Cahier des charges

# I – Définition et objectif

Un « Job Scheduler » permet de programmer l’envoi d’événements programmés (« delayed event dispatching ») à travers un message broker.

Le service est « platform agnostic » et « protocol agnostic », on peut créer des modules pour utiliser différents protocoles de communication.

Le « Job Scheduler » doit avoir les facultés suivantes :

* être fiable : un ∆ maximum de 2 secondes entre le timestamp prévu et le timestamp réel auquel l’événement est dispatché
* être résilient : il doit gérer plus de 10M d’événements en parallèle sans de hausse du ∆
* être « error-proof » : il doit permettre la persistance des données en cas de problème du serveur ou d’extinction du service et en même temps permettre un accès rapide aux données

# II – Stack et fonctionnement simplifié

On utilisera deux caches, un de LEVEL 1 en mémoire et un deuxième de LEVEL 2, Redis.

Pour la persistance, on utilise une base de données PostgreSQL.

Le code d’exemple Javascript ci-dessous ne prend pas en compte la persistance et les caches.

const CONFIGURATION = {

  numberOfStacks: 5

};

class JobEvent {

  shouldExecuteAt = new Date();

  payload = null;

  cronExpression = "";

  constructor(cronExpression, payload) {

    this.cronExpression = cronExpression;

    this.payload = payload;

  }

  nextTimestamp() {

*// Dummy code here*

    this.shouldExecuteAt = new Date();

*return* this.shouldExecuteAt;

  }

  copy() {

*return* new JobEvent(this.cronExpression, this.payload);

  }

}

class StackManager {

  stacks = [];

leastFilledStackIndex = 0;

  constructor(numberOfStacks) {

*for*(let i = 0; i < numberOfStacks; i++)

      this.stacks.push([]);

  }

  push(e) {

this.stacks[this.leastFilledStackIndex].push(e);

*// Resort the stack*

    this.stacks[this.leastFilledStackIndex] = this.stacks[this.leastFilledStackIndex].sort((a, b) => a.shouldExecuteAt < b.shouldExecuteAt);

*// Put the least filled stack first*

    this.leastFilledStackIndex = this.stacks.map((s, i) => ({length: s.length, index: i})).sort((a, b) => a.length < b.length)[0].index;

  }

  get(i, j) {

*return* this.stacks[i][j];

  }

  remove(i, j) {

    this.stacks[i] = this.stacks[i].filter((\_, a) => a !== j);

  }

  pop(i) {

    const el = this.stacks[i].pop();

    const elCopy = el.copy();

    elCopy.nextTimestamp();

    this.push(elCopy);

*return* el;

  }

}

class Worker {

  stackManager = null;

  stackIndex = null;

  shouldRunFn = null;

  dispatchEventFn = null;

  constructor(stackManager, stackIndex, shouldRunFn, dispatchEventFn) {

    this.stackManager = stackManager;

    this.stackIndex = stackIndex;

    this.shouldRunFn = shouldRunFn;

    this.dispatchEventFn = dispatchEventFn;

  }

  run() {

    let ev = this.stackManager.pop(this.stackIndex);

    const getNext = () => {

      ev = this.stackManager.pop(this.stackIndex);

    };

*while*(this.shouldRunFn()) {

*if* (ev.shouldExecuteAt.getTime() >= Date.now())

*continue*;

      this.dispatchEventFn(ev);

      getNext();

    }

  }

}