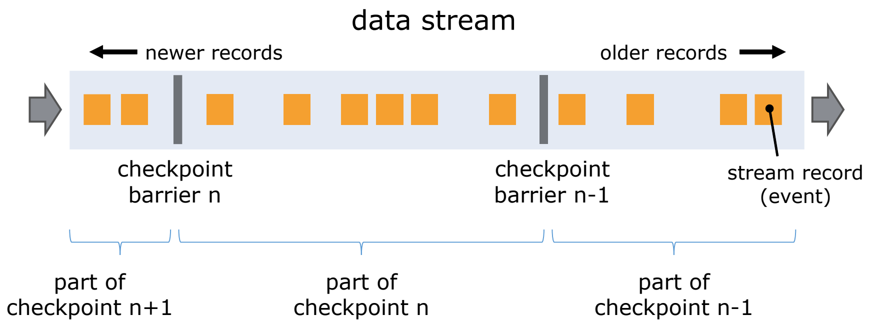
Note: Because Flink's checkpoints are realized through distributed snapshots, we use the words **snapshot** and **checkpoint** interchangeably.

Checkpointing

The central part of Flink's fault tolerance mechanism is drawing consistent snapshots of the distributed data stream and operator state. These snapshots act as consistent checkpoints to which the system can fall back in case of a failure. Flink's mechanism for drawing these snapshots is described in "Lightweight Asynchronous Snapshots for Distributed Dataflows (http://arxiv.org/abs/1506.08603)". It is inspired by the standard Chandy-Lamport algorithm (http://research.microsoft.com/en-us/um/people/lamport/pubs/chandy.pdf) for distributed snapshots and is specifically tailored to Flink's execution model.

Barriers

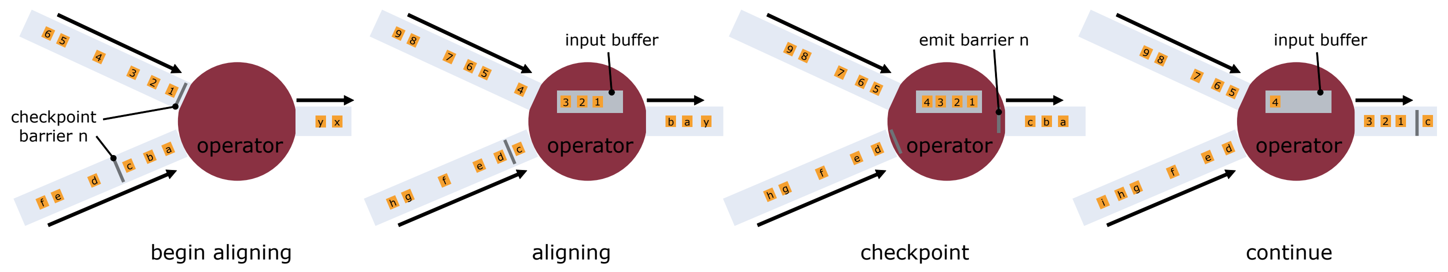
A core element in Flink's distributed snapshotting are the **stream barriers**. These barriers are injected into the data stream and flow with the records as part of the data stream. Barriers never overtake records, the flow strictly in line. A barrier separates the records in the data stream into the set of records that goes into the current snapshot, and the records that go into the next snapshot. Each barrier carries the ID of the snapshot whose records it pushed in front of it. Barriers do not interrupt the flow of the stream and are hence very lightweight. Multiple barriers from different snapshots can be in the stream at the same time, which means that various snapshots may happen concurrently.



Stream barriers are injected into the parallel data flow at the stream sources. The point where the barriers for snapshot **n** are injected (let's call it S_n) is the position in the source stream up to which the snapshot covers the data. For example, in Apache Kafka, this position would be the last record's offset in the partition. This position S_n is reported to the **checkpoint coordinator** (Flink's JobManager).

The barriers then flow downstream. When an intermediate operator has received a barrier for snapshot **n** from all of its input streams, it emits a barrier for snapshot **n** into all of its outgoing streams. Once a sink operator (the end of a streaming DAG) has received the barrier **n** from all of its input streams, it acknowledges that snapshot **n** to the checkpoint coordinator. After all sinks have acknowledged a snapshot, it is considered completed.

Once snapshot **n** has been completed, the job will never again ask the source for records from before S_n, since at that point these records (and their descendant records) will have passed through the entire data flow topology.



Operators that receive more than one input stream need to **align** the input streams on the snapshot barriers. The figure above illustrates this:

- As soon as the operator receives snapshot barrier **n** from an incoming stream, it cannot process any further records from that stream until it has received the barrier **n** from the other inputs as well. Otherwise, it would mix records that belong to snapshot **n** and with records that belong to snapshot **n+1**.
- Streams that report barrier **n** are temporarily set aside. Records that are received from these streams are not processed, but put into an input buffer.
- Once the last stream has received barrier **n**, the operator emits all pending outgoing records, and then emits snapshot **n** barriers itself.
- After that, it resumes processing records from all input streams, processing records from the input buffers before processing the records from the streams.

State

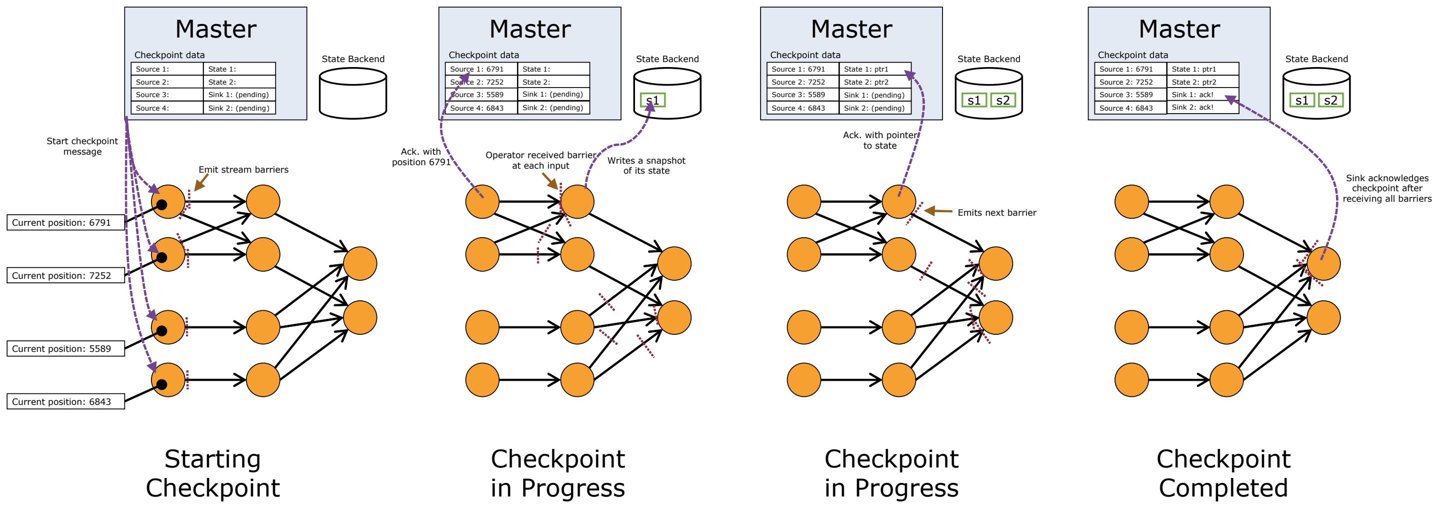
When operators contain any form of **state**, this state must be part of the snapshots as well. Operator state comes in different forms:

- **User-defined state**: This is state that is created and modified directly by the transformation functions (like `map()` or `filter()`). See State in Streaming Applications (//ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/state.html) for details.
- **System state**: This state refers to data buffers that are part of the operator's computation. A typical example for this state are the **window buffers**, inside which the system collects (and aggregates) records for windows until the window is evaluated and evicted.

Operators snapshot their state at the point in time when they have received all snapshot barriers from their input streams, and before emitting the barriers to their output streams. At that point, all updates to the state from records before the barriers will have been made, and no updates that depend on records from after the barriers have been applied. Because the state of a snapshot may be large, it is stored in a configurable **state backend (//ci.apache.org/projects/flink/flink-docs-release-1.3/ops/state_backends.html)**. By default, this is the JobManager's memory, but for production use a distributed reliable storage should be configured (such as HDFS). After the state has been stored, the operator acknowledges the checkpoint, emits the snapshot barrier into the output streams, and proceeds.

The resulting snapshot now contains:

- For each parallel stream data source, the offset/position in the stream when the snapshot was started
- For each operator, a pointer to the state that was stored as part of the snapshot



Exactly Once vs. At Least Once

The alignment step may add latency to the streaming program. Usually, this extra latency is on the order of a few milliseconds, but we have seen cases where the latency of some outliers increased noticeably. For applications that require consistently super low latencies (few milliseconds) for all records, Flink has a switch to skip the stream alignment during a checkpoint. Checkpoint snapshots are still drawn as soon as an operator has seen the checkpoint barrier from each input.

When the alignment is skipped, an operator keeps processing all inputs, even after some checkpoint barriers for checkpoint **n** arrived. That way, the operator also processes elements that belong to checkpoint **n+1** before the state snapshot for checkpoint **n** was taken. On a restore, these records will occur as duplicates, because they are both included in the state snapshot of checkpoint **n**, and will be replayed as part of the data after checkpoint **n**.

NOTE: Alignment happens only for operators with multiple predecessors (joins) as well as operators with multiple senders (after a stream repartitioning/shuffle). Because of that, dataflows with only embarrassingly parallel streaming operations (`map()`, `flatMap()`, `filter()`, ...) actually give **exactly once** guarantees even in **at least once** mode.

Asynchronous State Snapshots

Note that the above described mechanism implies that operators stop processing input records while they are storing a snapshot of their state in the **state backend**. This **synchronous** state snapshot introduces a delay every time a snapshot is taken.

It is possible to let an operator continue processing while it stores its state snapshot, effectively letting the state snapshots happen **asynchronously** in the background. To do that, the operator must be able to produce a state object that should be stored in a way such that further modifications to the operator state do not affect that state object. For example, **copy-on-write** data structures, such as are used in RocksDB, have this behavior.

After receiving the checkpoint barriers on its inputs, the operator starts the asynchronous snapshot copying of its state. It immediately emits the barrier to its outputs and continues with the regular stream processing. Once the background copy process has completed, it acknowledges the checkpoint to the checkpoint coordinator (the JobManager). The checkpoint is now only complete after all sinks have received the barriers and all stateful operators have acknowledged their completed backup (which may be after the barriers reach the sinks).

See State Backends (//ci.apache.org/projects/flink/flink-docs-release-1.3/ops/state_backends.html) for details on the state snapshots.

Recovery

Recovery under this mechanism is straightforward: Upon a failure, Flink selects the latest completed checkpoint **k**. The system then re-deploys the entire distributed dataflow, and gives each operator the state that was snapshotted as part of checkpoint **k**. The sources are set to start reading the stream from position $\mathbf{S_k}$. For example in Apache Kafka, that means telling the consumer to start fetching from offset $\mathbf{S_k}$.

If state was snapshotted incrementally, the operators start with the state of the latest full snapshot and then apply a series of incremental snapshot updates to that state.

See Restart Strategies (//ci.apache.org/projects/flink/flink-docs-release-1.3/dev/restart_strategies.html) for more information.

Operator Snapshot Implementation

When operator snapshots are taken, there are two parts: the **synchronous** and the **asynchronous** parts.

Operators and state backends provide their snapshots as a Java `FutureTask`. That task contains the state where the **synchronous** part is completed and the **asynchronous** part is pending. The asynchronous part is then executed by a background thread for that checkpoint.

Operators that checkpoint purely synchronously return an already completed `FutureTask`. If an asynchronous operation needs to be performed, it is executed in the `run()` method of that `FutureTask`.

The tasks are cancelable, so that streams and other resource consuming handles can be released.