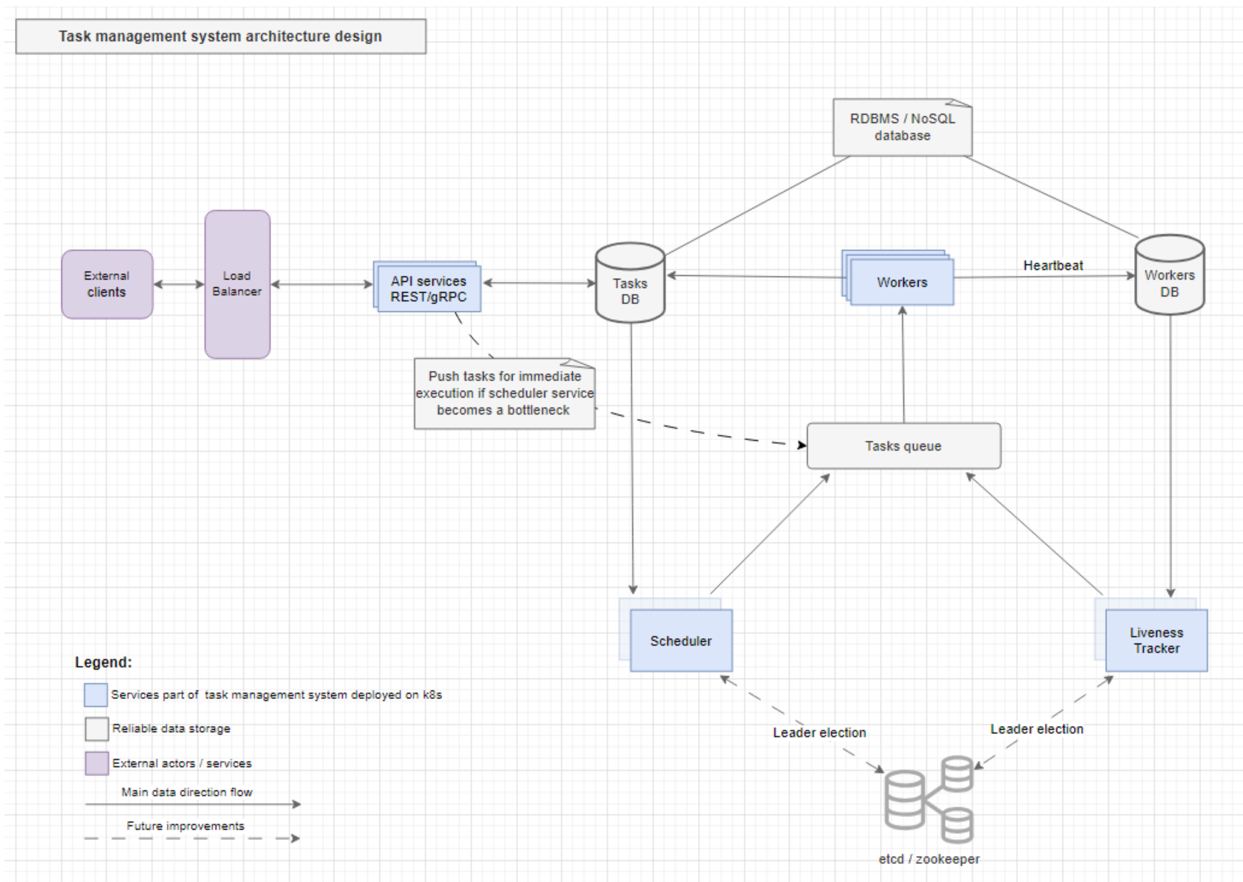
Task management system design

Architecture

General architecture design
https://drive.google.com/file/d/17mZ1-tJQhcNmXO3QX3nEnFNOCdvPviOx/view?usp=sharing



Task lifecycle

Task statuses:

- `SCHEDULED` - The task is scheduled for execution in future (`task.start_at > now()`)
- `PENDING` - The task is waiting for a free worker to pick it up and start execution.
- `RUNNING` - The task is currently in-progress and being executed by a worker.
- `FINISHED` - Task execution complete (and the task is not recurring).

When a task is submitted to the system via the API it is created in the DB with `SCHEDULED` status and `task.start_at` is set to a date-time when that task execution should start (for tasks that should start immediately `task.start_at = now()`). When it's time for the task to be executed, Scheduler Service picks it up from the database, sets the status to `PENDING` and sends it's Id to the Tasks Queue. When any worker has free capacity it pools a task Id from the

Tasks Queue, sets the status to `RUNNING` and starts the execution. After the worker finished with the task it either sets it in `FINISHED` state (for non-recurring tasks) or re-schedule it again for another execution setting status to `SCHEDULED` with an updated `task.start_at` time based on the recurrence interval.

Scheduler Service

Scheduler Service is responsible to periodically pool 'ready' tasks Ids (`task.start_at < now()`) from the database in large batches (`MAX_POLL_RECORDS`) and send them to Tasks Queue for asynchronous execution in future.

Worker Service

Worker service pools `PENDING` task Ids from Tasks Queue and runs up to `CONFIG:CONCURENT_TASKS` tasks concurrently on each node. The service sets the task status to `RUNNING` and inserts a record in WorkersDB to associate the task with the current worker before execution starts. After the execution finishes the status is set to `FINISHED`/`SCHEDULED` and the association record is removed.
Worker Service periodically updates its 'heartbeat' record in Workers DB (every `CONFIG:HEART_BEAT_INTERVAL_SEC` seconds).If the record can not be updated successfully for `CONFIG:WORKER_TIMEOUT_SEC` seconds the service exits.

Liveness Tracker Service

Liveness Tracker Service checks Worker's heartbeats periodically. If a worker does not heartbeat in `CONFIG:WORKER_TIMEOUT_SEC` seconds the worker is considered dead and Liveness Tracker re-enqueue (set status to `PENDING`) all worker tasks so they can be picked up by another healthy worker.

Database (ScyllaDB for Tasks/Workers)

For a range of a million tasks the database can be either an ACID compliant RDBMS or an eventually consistent NoSql database. Current implementation uses ScyllaDB - an eventually consistent Cassandra compliant database.

Pros/Cons for using RDBMS

- Pros:
    - Simpler design and implementation of the system.
    - Easier to prove correctness of the implementation.
    - No corner cases related to clock drifts.
- Cons
    - Not horizontally scalable.
    - Single point of failure.

Pros/Cons for using NoSql database

- Pros:

        - Highly scalable.
        - Highly available / no single point of failure.
    - Cons
        - More complex design.
        - Harder to prove correctness, many edge cases related to updates ordering.

Database schema

Tasks table definition

```
CREATE TABLE tasks (
    id text,
    status int,
    start_at timestamp,
    recurring text,
    command text,
    PRIMARY KEY (id)
);
```

Scheduled tasks table definition

```
CREATE TABLE tasks_by_schedule (
    when date,
    scheduled_for timestamp,
    task_id text,
    PRIMARY KEY (when, scheduled_for, task_id)
) WITH CLUSTERING ORDER BY (scheduled_for ASC);
```

Workers table definition

```
CREATE TABLE workers (
    id text,
    last_heartbeat timestamp,
    PRIMARY KEY (id, last_heartbeat)
) WITH CLUSTERING ORDER BY (last_heartbeat ASC);
```

Tasks by worker table definition

```
CREATE TABLE tasks_by_worker (
    worker_id text,
    task_id text,
    PRIMARY KEY (worker_id, task_id);
)
```

Tasks Queue (Kafka)

API Service

API service exposes a simple REST API for creating and retrieving tasks. Currently the system does not run real commands, but instead the `command` parameter from `ScheduleTaskRequest` specifies a duration to sleep thus simulating real work with varying duration.

Request / Response JSON format

```
ScheduleTaskRequest:
{
    "command": "<sleep duration string (3s | 10ms …)>",
    "start_at": "<RFC 3339 date-time>",
    "recurring": "<recurring duration string (3s | 10ms …)>"
}
```

- `command [Required]:` Sleep duration.
- `start_at [Optional]:` When the task should start execution. If not provided the task will be scheduled for execution immediately.
- `recurring [Optional]:` How long to wait before the next task execution. If not provided the task is not recurring and will be executed once.

```
TaskResponse:
{
    "id": "<uuid>",
    "status": "SCHEDULED | PENDING | RUNNING | FINISHED",
    "start_at": "<RFC 3339 date-time>",
    "recurring": "<recurring string (3s | 10ms …)>",
    "command": "command string"
}
```

Endpoints

Schedule Task

HTTP Request:

```
URL:            /tasks
Method:         POST
Body (JSON):    <ScheduleTaskRequest>
```

HTTP Response status codes and payload:

```
HTTP 200: <TaskResponse>
HTTP 400: Invalid request payload
HTTP 500: Error description
```

List tasks

NOTE: This endpoint is for demo/test only
HTTP Request:

```
URL:       /tasks
Method:    GET
```

HTTP Response status codes and payload:

```
HTTP 200: [<TaskResponse>, <TaskResponse>, …]
HTTP 500: Error description
```

Get task

HTTP Request:

```
URL:       /tasks/<task_id>
Method:    GET
```

HTTP Response status codes and payload:

```
HTTP 200: <TaskResponse>
HTTP 404: Task not found
HTTP 500: Error description
```

In scope

- Implementation of Worker service
- Implementation of Scheduler Service
- Implementation of Liveness Tracker Service
- Implementation of API Service

Out of scope / Todo

- Units and integration tests.
- Better code documentation.
- Correct API for retrieving tasks with filtering/limiting result set.
- API for tasks cancelation / stop recurring tasks.
- Creation of container images and resources for Kubernetes deployment.
- Swagger documentation.
- Metrics/Logging.
- Throttling.
- Leader election for Scheduler/LivenessTracker
- Authentication and authorization.

Scalability

Scalability of the individual components:

- API service is stateless and can scale horizontally to a large number of pods by kubernetes HPA based on current service load.
- Worker service can effectively scale up to the number of partitions in Kafka topics. Increasing the number of partitions allows the service to scale up horizontally.
- Kafka scales horizontally by increasing the number of partitions.
- ScyllaDB is by design highly available and scalable database.

Availability

Availability of the system is determined by the number of healthy vs fault nodes running Kafka, Zookeeper and ScyllaDB clusters.

- Zookeeper cluster (or KRaft) is available and can make progress if at least N/2+1 of the nodes forming the cluster are healthy.
- Kafka topics are available and can accept new records if at least `min.insync.replicas` nodes are healthy and are accessible by the topic leader.
- ScyllaDB can accept writes if at least N/2+1 replica nodes are healthy.

Guarantees

The system guarantees "at least once" execution by:

- API service writes to the database with `CONSISTENCY=QUORUM` to ensure the majority of the replica nodes have the task persisted before a successful HTTP 200 response is returned.
- Scheduler Service sends records to Kafka with `acks=all` configuration, requesting that all ISR replicas have the record persisted before a successful response is returned.
- In addition, Kafka cluster must be configured with `unclean.leader.election.enable=false`. This ensures that once a message is committed it can not be lost due to a leader broker crash.
- Worker Service commits his current progress (Kafka partition offset) after the task Id is persisted in Workers DB (as "owned" by the current worker) with `CONSISTENCY=QUORUM`.

Run locally:

1. Run ScyllaDB and Kafka in docker with docker-compose
   `docker-compose -f .\docker\docker-compose.yaml up -d`
2. Create a kafka topic named `tms` with multiple partitions
3. Create database schema with cqlsh using schema file:
   `.\internal\store\scylla_create.cql`
4. Run Scheduler Service
   `go run .\cmd\scheduler\main.go`
5. Run Liveness Tracker Service
   `go run .\cmd\liveness-tracker\main.go`
6. Run multiple instances of Worker Service
   `go run .\cmd\worker\main.go`
7. Run API Service
   `go run .\cmd\http_api\main.go`
8. Sen a dummy task (sleep for 5 seconds) for execution with curl or postman
   `curl -X POST http://localhost:3369/tasks -H 'Content-Type: application/json' -d '{"command": "5s"}'`