CronFlow : Distributed Job Scheduling System

Ahmed Rakan

Abstract

CronFlow is a highly available, strongly consistent, and fault-tolerant distributed job scheduling system built on a leader-follower architecture. It leverages centralized in-memory coordination (e.g., Redis) for fast decision-making while ensuring durable persistence of job logs, state, and history for reliability.

The system follows a leader-follower model with automatic failover, allowing any node to assume leadership if and only if no active leader exists. Once elected, the leader must periodically confirm its liveness through heartbeats, ensuring continuous operation. Strong consistency is maintained via distributed locks with fencing tokens and quorum-based writes, preventing split-brain scenarios and data corruption.

For maximum fault tolerance, CronFlow incorporates self-healing workers, automatic job retries, and state reconciliation, enabling graceful degradation under failures. Workers operate in a stateless manner, with dynamic load balancing managed by the elected leader (which can be any node in the cluster). All components—workers, leaders, and the API server—emit real-time events, enabling robust logging, proactive alerting, and comprehensive monitoring for operational visibility.

1. Architecture Overview

CronFlow consists of five core components, each serving a distinct purpose while working together to fulfill the system's objectives. **Leader Node** : The leader is responsible for coordinating job scheduling, managing distributed locks, and handling automatic failover. If the leader fails, any worker node can elect itself as the new leader, ensuring uninterrupted operation. The leadership contract is acquired by the first starting worker node at startup and maintained via leadership heartbeat with configurable TTL. **Worker Nodes**: Workers execute scheduled jobs, report health status, and participate in leader elections. They are stateless, allowing for dynamic scaling and load distribution. All nodes report their state at all times, in-case the leader node fails, leader election happens, in-case worker node fails load distribution happens. The system needs at least one worker node, two leader nodes for high availability. The system cannot operate with a single node as it defeats the main purpose of designing such a solution. **Redis :** Acts as the central coordination layer, managing system state, leader election, and inter-node communication via pub/sub. Its in-memory performance ensures low-latency operations. **MongoDB** : Provides persistent storage for job metadata, execution history, and system state. It is chosen for its scalability and flexible schema, aligning with CronFlow's demands. **API Server & Dashboard** : The API layer handles job management, registering, monitoring, and external requests, while the admin dashboard offers a simplified interface for real-time logs and system observability. Both Redis and

MongoDB are scaled to meet the system's requirements, ensuring CronFlow delivers high availability and consistent performance.

2. Core Design Principles

2.1 Scalability

The system implements stateless workers that auto-scale load-deistribute on demand based on the job queue depth. In-case of failure of any of the worker nodes a robust load-balancing algorithm kicks in ensuring not only the node fails gracefully but also prevents other nodes from failing due to overwhelming system resources with jobs.

This can be achieved via the designed system state as each worker node reports consistently the available system resources. The leader node task is to run the load-balancing algorithm to redistribute the failing node workloads. This allows for handling millions of scheduling jobs for millions of users.

2.2 High Availability

The system implements automatic failover for both worker nodes and leader nodes. Leader nodes send heartbeat to renew contract every configurable millisecond. The worker nodes update its state consistently which allows for graceful degradation and high availability for both worker nodes and leader nodes.

Every worker node has a control panel and the logic needed to be elected as leader node.

2.3 Consistency

Distributed locks with fencing tokens allow for a strong consistency model. Before any scheduling on any worker node, workers are required to acquire a lock on that specific job. If any node acquired a lock on that job, other nodes cannot schedule it.

A fencing token prevents out of order execution which is a unique monotonically increasing number when the node acquires a lock on job to schedule it.

2.4 Fault Tolerance

Jobs are retried with backoff both the retry interval and the backoff are configurable with max retries. Jobs are recovered offhandedly. When a job is stuck and timeout it's reassigned via a verification sweep mechanism. The verification sweep is a cron job that runs at configurable time daily preferably at time where there is not much traffic (e.g. 00:00 UTC).

When a node fails or external service the failure should not cascade to other nodes. This can be implemented via circuit-breaker.

3. Leader-Follower Implementation

3.1 Leader Election

The first node spin in the system will acquire a lock from system memory (Redis) with

configurable TTL leader lock for 30s. The leader node sends a heartbeat update to renew the leadership contract every 15s. Followers trigger failover detection every 45s so they can become the new leader if the leader node fails.

3.2 Job Distribution

Leader assigns jobs via Redis pub/sub. The leader gets triggered by a published event from the API-server. Workers pull jobs based on capacity (CPU/Memory). All worker nodes are required to keep buffer for any of them to take on leadership when needed.

4. Distributed Locking Mechanism

4.1 Lock Acquisition

Every job is locked via a simple k,v data structure in redis as follows *lock:job:<jobId>* . The lock has configured TTL with 5s buffer in-case the node can't schedule it for whatever reason, the job scheduling will timeout and publish to another node to reschedule it. The fencing token will be a timestamp generated from 01 January 1970 UTC. This allows for overlocking the networking partitioning and ntp servers since most programming languages generate the exact increasing monotonically timestamp.

4.2 Distributed Lock Fault tolerance and error handling

Three main scenarios the system must handle well in cause of failures :

1. Worker crashes.
2. Network partition.
3. Long GC pauses.

In case of a worker crashing the lock designed to expire after set time and the job is reassigned to another node. Fencing tokens reject stale requests preventing network partitioning, the timestamp is generated independently of the operating system with consistent across time-zones, machines algorithm. The lock is released at specific time preventing long GC pauses or anything of that nature to prevent the job from being scheduled.

5. Failure Recovery and Self-healing

6.1 Worker failure

Workers send health checks intervally a missed heart beat will render the worker node as unhealthy thus no new job will be scheduled on it, and at set time it will be drained and jobs will be requeued elsewhere.

6.2 Leader failure

Followers detect missed leadership renewal every 45s. The time is chosen carefully as there is no need to overwhelm the system

with multiple leadership contract health checks. The 45s is a delay of scheduling new jobs; however the jobs will not be lost as soon as worker nodes are elected as new leaders.

6. Security Model

Api-server uses JWT Auth, Rate limiting for the admin portal. The credentials of the admin are configured via system parameters. The admin portal is currently nothing but a real-time dashboard and logs.

Best security practices in server, databases, frontend are overall followed.

Workers communicate with master via redis eliminating the need for authentication in registering. As all nodes needs the redis cluster credentials to operate in the system.

7. Monitoring

The dashboard shows metrics for the leader node, worker nodes with graphs and statistics that allow the admin system to gauge a view over the performance of the system in real-time. No interval refreshing in the UI/UX, a websocket is used and events are streamed into a single entity that the api-server publishes every second.

The refresh is not noticable for the user and the dashboard should look fluent and visually appealing.

Logs as well are events published over the websocket and persisted in the database with configurable retention period.

8. Conclusion

CronFlow provides a simple yet robust distributed scheduling system that is ready for any scale with the following guarantees :

1. Strong consistency : via fencing tokens and quorum writes.
2. High Availability : Through leader, worker failover and self-healing workers.
3. Fault tolerance with retries, reassignment and circuit breakers.
4. Horizontal scalability via stateless workers and shared storage.