基于OpenStack的网络通信数据采集研究 期末报告

董宇茜 2010011250　　王国赛 2010011254
指导老师：徐葳

一、选题背景

随着近年云计算的普及，越来越多的小型网络开始搭建私有的云平台。OpenStack作为旨在为私有云提供建设和管理软件的项目应运而生。目前OpenStack主要提供服务器上虚拟机的管理平台，为用户提供建立虚拟机以及使用的服务。服务器上虚拟机数量、分布以及运行程序的不同会导致实际物理网络的流量变化，基于此情形，我们希望对网络数据进行采集分析，以求改进某些方面的性能。

二、研究实践计划

对OpenStack的研究首先需要以下准备步骤：

1. 搭建OpenStack，熟悉搭建过程，为未来在机房中搭建更大规模的OpenStackde做准备。
2. 阅读OpenStack官网上的文档，结合了解各OpenStack各个组件的工作原理，依赖关系。
3. 了解Nova-Scheduler的工作原理、调度策略，并尝试调整、修改Nova-Scheduler的调度算法。
4. 在OpenStack上建立虚拟机并搭建Hadoop集群。

做好以上步骤后，可以开始对OpenStack本身的研究工作。具体如下：

1. 阅读openstack源码，并尝试在源码中修改以及添加代码，进行相关数据的采集，通过对数据进行整理并分析来了解网络的通讯状况与调度算法的关系。具体而言，是在openstack上部署Hadoop分布式集群，并获取虚拟机间通信的时序的traffic matrix。
2. 修改Nova-Scheduler，在不同调度策略下采集数据并分析各种调度对网络的影响。

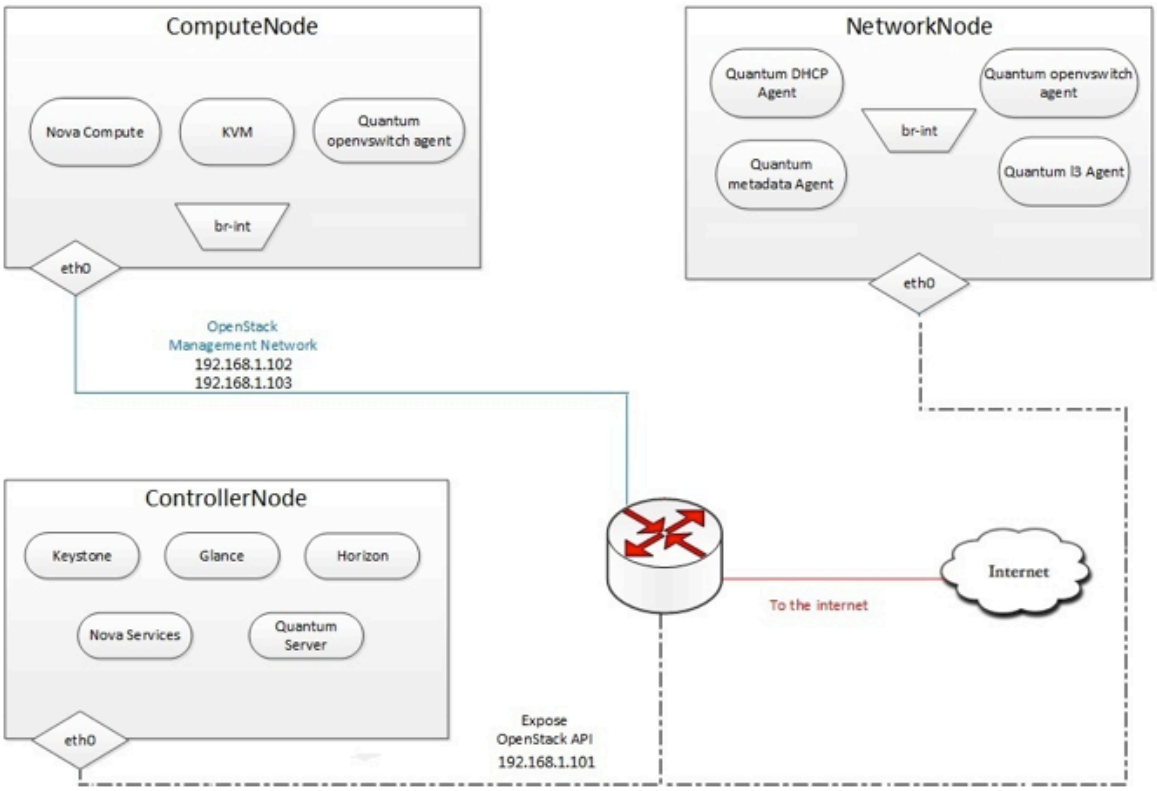
三、研究目标

对要做的每一步估算时间后我们认为，在进展比较顺利的情况下，本学期可以完成至网络通信数据采集。同时在研究过程中较深入掌握OpenStack的工作机理和搭建方式，为今后实验室搭建更大规模的OpenStack做准备，并为未来基于OpenStack展开一系列研究工作奠定基础。

四、中期进展

在这一个月中，我们通过阅读官方文档，对OpenStack的组件构成进行了详细了解，并按照Grizzly版本的一份安装指南尝试搭建OpenStack。下面的报告中我们记录了我们搭建OpenStack的过程及遇到的问题。

搭建使用3台PC，每台的配置为：4G内存，Intel(R) Core(TM) i7 CPU，320G硬盘，单网卡。我们计划的网络结构如下：



我们计划在一台PC上搭建ControllerNode和NetworkNode（openstack000），把其他两台PC作为ComputeNode（openstack002&openstack003）。图中的Switch用一台TP-Link实现，网段192.168.1.0/24。openstack000的IP配置如下

```
openstack000@ubuntu:~$ cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
#auto eth0
#iface eth0 inet dhcp

auto eth0
iface eth0 inet static
address 192.168.1.101
gateway 192.168.1.1
netmask 255.255.255.0
network 192.168.1.0
broadcast 192.168.1.255
dns-nameservers 8.8.8.8
```

其他两台的配置与上图几乎相同，除了IP地址（openstack002：192.168.1.102，openstack003:192.168.1.103）。对每台PC，我们先添加Grizzly的源，并对系统的源进行更新和内核升级。然后配置网络，开启路由转发，安装NTP服务。

openstack000上需要安装的组件与其他两个PC不同，主要内容如下（安装过程全部在超级用户模式下完成）：

1. 数据库MySQL

几乎OpenStack的所有服务都需要用到数据库，安装MySQL后，要对之后每个即将安装的组件创建相应的数据表。以keystone组件为例，创建数据表如下：

```
root@ubuntu:~# mysql -u root -p
mysql> CREATE DATABASE keystone;
mysql> GRANT ALL ON keystone.* TO 'keystoneUser'@'%' IDENTIFIED BY 'keystonePass';
```

该命令创建了一个新的数据表keystone，并设置该使用keystoneUser作为用户名，keystonePass作为密码访问该表。

2. Keystone

Keystone提供所有认证服务，包括身份验证、服务规则和服务令牌等功能。安装这个组件后，要改写配置文件/etc/keystone/keystone.conf提供正确的数据库连接（在后面的所有组件中都需要改写这一功能对应的配置文件）。改写后重启keystone服务并同步数据库：

```
root@ubuntu:~# openssl rand -hex 10
d6ee69149866667fc9d1
root@ubuntu:~# cat auth
export OS_SERVICE_TOKEN=d6ee69149866667fc9d1
export OS_SERVICE_ENDPOINT=http://192.168.1.101:35357/v2.0
```

接下来我们要向keystone的数据表填充用户、租户和服务信息。首先我们要生成一个service token并放入一个凭据文件中：

```
root@ubuntu:~# service keystone restart
keystone stop/waiting
keystone start/running, process 7192
root@ubuntu:~# keystone-manage db_sync
```

然后我们使用一个脚本文件来填充信息，所有用户如下：

```
# Tenant                User      Roles
# -------------------------------------------------------
# demo                  admin     admin
# service               glance    admin
# service               nova      admin
# service               ec2       admin
# service               swift     admin
# service               cinder    admin  //added
# service               quantum   admin  //added
```

我们给出添加一个user，并给这个user添加对应service endpoint的样例：

```
QUANTUM_USER=$(get_id keystone user-create --name=quantum \
                                           --pass="${QUANTUM_PASSWORD}" \
                                           --tenant-id $SERVICE_TENANT)

keystone user-role-add --user-id $QUANTUM_USER \
                       --role-id $ADMIN_ROLE \
                       --tenant-id $SERVICE_TENANT
```

```
QUANTUM_SERVICE=$(get_id \
keystone service-create --name=quantum \
                        --type="object-store" \
                        --description="Quantum Service")
if [[ -z "$DISABLE_ENDPOINTS" ]]; then
    keystone endpoint-create --region RegionOne --service-id $QUANTUM_SERVICE \
        --publicurl   "http://$CONTROLLER_PUBLIC_ADDRESS:9696/" \
        --adminurl    "http://$CONTROLLER_ADMIN_ADDRESS:9696/" \
        --internalurl "http://$CONTROLLER_INTERNAL_ADDRESS:9696/"
fi
```

然后我们建立一个凭据文件，可以使我们输入命令时不必每次都写明用户、密码和认证URL等信息。

```
root@ubuntu:~# cat creds-admin
export OS_TENANT_NAME=demo
export OS_USERNAME=admin
export OS_PASSWORD=secrete
export OS_AUTH_URL="http://192.168.1.101:5000/v2.0/"
```

执行结束后查看是否填充好我们想要的内容：

```
root@ubuntu:~# keystone user-list
+----------------------------------+---------+---------+-------+
|                id                |   name  | enabled | email |
+----------------------------------+---------+---------+-------+
| 693f3d1340c449caa99b92f2fbda9524 |  admin  |   True  |       |
| 3a21643dad034670ace5722cb40a1386 |  cinder |   True  |       |
| 0252705b6a4e43d39175d8005f71e358 |   ec2   |   True  |       |
| 4d319327ffd841e5a04169af1b854a67 |  glance |   True  |       |
| 5b3098d2eaaa464db9dc66d91afd718f |   nova  |   True  |       |
| 5697bd62f1db4e25b1c0672f4298ffd2 | quantum |   True  |       |
| d05aee6140a845f18aa64b104aece33d |  swift  |   True  |       |
+----------------------------------+---------+---------+-------+
```

3. Glance

下载安装后，改写配置文件/etc/glance/glance-api-paste.ini以及/etc/glance/glance-registry-paste.ini，为Glance提供身份认证地址及认证信息（用户名和密码）。

```
[filter:authtoken]
paste.filter_factory = keystoneclient.middleware.auth_token:filter_factory
delay_auth_decision = true
auth_host = 192.168.1.101
auth_port = 35357
auth_protocol = http
admin_tenant_name = service
admin_user = glance
admin_password = glance
```

配置好后下载一个用于测试的镜像cirros-0.3.1-x86_64-disk.img，上传这个镜像到镜像服务

```
root@ubuntu:~/img# glance image-create --name="CirrOS 0.3.1" --disk-format=qcow2 \
> --container-format=bare --is-public=true < cirros-0.3.1-x86_64-disk.img
+------------------+--------------------------------------+
| Property         | Value                                |
+------------------+--------------------------------------+
| checksum         | d972013792949d0d3ba628fbe8685bce     |
| container_format | bare                                 |
| created_at       | 2013-11-10T09:43:57                  |
| deleted          | False                                |
| deleted_at       | None                                 |
| disk_format      | qcow2                                |
| id               | 47503573-3847-4b75-bf96-a763474169fb |
| is_public        | True                                 |
| min_disk         | 0                                    |
| min_ram          | 0                                    |
| name             | CirrOS 0.3.1                         |
| owner            | 858ebf2e94a34986b8545fca15ccbb55     |
| protected        | False                                |
| size             | 13147648                             |
| status           | active                               |
| updated_at       | 2013-11-10T09:43:57                  |
+------------------+--------------------------------------+
```

4. Quantum和Nova

这两部分的安装没有什么特殊的，下载后改写几个配置文件，重启服务即可。对于nova，可以在安装完毕后检查服务是否启动正常

```
root@ubuntu:~# nova-manage service list
Binary           Host                                 Zone             Status     State Updated_At
nova-conductor   ubuntu                               internal         enabled    :-)   2013-11-10 09:47:19
nova-cert        ubuntu                               internal         enabled    :-)   2013-11-10 09:47:22
nova-consoleauth ubuntu                               internal         enabled    :-)   2013-11-10 09:47:26
nova-scheduler   ubuntu                               internal         enabled    :-)   2013-11-10 09:47:22
nova-compute     openstack003                         nova             enabled    :-)   2013-11-10 09:47:18
nova-compute     openstack002                         nova             enabled    :-)   2013-11-10 09:47:25
```

5. Cinder

安装配置好Cinder后，需要创建物理卷和逻辑卷，以及一个卷组：

```
root@ubuntu:~# pvcreate /dev/sdb
  Device /dev/sdb not found (or ignored by filtering).
root@ubuntu:~# dd if=/dev/zero of=cinder-volumes bs=1 count=0 seek=2G
0+0 records in
0+0 records out
0 bytes (0 B) copied, 2.5647e-05 s, 0.0 kB/s
root@ubuntu:~# losetup /dev/loop2 cinder-volumes
root@ubuntu:~# pvcreate /dev/loop2
  Physical volume "/dev/loop2" successfully created
root@ubuntu:~# vgcreate cinder-volumes /dev/loop2
  Volume group "cinder-volumes" successfully created
```

这里要注意，重启后卷组不会自动挂载，若想自动挂载可以修改/etc/rc.local。

6. Horizon

这个组件提供一个图形界面，只需要安装即可，不用配置。

7. OpenVSwitch

这是安装花费时间最长的一部分。遇到的主要问题是OpenVSwitch的安装包与当前内核版本不兼容。最开始openstack000上的内核版本是3.8.0-32-generic，但安装openvswitch-switch时提示没有符合该内核版本的安装包。后来重新安装过一次Ubuntu，在3.5.0-42-generic版本下OpenVSwitch可以顺利安装。但实际上，其他两台PC可以在3.8.0-32-generic下安装OpenVSwitch，所以还不知道openstack000上不能安装的问题源自哪里。

计算节点需要安装的内容比控制节点少很多，只需要安装并配置nova-compute、quantum-plugin-openvswitch-agent、openvswitch和kvm即可。

目前我们已经在三台机器上安装配置好所有需要的组件。可以用命令行启动一个实例：

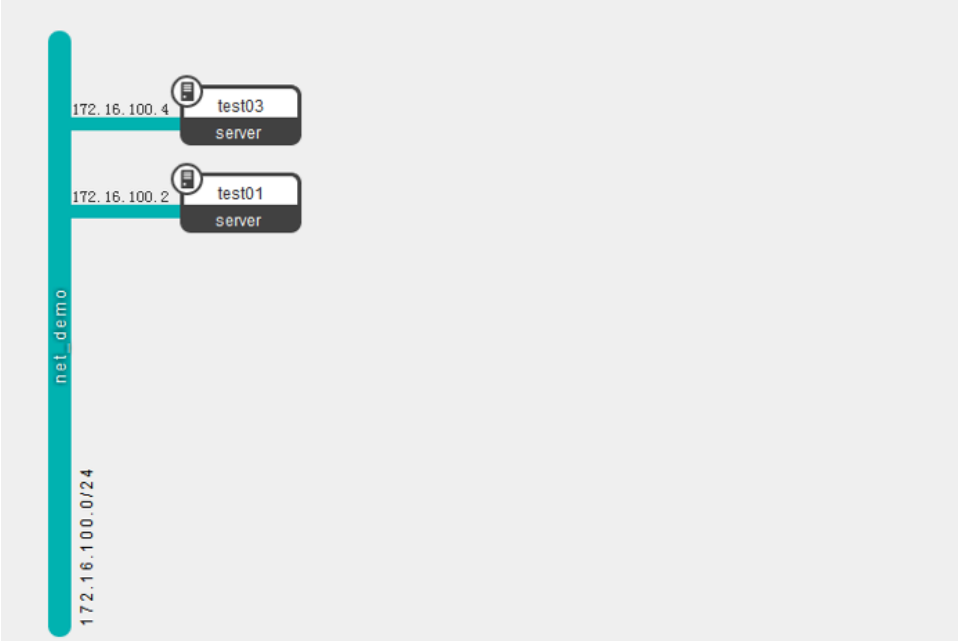
```
root@ubuntu:~# nova boot --flavor 1 --key_name mykey --image 8b5851c1-7e7b-4844-b6fa-659578db6c49 --security_group default test01
+-------------------------------------+--------------------------------------+
| Property                            | Value                                |
+-------------------------------------+--------------------------------------+
| status                              | BUILD                                |
| updated                             | 2013-11-10T12:50:34Z                 |
| OS-EXT-STS:task_state               | scheduling                           |
| OS-EXT-SRV-ATTR:host                | None                                 |
| key_name                            | mykey                                |
| image                               | cirros                               |
| hostId                              |                                      |
| OS-EXT-STS:vm_state                 | building                             |
| OS-EXT-SRV-ATTR:instance_name       | instance-00000003                    |
| OS-EXT-SRV-ATTR:hypervisor_hostname | None                                 |
| flavor                              | m1.tiny                              |
| id                                  | b9e7cee7-5290-42a7-a32a-6dffc90713a7 |
| security_groups                     | [{u'name': u'default'}]              |
| user_id                             | 693f3d1340c449caa99b92f2fbda9524     |
| name                                | test01                               |
| adminPass                           | fdfc7jWuURWW                         |
| tenant_id                           | 858ebf2e94a34986b8545fca15ccbb55     |
| created                             | 2013-11-10T12:50:34Z                 |
| OS-DCF:diskConfig                   | MANUAL                               |
| metadata                            | {}                                   |
| accessIPv4                          |                                      |
| accessIPv6                          |                                      |
| progress                            | 0                                    |
| OS-EXT-STS:power_state              | 0                                    |
| OS-EXT-AZ:availability_zone         | nova                                 |
| config_drive                        |                                      |
+-------------------------------------+--------------------------------------+
```

该实例已经被创建，等一段时间后（等待实例启动完成）查看实例状态：

```
root@ubuntu:~# nova list
+--------------------------------------+--------+--------+-----------------------+
| ID                                   | Name   | Status | Networks              |
+--------------------------------------+--------+--------+-----------------------+
| b9e7cee7-5290-42a7-a32a-6dffc90713a7 | test01 | ACTIVE | net_demo=172.16.100.2 |
+--------------------------------------+--------+--------+-----------------------+
```

可以看到实例启动完毕并且分配到了ip地址。

在Horizon中可以查看网络拓扑结构：



但虚拟机不是一直都可以顺利启动的，查看当前已启动的虚拟机：

```
root@ubuntu:~# nova list
+--------------------------------------+--------+--------+-----------------------+
| ID                                   | Name   | Status | Networks              |
+--------------------------------------+--------+--------+-----------------------+
| b9e7cee7-5290-42a7-a32a-6dffc90713a7 | test01 | ACTIVE | net_demo=172.16.100.2 |
| a69fe04b-654c-4d06-b41f-04b077766f00 | test02 | ERROR  |                       |
| 44b68061-0b65-4b63-94be-2aee594adbc3 | test03 | ACTIVE | net_demo=172.16.100.4 |
+--------------------------------------+--------+--------+-----------------------+
```

可以看到test02没有启动成功，并且172.16.100.3这一IP被跳过。现在还不知道是什么原因导致的。另外，查看当前路由代理类型：

```
root@ubuntu:~# quantum agent-list
+--------------------------------------+--------------------+--------+-------+----------------+
| id                                   | agent_type         | host   | alive | admin_state_up |
+--------------------------------------+--------------------+--------+-------+----------------+
| 0648e8e1-ba5e-40d1-b889-dd5568f66311 | Open vSwitch agent | ubuntu | xxx   | True           |
| 5abea346-dbe4-4393-ba72-2c7f9d36244e | DHCP agent         | ubuntu | :-)   | True           |
| 7dbb3917-7f2f-40e2-b1a1-09513fa22162 | L3 agent           | ubuntu | :-)   | True           |
+--------------------------------------+--------------------+--------+-------+----------------+
```

看到Open vSwitch agent没有正常启动，目前还在查找问题当中。

我们接下来的任务是继续搭建OpenStack平台并对其进行测试，直到这一环境可以完全正常运行，以保证我们进行后续有关Hadoop的安装、监测网络流量等工作不会被这个平台的错误影响。但到目前为止我们还未能成功地在我们搭建的OpenStack上通畅地运行实例。蒙民伟科技楼的实验室即将使用Campass完成Openstack在所有服务器上的部署，不久的将来我们将可以使用实验室的服务器对OpenStack展开进一步的研究。

五、有关计算中心集群调度算法的研究

在下半学期，除了对OpenStack的持续探究之外，基于调度算法这一切入点我们也展开了相关的研究。不同于虚拟网络环境中对虚拟机的调度，我们研究的对象主要是分布式系统中作业或任务级别的调度算法。我们探究了应用于研究领域或生产领域中的一些典型的分布式系统所采用的调度策略，并进行了比较与总结。

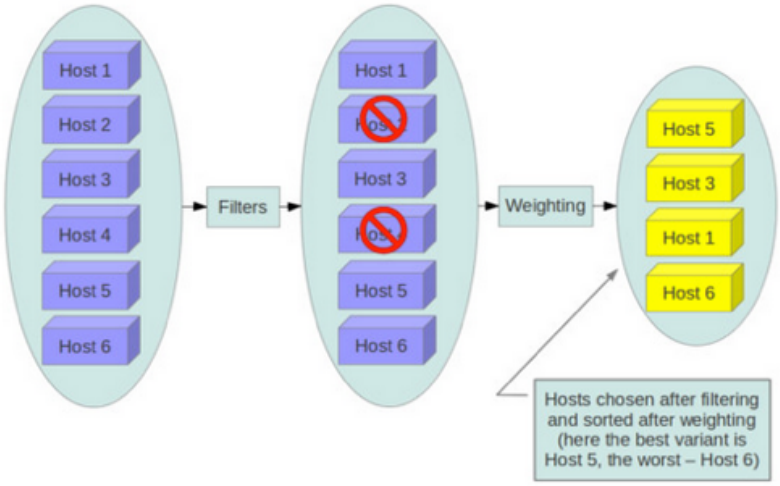
对调度算法的研究需求主要来自于分布式系统对多任务运行和资源复用的需求。好的调度算法能够提升数据中心的资源利用率，缩短大量数据计算分析任务的等待时间以及调度造成的时间延迟，以及有效地在多用户间实现公平合理地资源分配。

一般地，调度器锁面临的任务是：计算中心集群所运行的应用（如各种计算框架）会不断产生多个job，每个job包含一个或多个task，调度器需要做出决策完成对计算资源和task的匹配，即决定如何将资源分配给task。调度器有多种分类方式：根据调度器在系统架构中得位置，可以分为中心化调度器（如第一代Hadoop MapReduce的Job Tracker）和去中心化（分布式）调度器（如Mesos）；根据面向的系统状态可以分为面向系统当前状态（stateless）的调度器（如Sparrow）和面向过去全部状态的调度器；根据调度的对象资源可以分为面向服务器状态的调度器和面向网络状态的调度器等等。根据我们的研究，调度器所采用的调度算法多数本质上属于贪心算法，也有调度器采用其他类型的算法。以下分别进行论述。

【贪心算法】

1. Nova-scheduler

Nova-scheduler涉及的是虚拟机的调度，是我们OpenStack研究的一部分。调度器先通过用户指定的条件过滤掉不符合条件的host，对剩下的host，计算每个host的权值weight，选择权值最大的host放置虚拟机。目前nova中默认的scheduler的权值计算方式为weight=剩余RAM。

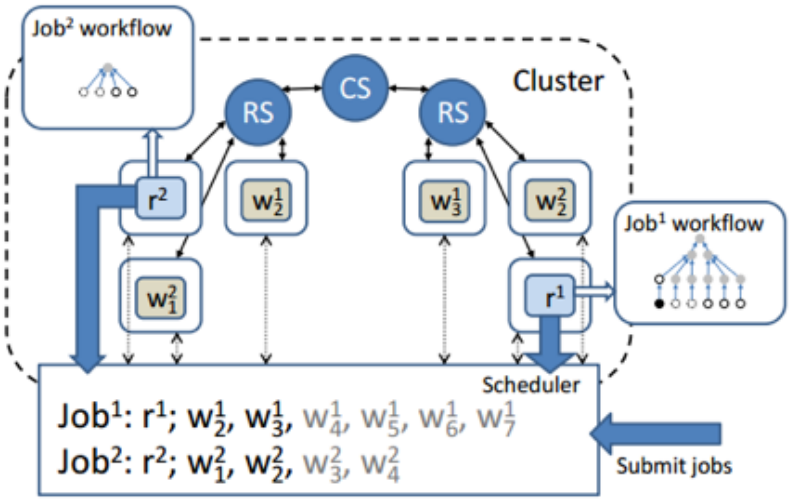


2. Queue-based scheduling

这是一类调度算法的集合，这类调度算法的一般模型是:

(1) 集群架构中，每个rack上的服务器由一个local switch连接，所有rack switch由一个core switch连接（好像跟我们机房是一样的），所以同一个rack上的computers之间通信代价要比不同rack间的通信代价低。

(2) 每个Job由一个被称为root task的process（每个Job有1个）管理这个Job的workflow，workflow是一个DAG，每个点包含一个task的状态（完成/运行中/可以开始运行），边代表task之间的依赖关系。由于运行一个task时，要获取跟这个task相关的数据，为减小数据传输代价，每个task会有一个computer preferred list。



2.1 算法总述

scheduler为每个computer、rack和整个集群各维护一个task的队列。当一个task被分配到一台computer时，将被从所有包含它的队列中移除。算法忽略传输数据大小，并将所有可用的computer视为一样的。当一个新的Job产生时，首先找一个空闲的机器运行这个Job的root task，若没有空闲机器，就找一台机器运行root task并把这台机器上正在运行的task踢掉，将这个task重新丢给scheduler调度。

2.2 最简单的贪心

当一个compute空闲时，选择该computer的task队列第一个ready的task，若不存在，从该computer所在的rack的task队列选择，若这个队列中也不存在ready task，就从整个cluster的队列中选择。

2.3 带公平性的贪心

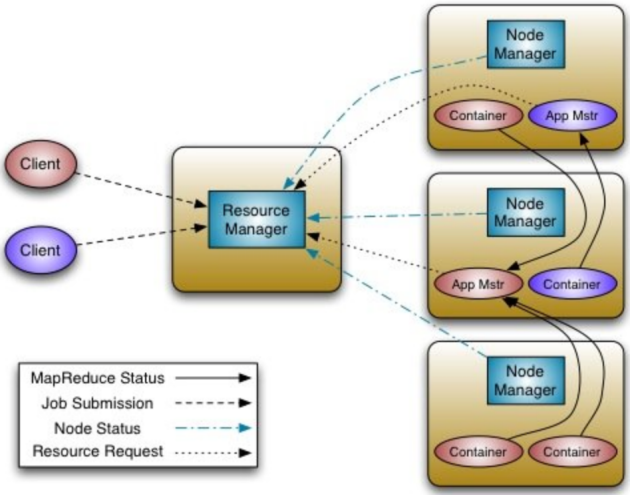
当一个computer空闲时，为每个Job j计算分配的computer数量 $A_j^* = \min(\lfloor M / K \rfloor, N_j)$，这里M是computer数量，K是在运行的Job个数，N_j是当前运行Job j的task，或有Job j的task在等待的computer个数。对于所有Job j，将A_j^*个computer分配后，若$\sum_j A_j^* < M$，即有computer剩余，则将剩余的computer均分给$A_j^* < N_j$的Job，分配后每个Job得到了A_j^*。对每个Job，若它占用的computer个数大于等于A_j^*，则将它block掉，不允许它的task被加入任何队列。

2.4 抢占式带公平性的贪心

当一个Job j用了大于A_j^*个worker，scheduler将它多运行的tasks全部kill掉，然后调度最近被调度的task。

3. 实际应用的分布式系统中贪心算法的应用

Hadoop YARN是Hadoop的新一代MapReduce框架[1]，主要是为了弥补第一代的MapReduce扩展性较差，不支持多计算框架的不足，并提供更高的可用性，提升集群的利用率。相比第一代MapReduce，YARN在架构上的一项重要变革是将JobTracker拆分成了两个独立的组件：resource mangement组件和job scheduling/monitoring组件。ResourceManager（在MapReduce 2.0中它就叫作Scheduler）负责调度工作，而针对于每个应用的ApplicationMaster只负责调度任务的管理和应用状态的监控。因此我们认为它采取的是中央调度器架构，而不是一些资料上所说的双层调度器架构。Google在他们的论文Omega:flexible, scalable schedulers for large compute cluster[2]中也有相关的论述支撑这一点。



YARN的Scheduler支持多层级的应用队列。YARN的调度器属于粗粒度的中心化的调度器。它提供的默认调度策略是FIFO，同时也支持其他调度策略比如CapacityScheduler和FairScheduler。

CapacityScheduler算法是基于队列的贪心算法。一般地，每项job被分配给一个队列，每个队列占据总计算资源的一部分。相同队列中得不同job共享同一部分计算资源。当某个task tracker上出现空闲slot时：

1. 调度器将所有队列按照资源使用率（numSlotsOccupied/capacity）的升序排序，从小到大依次按步骤2处理直至选定了队列

[1] http://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html

[2] Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]//Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

2. 在队列中将所有job按照提交时间和job优先级排序，调度器选择符合两个条件的job：job的用户使用的资源未达上限；当前空闲task tracker所在的节点内存足够该job的task使用。
3. 选择task并进行资源分配

FairScheduler算法想达到的核心目的是，采取一种公平的方式把资源分配给用户的各个job使得随着时间的累积，每个job分享到的资源是加权一致的。算法来自于论文Job Scheduling for Multi-User MapReduce Clusters[3]。FairScheduler用资源池来管理job，每个用户使用一个资源池，每个资源池分配一定份额的资源。当调度器在资源池内部分配slots时，FairScheduler采取max-min fairness或FIFO机制。每个资源池可以指定一个最小共享量。同时当某些资源池的资源有剩余时，调度器会将这些资源分配给其他需要的资源池以实现资源共享。每个资源池的最小共享量加上它获得的共享资源就等于它的公平共享量。基于这些参数调度器可以实现资源抢占功能，以杀死某些task来抢占其资源供给给某些需求达不到的资源池。另外，FairScheduler也能限制每个用户及每个资源池并行运行的资源数。另外论文中还采用了delay scheduling的策略来提升locality。它的核心思想是，当即将被分配某节点资源的task不具有该节点的本地性时，调度器将将它延后处理并处理队列后续的其他task，除非它已等待了一定长的时间。这个策略可以有效提升应用的locality因而提升应用的性能。

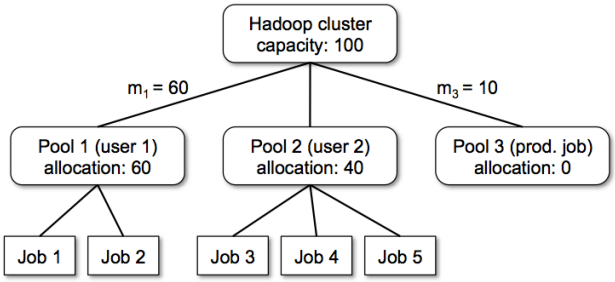


Figure 2: Example of allocations in our scheduler. Pools 1 and 3 have minimum shares of 60, and 10 slots, respectively. Because Pool 3 is not using its share, its slots are given to Pool 2.

相对于中央调度器，双层调度器（two-level scheduler）是另一类典型的调度器。它弥补了中央调度器可扩展性差和对不同计算框架采取的各种具体的调度策略的支持性差的不足。我们主要研究了双层调度器中具有代表性的Apache Mesos[4]。

Mesos是一个分布式的资源管理平台，它的主要设计目也是为了能够支持多种不同的计算框架（如Hadoop，MPI，Spark等等），使得它们能够共享系统集群的计算资源。Mesos采取一种名为resoruce offers的调度机制，即Mesos决定给每个计算框架分配多少资源，同时每个框架决定接受哪些资源以及运行哪些具体的计算任务。

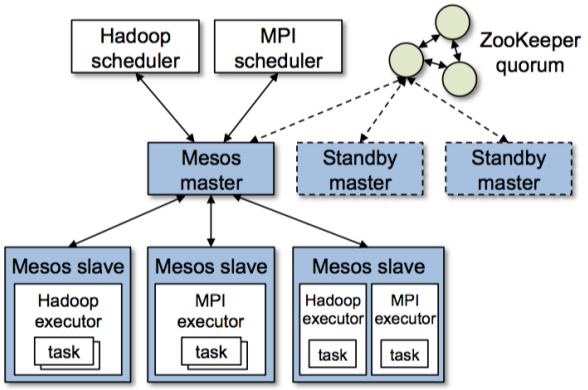


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

ıling for multi-user mapreduce clusters[J]. EECS ep. USB/EECS-2009-55, 2009.

[4] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center[C]//Proceedings of the 8th USENIX conference on Networked systems design and implementation. USENIX Association, 2011: 22-22.

Mesos的架构主要由Mesos master和Mesos slave两类节点构成。其中Mesos master担任第一层的调度任务，可以看做一个全局的资源调度器。每个运行在Mesos上的计算框架可以向Mesos master注册它们自己的调度器。当新的空闲资源产生时，它首先会经过Mesos master的第一次调度（分配给某个计算框架），其次会接受具体的计算框架的第二次调度。不同计算框架可以根据自身的需求采取多样化的调度策略，同时并不增加Mesos本身的设计复杂度。所以总体而言Mesos采取的是分布式的调度机制。

Mesos master的第一层调度所采用的机制可以采用DRF（Dominant Resource Fairness）[5]，这个算法是max-min fairness的一种扩展，关注于系统中含有多种资源类型（如CPU、内存等等），而共享资源的不同用户对每种资源有不同的需求时，如何进行公平地资源分配的问题。算法伪代码如下：

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
  $C = C + D_i$ ▷ update consumed vector
  $U_i = U_i + D_i$ ▷ update i's allocation vector
  $s_i = \max_{j=1}^{m} \{u_{i,j}/r_j\}$
else
  return ▷ the cluster is full
end if

它的核心思想是，不同用户可能倾向于需求不同类型的资源，比如用户A更关注内存资源，用户B更关注CPU资源，算法通过 $s_i = \max_{j=1}^{m} \{u_{i,j} / r_j\}$ 来计算每个用户的dominant share，并在每次调度时选择dominant share最小的用户给他分配资源来实现max-min，使每个用户的dominant share随时间的累积趋于一致。

Sparrow[6]是另一个值得一提的去中心化的调度器。Sparrow产生的背景是当今计算中心中大规模的计算集群所处理的数据量越来越大，而集群中运行的task有着越来越短小的趋势。这一方面是

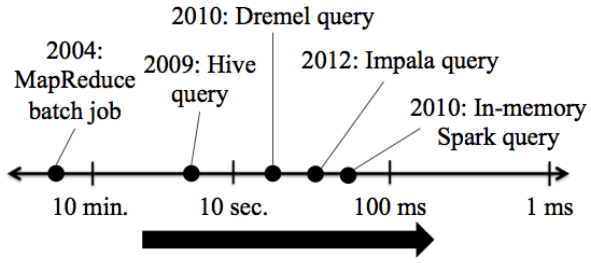


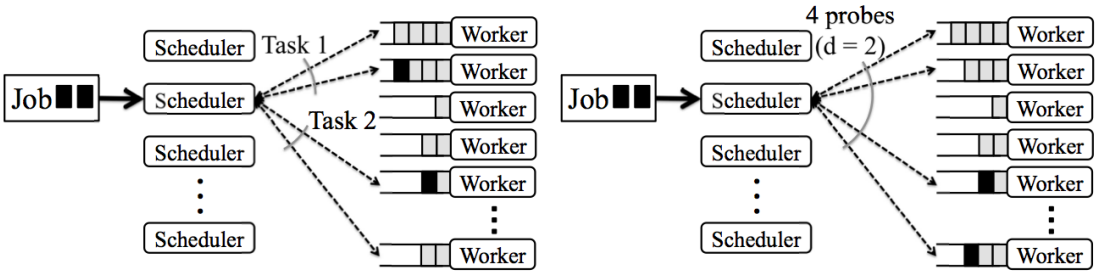
Figure 1: Data analytics frameworks can analyze large volumes of data with ever lower latency.

由于计算框架越来越追求计算任务的低延迟，另一方面是为了提升公平性和缓解掉队任务带来的负面影响。因此在计算任务时长已达亚秒级别的情形下，优化调度过程带来的延迟，将其控制在百毫

[5] Ghodsi A, Zaharia M, Hindman B, et al. Dominant resource fairness: fair allocation of multiple resource types[C]//USENIX NSDI. 2011.

[6] Ousterhout K, Wendell P, Zaharia M, et al. Sparrow: distributed, low latency scheduling[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 69-84.

秒级别上已成为问题的关键。Sparrow采取并行化的、去中心化的方式来进行调度，主要是因为由于调度任务的执行时间越来越短，调度的次数将变得非常多（每秒上百万次），如此高的调度频率将对中心化的调度器带来繁重的压力；另一方面，去中心化的调度器架构也可以提升系统的可扩展性，



(a) Per-task sampling selects queues of length 1 and 3. (b) Batch sampling selects queues of length 1 and 2.

Figure 2: Placing a parallel, two-task job. Batch sampling outperforms per-task sampling because tasks are placed in the least loaded of the entire *batch* of sampled queues.

系统可以通过增加调度器的数量来应对单位时间内更多地调度请求。

Sparrow主要基于The Power of Two Choices in Randomized Load Balancing[7]这篇论文的思想对其算法做了一些改进。 这篇论文讨论了如下情形：将n个小球的每一个独立地以等概率随机投入n个盒子中，则球数最多的盒子里的球数将较大概率地是近似 $\log n / \log \log n$ 。而每次扔小球时随机选取d个盒子并将球扔进它们中球数最少的一个，则上述的最大球数值将指数地衰减为 $\log \log n / \log d + O(1)$ 。这篇论文的核心思想就是在负载均衡问题中利用这样一个特性来设计算法，来获得较好的负载均衡的效果。Sparrow基于次核心思想，采用了三项技术进行了改进：Batch Sampling，Late Binding和Policies and Constraints。Batch Sampling是指调度器每次从队列中调度m个task，并从所有计算节点中随机选取m·d个（d > 1），并选取期中负载最小的（若无空闲资源，则选取等待队列长度最短的）m个节点，把这m个task分配给它们。Late Binding是指由于等待队列长度短并不代表队列中得task执行时间短，调度器采用的策略是它并不立即把task分配至计算节点上而是在对于的m·d个节点的队列中插入一个reservation，当它出队列时（有足够的空闲资源可以运行对应task时）计算节点通过RPC向调度器索取task。Polices and Constraints是指Sparrow在计算节点上维护多个队列，并采取一些机制来满足基于job的或者基于task的一些限制条件。这三项技术使得Sparrow的优化性能大大提升。相比于其他典型的调度器，Sparrow关注的是非常细粒度的调度，以及尽可能地缩短调度的延迟，因此它的调度算法设计相对简单，也不支持复杂的调度策略。

Omega[8]是谷歌公司下一代的集群管理系统。它使用的并行化的、共享状态的、乐观锁多并发的调度器，相比于谷歌公司过去采用的中心化的调度器和双层调度器而言，它具有更好地可扩展性和具体实现的可延展性。Omega使每个调度器都可以观测到整个集群的活动状态。Omega没有中心化的资源调度器，所有的资源分配决策都由分布式的调度器完成。每个调度器都维护一个波爱吃更新的包含集群全局状态信息sell state。每个调度器只要有合适的权限和优先级，它就可以随时地自由地向任何可用的集群资源发出请求。一旦一个调度器做出了资源分配的决策，它将进行一次原子提交操作来更新它共享的cell state的拷贝。在多个调度器的提交操作发生冲突时，最多一个操作会被成功执行。Omega的调度器是完全并行化运行的，所以不需要等待其他调度器对job的调度，也不会产生线头阻塞。调度器使用incremental transaction机制来防止冲突造成的饥饿现象，也可以采取all-or-nothing transaction来实现gang scheduling。调度器的性能主要受限于调度决策冲突发生的频率，但谷歌公司的真实负载测试表明，这种调度方式带来的冲突次数是可以接受的。

---

[7] Mitzenmacher M. The power of two choices in randomized load balancing[J]. Parallel and Distributed Systems, IEEE Transactions on, 2001, 12(10): 1094-1104.

[8] Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]//Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

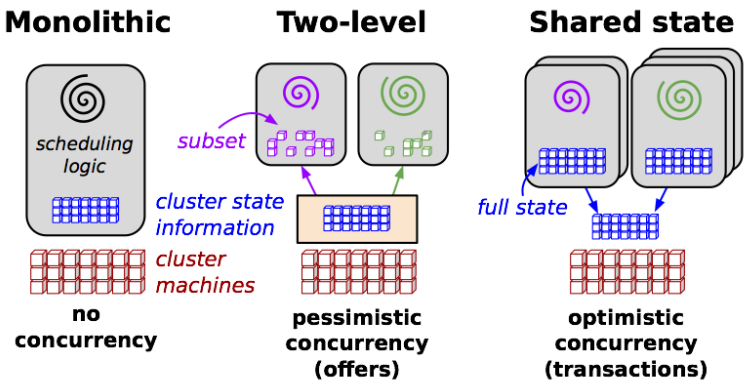


Figure 1: Schematic overview of the scheduling architectures explored in this paper.

【网络流算法】

Quincy[9]采用的调度算法是：建立一个流图，为每个Job(u_j)、每个Job的workflow中的每个节点(w_j^i)、每个root task(r^i)、rack(R_i)、computer(C_i)和整个集群(x)各建立一个对应的节点，S是汇点。从每个root task向运行它的computer连边；从每个worker task向它prefer的computer、rack和整个集群、它所属的Job连边；从每个运行中的worker task向运行它的computer连边。每条边有费用和最大流量，这里不做叙述。对这张图求最小费用流可以等价地给出平衡了数据本地化、公平性和避免饥饿的最优调度。

[9] Isard M, Prabhakaran V, Currey J, et al. Quincy: fair scheduling for distributed computing clusters[C]// Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 261-276.

参考论文：

Zaharia M, Borthakur D, Sarma J S, et al. Job scheduling for multi-user mapreduce clusters[J]. EECS Department, University of California, Berkeley, Tech. Rep. USB/EECS-2009-55, 2009.

Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]//Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

Ousterhout K, Wendell P, Zaharia M, et al. Sparrow: distributed, low latency scheduling[C]// Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 69-84.

Mitzenmacher M. The power of two choices in randomized load balancing[J]. Parallel and Distributed Systems, IEEE Transactions on, 2001, 12(10): 1094-1104.

Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center[C]//Proceedings of the 8th USENIX conference on Networked systems design and implementation. USENIX Association, 2011: 22-22.

Ghodsi A, Zaharia M, Hindman B, et al. Dominant resource fairness: fair allocation of multiple resource types[C]//USENIX NSDI. 2011.

Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]//Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

Isard M, Prabhakaran V, Currey J, et al. Quincy: fair scheduling for distributed computing clusters[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 261-276.