Selinon – Dynamic Distributed Task Flows

Fridolín Pokorný

Red Hat Czech s.r.o., Purkyňova 99, 612 00 Brno
fridolin@redhat.com

Abstract. There has been a significant growth in volume of data produced and available publicly on the Internet as well as in private sources. Processing such volume of data can be challenging and in many cases time consuming for a single computer. In this paper we present a distributed system made of simple tasks that can be executed with high level of parallelism using multiple cluster nodes. System behaviour is defined via a simple configuration file. This declarative abstraction model and distributed architecture allows to model complex data flows while assuring stability and resilience even during online production redeployment.

Keywords: big data · data processing · distributed systems · Python

1 Introduction

In the upcoming sections there will be discussed core concepts of distributed task queues, briefly introduced project Celery that is used by Selinon as a task scheduling backend, Celery's disadvantages and basic abstractions used in project Selinon we developed that helped us horizontally scale a self-sustaining system able to process big data.

2 Distributed Task Queues

The idea of distributed systems is definitely not novel. There exist multiple implementations of distributed task queues – in the Python ecosystem, there can be found namely project Celery [3], Spotify Luigi [8], Apache Airflow [7] and others. Most of these solutions require a broker and workers. A broker (also known as message broker or message queue) handles queueing messages which carry information what should be computed. Subsequently, there are connected workers to the broker which consume messages and process them. If a message is processed, it is removed from message broker and the worker that processed the given message is available to process other messages queued on broker. Most often, there is also available a database or storage to which computed results are stored to. The described architecture can be seen on figure 1.

In many scenarios, having a single task for a request described by a message and processed by workers is not desired. In order to take advantage of horizontally scalable solution the request can be split into standalone units, called

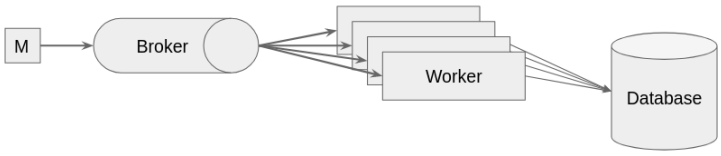

Fig. 1: An architecture of a distributed task queue.

tasks, that compute one type of a result which can be additionally processed by other standalone tasks (also known as task flows, see below). The advantage of such logical split is mostly in possible paralelism and task retries in case of task failures. If a task fails, it can be retried without any negative impact on other independent tasks that can meanwhile run in parallel on different workers.

3 Using Celery for Distributed Task Queue

All of the objectives mentioned in the previous section 2 can be accomplished with stated open source solutions from Python ecosystem. In fact, we used Celery to accomplish desired behaviour. Celery offers "Celery primitives" that allow to model task flows – dependencies between tasks that can be eigher data dependencies (waiting for results of a task in order to start other tasks) or time dependencies (waiting on external events to proceed to next tasks). Celery primitives allow grouping tasks so tasks are executed in parallel or sequentially possibly on different workers in parallel.

An example of task flow modeling using Celery primitives can be seen on figure 2 (the example is intentionally simplified, see Celery documentation [4] for more in depth Celery primitives explanation). With Celery primitives we can create "phases" which group tasks that can be executed in parallel as there are no inter-task dependencies inside phases. Phases need to be executed one after another sequentially due to time or data dependencies of tasks that create these phases.

The task flow made of phases using Celery's primitives is hard-coded inside application logic that makes application harder to maintain, harder to extend, more error prone with task or phase updates and adding a new task into an existing flow requires knowledge of internal dependencies of tasks that could be harder to see from source code. Also, in the example above, the long running tasks block execution of tasks that could be already started respecting task flow dependencies.

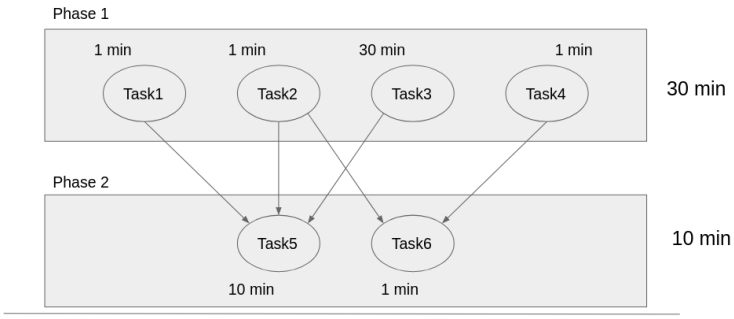

Fig. 2: Phases created using Celery's primitives for tasks. Task `Task6` could be executed after 1 minute assuming `Task2` and `Task4` finish in one minute, but `Task3` block execution of `Task6`.

4 Orchestrating Task Flows with Selinon

As we experienced drawbacks when using raw Celery, we managed to design a task flow orchestration tool on top of Celery which removes phase borders and all of the disadvantages stated above.

In the upcoming subsections, there are introduced Selinon abstractions. These abstractions create core components of Selinon design that act like a loosly coupled entities which, based on configuration, can create complex systems capable of serving large workload requests.

4.1 Tasks and Task Flows

A Selinon task is abstracted into a Python class respecting Selinon task interface. Implemented Selinon tasks are subsequently registered in the Selinon YAML configuration file and used to create task flows. The configuration of a flow states edges (or transitions) that model data or time dependencies between tasks. An example of such flow configuration can be seen on figure 3.

In the example shown in figure 3, Selinon starts `Task0` and `Task1`. Once `Task1` finishes, there are started tasks `Task2` and `Task3` that can run in parallel without possibly waiting for `Task0` to finish or fail. The `condition` section restricts `Task2` and `Task3` execution only for production systems (environment variable `PRODUCTION` is present). The conditional execution of tasks make the whole system dynamic – these conditions are evaluated each time an edge is fired and they can perform various actions such as checking external events or inspecting results of previous tasks. Their main purpose is to abstract away conditional task execution from actual task implementation logic as well as an optimization to prevent from queueing overhead needed to schedule unwanted tasks.

Besides scheduling tasks directly, Selinon is capable of creating nested flows – scheduling flows from within a flow. Moreover, the scheduling algorithm that

```
flow-definitions:
  - name: flow1
      - from:
        to:
          - Task0
          - Task1
      - from: Task1
        to:
          - Task2
          - Task3
        condition:
          name: envExist
          args:
            name: PRODUCTION
```

Fig. 3: An example of Selinon flow configuration.

Selinon offers removed a restriction on directed acyclic graph (DAG) in flow design which other open source alternatives have.

4.2 Storage and Database Adapters

If there is a requirement for storing computed results there can be assigned storage adapters that handle task result storing and possible task result retrieval. In order to use an adapter, the adapter needs to be registered into the system in the YAML configuration file and assigned to a task.

This separation of task logic (how results are computed) from actual storing and retrieval mechanism allowed us to use transparently different storage and database solutions to store data in a way they could be later easily queryable based on data characteristics and query types we wanted to perform on computed results.

4.3 Application Monitoring and Healthiness

Selinon offers a built in support for application monitoring using flow tracepoints. Tracepoints are events that occur in cluster (such as a flow has started, a task has started, a task has failed) that help to monitor cluster behaviour. Selinon comes with a built-in support of error tracking software (such as Sentry) as well as structured well-defined logs that can be aggregated and further inspected using cluster-wide aggregated logging.

5 Conclusion

Selinon is an open source project, it's source code is available on GitHub [3]. It is used in the OpenShift.io [6] project for gathering and processing big data about

packages in various ecosystems. With Selinon, we were able to create complex task flows that processed large amount of data stored in different database systems based on requirements. The configuration-centric design of loosely coupled components allowed us to perform system checks before actual deployment and gave us an ability to create easy to monitor dynamic system which behaviour can be changed even in online production systems without affecting already queued workload.

References

1. Selinon project homepage, https://selinon.readthedocs.io/. Last accessed 28 May 2018
2. Selinon project on GitHub, https://github.com/selinon/. Last accessed 28 May 2018
3. Celery project homepage, https://celeryproject.org. Last accessed 28 May 2018
4. Celery project documentation, docs.celeryproject.org/. Last accessed 28 May 2018
5. The Official YAML Web Site, http://yaml.org/. Last accessed 28 May 2018
6. OpenShift.io: An Open Online Development Environment, https://www.openshift.io. Last accessed 28 May 2018
7. Apache Airflow documentation, https://airflow.apache.org. Last accessed 28 May 2018
8. Spotify Luigi project on GitHub, https://github.com/spotify/luigi. Last accessed 28 May 2018
9. Sentry: Error Tracking Software, https://sentry.io. Last accessed 28 May 2018