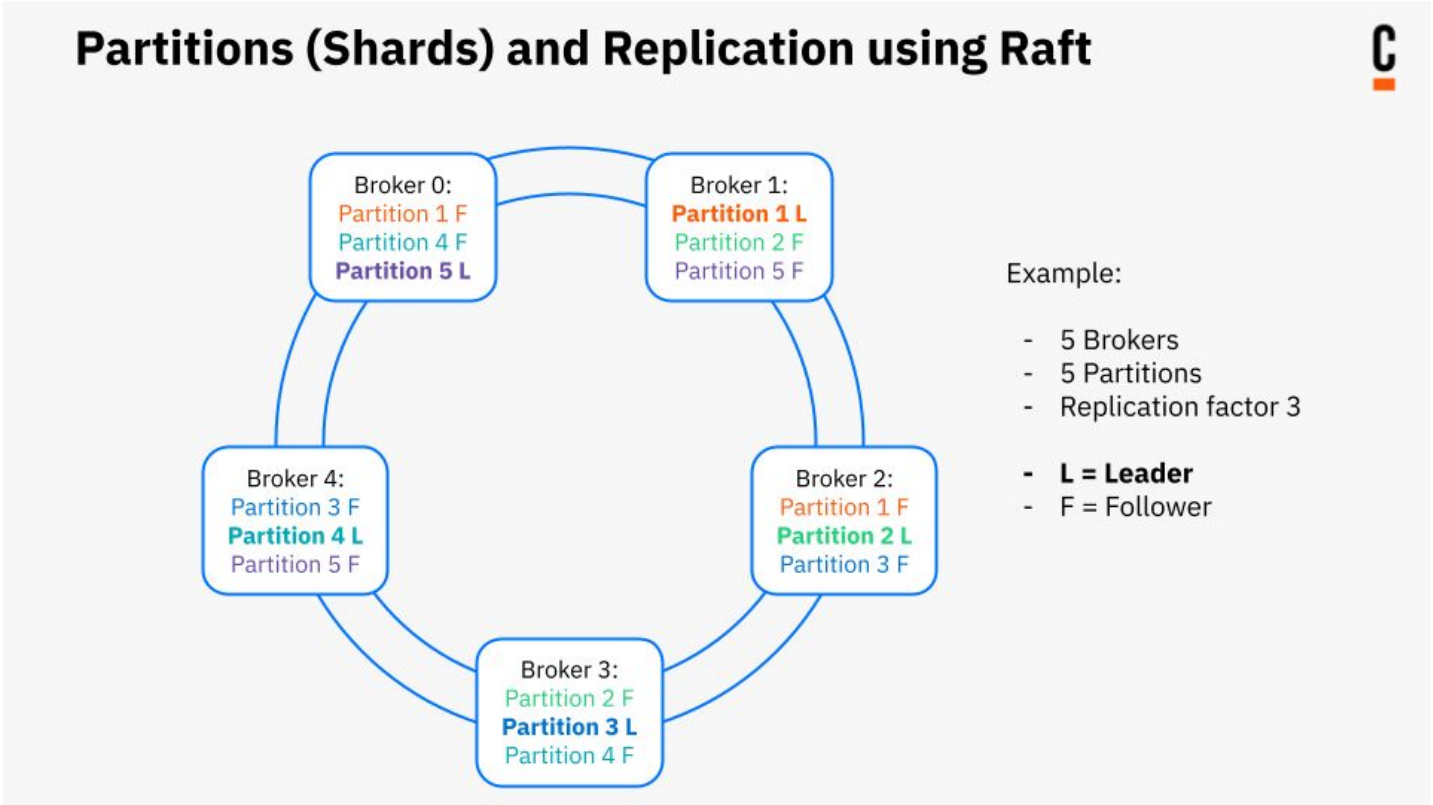
Camunda 8 clustering

Partitions and Replication using Raft



- Each partition has 1 leader + 2 followers (replication factor = 3).
- **Leaders are distributed** across brokers for **load balancing**.
- **Followers replicate the leader's state** to ensure **fault tolerance**.
- If any broker/server goes down, the remaining brokers can promote a follower to leader for the affected partitions.

Each broker usually runs on its **own machine** / **VM** / **Kubernetes pod**.

This is how you achieve fault tolerance:

- If one server fails, the other brokers (on other servers) still hold replicas of the partitions.
- Leaders can fail over to followers on different machines.

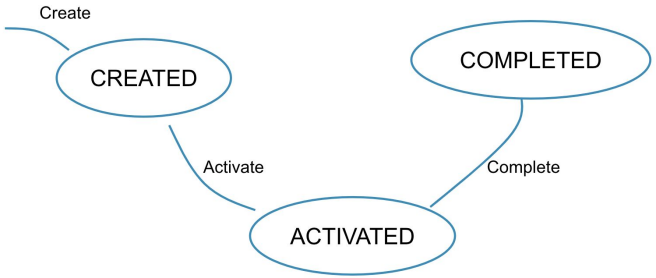
State Machines

Zeebe manages stateful entities like jobs and processes. Internally, these entities are implemented as state machines managed by a stream processor.

From **each state**, a set of **transitions** defines the next possible states. Transitioning into a new state may produce outputs/side effects.

Let's look at the state machine for jobs (CREATED, ACTIVATED, COMPLETED):

https://docs.camunda.io/docs/components/zeebe/technical-concepts/internal-processing/



Stateful Stream Processing

A stream processor reads the record stream sequentially and interprets the commands with respect to the addressed entity's lifecycle. More specifically, a stream processor repeatedly performs the following steps:

- Consume the next command from the stream.
- Determine if the command is applicable based on the state lifecycle and the entity's current state.
- If the command is applicable, apply it to the state machine. If the command was sent by a client, send a reply/response.
- If the command is not applicable, reject it. If it was sent by a client, send an error reply/response.
- Publish an event reporting the entity's new state.

For example, processing the **Create Job command** produces the **event Job Created**.

https://docs.camunda.io/docs/components/zeebe/technical-concepts/internal-processing/#events-and-commands

Why both Log Stream & RocksDB

Log Stream:

- Every change in Zeebe (process instance started, variable updated, job completed, etc.) is first written as an event to the log stream.
- The log stream is append-only (immutable history of events).
- It is replicated across the partition's leader and followers to guarantee durability and consistency.
- On replay, you can rebuild the broker's state from scratch by consuming the log stream events.

Why both Log Stream & RocksDB

RocksDB:

- RocksDB holds the **materialized** state derived from the log stream.
- When an event is applied (e.g., “Job X completed”), Zeebe updates RocksDB to reflect the latest state. For example:
  - Active process instances
  - Current variables for a scope
  - Pending timers
  - Open message subscriptions
- RocksDB is updated in lockstep with log stream commits (to ensure consistency).

Why both?

- **Log stream** gives durability + replayability (you never lose the history).
- **RocksDB** gives fast query and state access (so you don’t need to scan the log every time you want to know “what’s the current variable value?”).

MATERIALIZED Explained

Suppose you have a sales table **with 100 million rows**.

You **frequently need a report: “Total sales per region for the current year.”**

```
SELECT region, SUM(amount)
FROM sales
WHERE year = 2025
GROUP BY region;
```

Running this query every time would be slow.

MATERIALIZED Explained

CREATE **MATERIALIZED** VIEW sales_summary_2025 AS

SELECT region, SUM(amount)

FROM sales

WHERE year = 2025

GROUP BY region;

Now, whenever someone queries sales_summary_2025, it's just reading **precomputed** results.

The materialized view can be refreshed periodically to stay up-to-date.

Log Stream = **source of truth** (append-only event log).

RocksDB = **current state materialization** (key–value store that holds the latest state of the workflow execution).