MesosCon
ASIA
NOVEMBER 18-19, 2016 | HANGZHOU, CHINA

Building MongoDB ReplicaSet Framework

Chao Liu, DC/OS Contributor
Sam Chen, DC/OS China Community

目的

- 针对初级及其中级框架开发人员
- 普及框架的开发和应用
- 提倡参与社区贡献
- 加大知识传播
- 增强开发框架意识
- 明确框架价值和意义

MongoDB Repo： https://github.com/dcos-labs

MongoDB 框架现状

Ansible script to install DC/OS on Google Compute Engine

Python ★ 20 4 Updated on 13 Oct

dcos-config-tool

A config UI to browse and create configurations for DC/OS packages

Scala ★ 10 Updated on 24 Sep

mesos

Forked from apache/mesos

Mirror of Apache Mesos

C++ ★ 1 1,073 Updated on 7 Sep

marathon-validate

A tiny command line tool to validate application or group configuration files for Marathon

JavaScript ★ 11 3 Updated on 17 Jun

drax

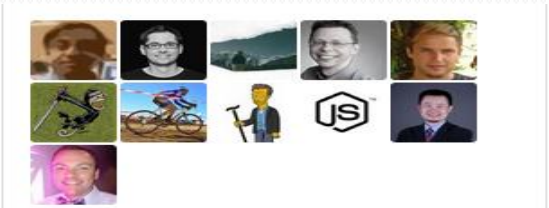
DC/OS Resilience Automated Xenodiagnosis tool

Go ★ 28 1 Updated on 15 Jun

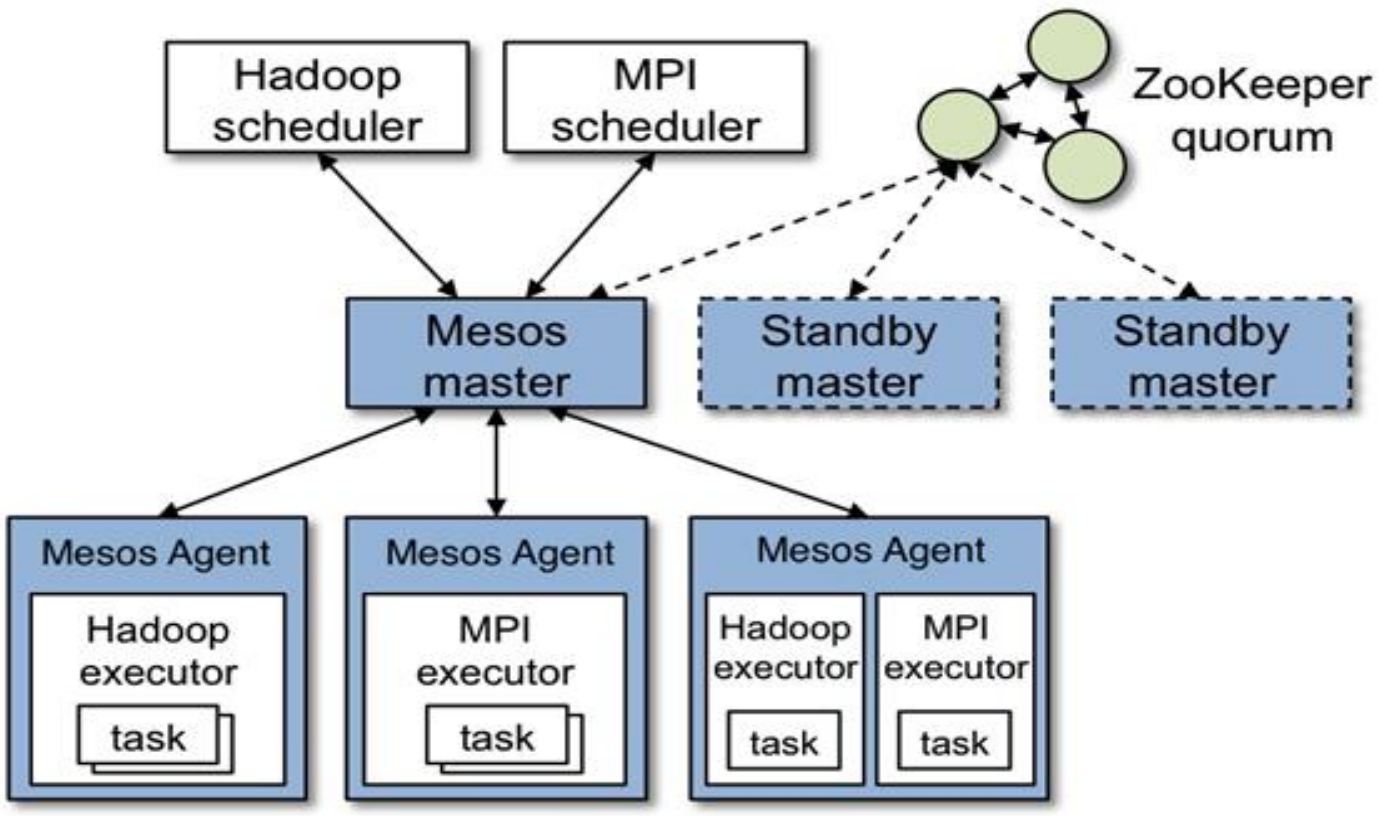
MongoDB

MongoDB Framework

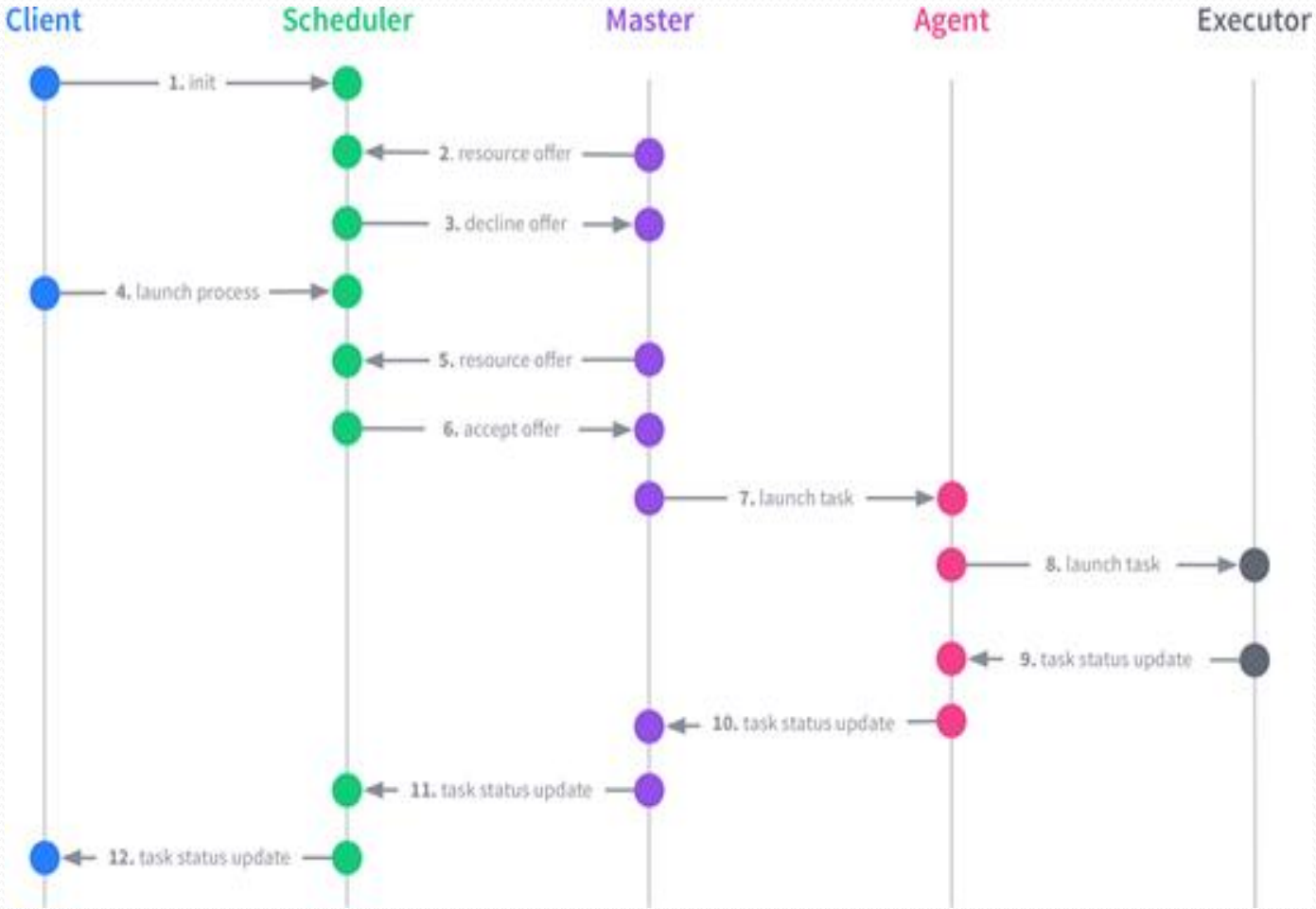
Go ★ 5 3 Updated on 12 Jun



Mesos架构

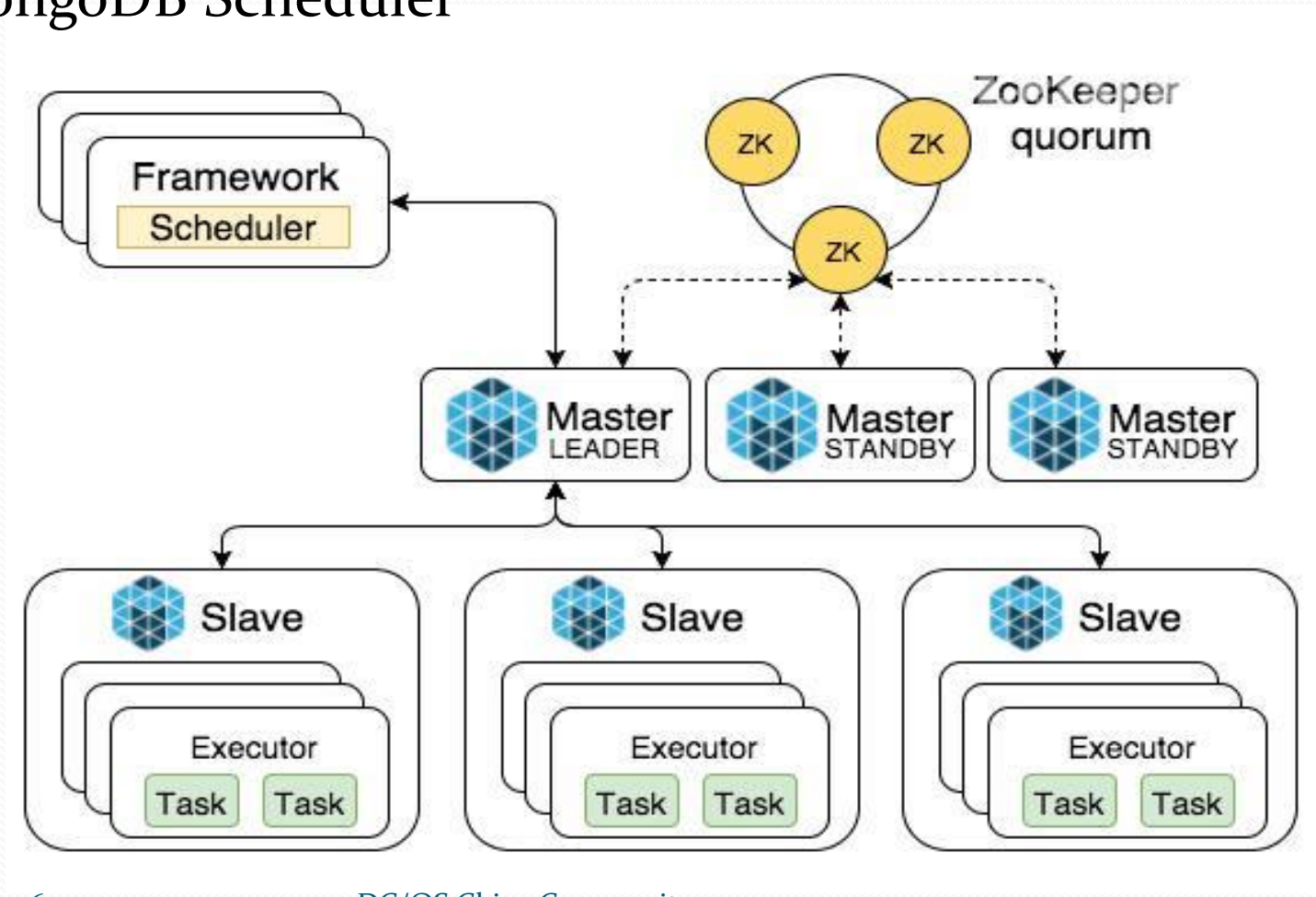

Hadoop scheduler
MPI scheduler
ZooKeeper quorum
Mesos master
Standby master
Standby master
Mesos Agent
Hadoop executor
task
Mesos Agent
MPI executor
task
Mesos Agent
Hadoop executor
task
MPI executor
task

Mesos Task Lifecycle



MongoDB Framework

MongoDB Scheduler



第一层调度：HierarchicalDRF

- 主导资源公平算法 Dominant Resource Fairness
  - Framework拥有的全部资源类型份额中占最高百分比的就是Framework的主导份额。
  - DRF算法会使用所有已注册的Framework来计算主导份额，以确保每个Framework能接收到其主导资源的公平份额。

第一层调度： HierarchicalDRF

- 调用了三个排序器Sorter(quotaRoleSorter, roleSorter, frameworkSorter)，对所有的Framework进行排序，哪个先得到资源，哪个后得到资源。
- 总的来说分两大步：
  - 先保证有quota的role，调用quotaRoleSorter，然后其他的资源没有quota的再分，调用roleSorter。
  - 对于每一个大步分两个层次排序：
    - 一层是按照role排序
    - 第二层是相同的role的不同Framework排序，调用frameworkSorter。
      - 每一层的排序都是按照计算的share进行排序来先给谁，再给谁。Share的计算就是按照DRF算法。

第一层调度： HierarchicalDRF

- 1. 生成一个数据结构offerable，用于保存资源分配的结果
  - hashmap<FrameworkID, hashmap<SlaveID, Resources>> offerable;
  - 这是一个MAP，对于每一个Framework，都会有一个资源的MAP，保存的是每个slave上都有哪些资源。
- 2. 对于所有的slave打乱默认排序，从而使得资源分配相对均匀
  - std::random_shuffle(slaveIds.begin(), slaveIds.end());

第一层调度： HierarchicalDRF

- 3. 进行第一次三层循环，对于有quota的Framework进行排序
  - foreach (const SlaveID& slaveId, slaveIds) {
  - foreach (const string& role, quotaRoleSorter->sort()) {
  - foreach (const string& frameworkId_, frameworkSorters[role]->sort()) {
- 对于每一个slave，首先对role进行排序，对于每一个role，对于Framework进行排序，排序靠前的Framework优先获得这个slave。
- 排序的算法在DRFSorter里面实现，里面有一个函数calculateShare，里面的关键点在于进行了一个循环，对于每一种资源都计算如下的share值：share = std::max(share, allocation / _total);
- allocation除以total即一种资源占用的百分比，这里之所以求max，就是找资源占用百分比最高的资源，也即主导资源。
- 但是这个share不是直接进行排序，而是share / weights[name]除以权重进行排序。如果权重越大，这个值越小，这个role会排在前面，分配更多的资源。
- 排序结束后，对于每一个Framework，将当前slave的资源分配给它。Resources available = slaves[slaveId].total - slaves[slaveId].allocated;
- 首先查看这个slave的可用资源，也即总资源减去已经分配的资源。Resources resources = (available.unreserved() + available.reserved(role)).nonRevocable();
- 每个slave上没有预留的资源和已经预留给这个Framework的资源都会给这个Framework，当然如果上面有预留给其他Framework的资源是不会给当前的Framework的。offerable[frameworkId][slaveId] += resources; slaves[slaveId].allocated += resources;
- 分配的资源会保存在数据结构offerable中。

第一层调度： HierarchicalDRF

- 4. 进行第二次三层循环，对于没有quota的Framework进行排序
  - foreach (const SlaveID& slaveId, slaveIds) {
  - foreach (const string& role, roleSorter->sort()) {
  - foreach (const string& frameworkId_,frameworkSorters[role]->sort()) {
- 5. 全部分配结束后，将资源真正提供给各个Framework
  - foreachkey (const FrameworkID& frameworkId, offerable) {
  - offerCallback(frameworkId, offerable[frameworkId]);
  - }

第二层调度：写一个Scheduler

- 实现接口
  - registered、reregistered、disconnected
  - resourceOffer
  - offerRescinded
  - statusUpdate
  - frameworkMessage
  - slaveLost:
  - executorLost
  - error

第二层调度：写一个Scheduler

- MesosSchedulerDriver来运行这个Scheduler

```
MesosSchedulerDriver* driver;
TestScheduler scheduler(implicitAcknowledgements, executor, role);

driver = new MesosSchedulerDriver(
  &scheduler,
  framework,
  master.get(),
  implicitAcknowledgements,
  credential);
```

Developing an Executor

- 实现接口
  - Registered, reregistered, disconnected
  - launchTask
  - killTask
  - frameworkMessage
  - Shutdown
  - Error
- MesosExecutorDriver将这个Executor运行起来

Start the Framework

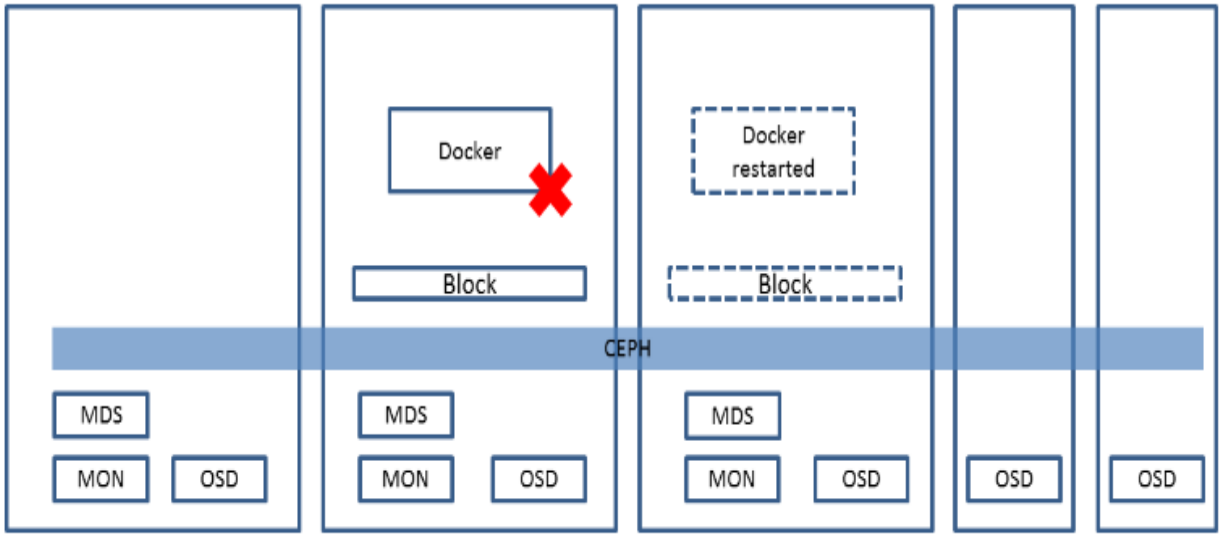
```
func Start(master *string) {
    log.Debugln("startScheduler master:", *master)

    fwinfo := &mesos.FrameworkInfo{
        User: proto.String(""),
        Name: proto.String("mongodb-mesos"),
        FailoverTimeout: proto.Float64(24*3600*1000),
        Checkpoint: proto.Bool(true),
    }

    config := sched.DriverConfig{
        Scheduler: newMongodbScheduler(),
        Framework: fwinfo,
        Master:    *master,
    }

    driver, err := sched.NewMesosSchedulerDriver(config)
    if err != nil {
        log.Errorln("Unable to create a SchedulerDriver ", err.Error())
    }

    stat, err := driver.Run()
    if err != nil {
        log.Infof("Framework stopped with status %s and error: %s", stat.String(), err.Error())
    }

    log.Infof("stat:%v", stat)
}
```

Launch a Task

```
task := &mesos.TaskInfo{
    Name:     proto.String(replicaTaskID(rs, db)),
    TaskId:   taskID,
    SlaveId: offer.SlaveId,
    Container: &mesos.ContainerInfo{
        Type: &taskType,
        Docker: &mesos.ContainerInfo_DockerInfo{
            Image: proto.String("mongo:3.2.6"),
            PortMappings: []*mesos.ContainerInfo_DockerInfo_PortMapping{
                &mesos.ContainerInfo_DockerInfo_PortMapping{
                    HostPort:      &hostPort32,
                    ContainerPort: &containerPort,
                    Protocol:      &protocol}},
            Network: &network,
        },
    },
    Command: &mesos.CommandInfo{
        Shell:     proto.Bool(false),
        Arguments: []string{"--replSet", rs.Name},
    },
    Resources: []*mesos.Resource{
        util.NewScalarResource("cpus", float64(db.Cpu)),
        util.NewScalarResource("mem", float64(db.Memory)),
        util.NewRangesResource("ports", []*mesos.Value_Range{
            &mesos.Value_Range{
                Begin: &hostPort,
                End:   &hostPort,
            }}),
    },
}
```

MongoDB的数据

- DC/OS的存储的管理主要分三种方式：
  - 本地临时存储(Local ephemeral storage)
    - 无状态任务
  - 本地持久化存储(Local persistent volumes)
    - 在创建服务的时候，所有的服务的资源，包含CPU，内存，硬盘都被预留reserved，从而当这个服务挂掉的时候，还是会从同样的机器上启动起来，容器内部的路径还是会被mount为相同的主机路径
  - 外部持久化存储(External persistent volumes)。



欢迎参加DC/OS和Mesos中国社区