上海交通大学软件学院硕士论文开题综述

**基于微服务架构的智慧社区应用服务与资源管理系统**

**Application Service and Resource Management System for Smart Community based on Micro-service Architecture**

|  |  |
| --- | --- |
| 系 别 | **软件学院** |
| 学科专业 | **软件工程** |
| 研究方向 | **软件工程** |
| 姓 名 | **李会一** |
| 导 师 | **饶若楠** |

上海交通大学软件学院

2015年6月21日

# 研究背景及意义

智慧社区是指充分利用物联网、云计算、移动互联网等新一代信息技术的集成应用，为居民、居委会、业委会、物业和其他服务机构等社区相关人员提供的一种智慧化的生活环境和新的社区管理形态。随着智慧社区应用在社会进步与发展过程中作用的不断提升，其信息化运营与管理成本也在社会经济中产生了重要的影响。面对智慧社区中海量应用所造成的巨大的系统集成、运维成本，建立统一、高效的应用、资源管理方案是降低运营成本，提高服务质量的迫切需要。

本文以实验室承担的智慧邻里社区服务软件系统研发实际项目为背景，该系统建立了统一智慧社区应用平台，将广泛的社区应用，如居民生活服务、居委会工作服务、物业服务、党建服务、便民便利服务等进行整合，形成完整的社区服务闭环。同时为社区治理中的生活群众、政府管理群众、服务群众提供有价值的服务。例如，社区内的居民可以通过系统内邻里社交、物业、生活（外卖、洗衣、超市、家政等）、二手市场、社区公告等服务中获得生活便利；社区中政府管理群众可通过系统内党建、人事管理、社区公告、活动、邮件等服务中增加与居民沟通渠道，提升社区管理服务质量；社区内服务群众可通过系统内便民便利服务平台发布自身服务，增加自身营收。

然而， 在构建智慧社区治理系统过程中，遇到了以下的难题：1）应用服务间耦合严重，随着应用范围以及复杂度的增加，扩容以及更新流程复杂度大大增加；2）系统计算、存储、网络资源利用率低，没有一个很好的资源管理平台进行统一管理；3）缺乏统一的邻里社交模型，系统内用户交流存在壁垒。因此，如何建立统一、高校的应用、资源管理方案，成为一大问题。

# 研究目标及内容

本文以实验室承担的智慧邻里社区服务软件系统研发实际项目为背景，在对微服务系统架构、即时通讯系统、统一资源管理平台以及服务治理等相关技术进行深入分析研究的基础上，设计并实现了一个基于微服务架构的应用服务与资源管理系统。该系统在提供基于实名制的邻里社交系统的基础上，利用基于Dubbox[48]的服务治理系统以及基于Apache Yarn的资源管理系统两大关键系统的结合解决了应用服务耦合严重以及平台资源利用率低的问题。并通过测试及试运行应用，以验证该系统的可行性及有效性。

围绕上述的研究目标，本文的研究内容主要有：

1）实名制邻里社交系统：

a）分析并建立基于实名制的邻里社交模型，对社区中的管理、生活、服务群众的社交模式进行抽象；

b）建立即时通讯系统，支持点对点、群组等社交方式；

c）实现用户状态管理服务，支持对用户或外接设备在线状态的管理；

d）实现应用消息推送服务，支持聊天消息、应用广告等信息推送；

2）实现基于Apache Yarn的支持“长服务”的统一资源管理系统：

a）改进Yarn自身基于jobId的服务管理机制，通过服务发现注册机制扩展服务管理的功能及灵活性。解决由于节点迁移后导致的整体服务不可用的问题，并提供更多的协议支持；

b）改进Yarn对于服务日志的管理机制，通过重写日志管理器及客户端API，支持在服务运行过程中对日志的管理，并支持日志管理策略的配置；

c）实现Yarn中Container资源动态变更功能，使Yarn能够在不重启Container的情况下对Container资源进行变更，从而增加服务的可用性；

3）基于微服务架构的社区管理系统：

a）对智慧社区应用进行抽象划分，微服务化；

b）基于服务治理框架Dubbox，实现服务发现、注册、监控调用统计、容量评估等功能，对服务进行管理。

c）与统一资源管理系统集成，以服务监控信息为依据，按照一定策略对服务进行自动水平扩展。

4）系统验证及测试：通过系统测试及试运行应用，验证该系统的可行性及有效性。

# 相关技术综述

## 资源管理平台

在目前的复杂的企业解决方案中，所有的服务器按照业务逻辑被拆封成若干个集群，这些集群各自运行着相对独立的业务，并由专门的运维人员管理。

然而，由于集群中应用特点的不同，它们的资源利用率可能不一致。例如，同一时间段内有的集群业务繁忙，资源利用率高，甚至可能出现资源排队等待的情况，而另外一些集群则处于业务空闲，资源利用率极低的状态。此外，多个集群也可能导致数据冗余度增加，例如一个集群中的业务若需要另一集群中的数据，可能需要跨集群拷贝数据，进而占用更多的磁盘空间。因此，为了解决多集群所带来的问题，如Borg[4]（Google）， Yarn[1]（Apache，Hadoop项目下一个分支，开源），Mesos[2]（Twitter，开源），Corona[5]（Facebook，开源）这一类统一资源管理系统相继出现。概括起来，这类系统主要解决以下三类问题：

1）提高集群资源利用率

在大数据时代，为了存储和处理海量数据，需要规模较大的服务器集群或者数据中心，一般说来，这些集群上运行着数量众多类型纷杂的应用程序和服务，比如离线作业，流式作业，迭代式作业，crawler server，web server等，传统的做法是，每种类型的作业或者服务对应一个单独的集群，以避免相互干扰。这样，集群被分割成数量众多的小集群，一部分集群运行Hadoop[25]，一部分运行Storm[10]，一部分运行Spark[11]，一部分运行web server，然而，由于不同类型的作业/服务需要的资源量不同，因此，这些小集群的利用率通常很不均衡，有的集群满负荷、资源紧张，而另外一些则长时间闲置、资源利用率极低，为了提高资源整体利用率，一种解决方案是将这些小集群合并成一个大集群，让它们共享这个大集群的资源，并由一个资源统一调度系统进行资源管理和分配，这就诞生了Borg，YARN，Mesos，Torca，Corona。从集群共享角度看，这类系统实际上将所有硬件资源抽象成一个台大型计算机，供所有用户使用。

2）服务自动化部署

一旦将所有计算资源抽象成一个“大型计算机”后，就会产生一个问题：各种不同的服务如何进行部署？同样，Borg/YARN/Mesos/Torca/Corona一类系统需要具备服务自动化部署的功能，需要系统能够支持从简单的web server到诸如Spark，Storm等复杂的分布式计算框架的自动部署，管理。因此，从服务部署的角度看，这类系统实际上是服务统一管理系统，这类系统提供服务资源申请，服务自动化部署，服务容错等动能。

3）动态调整切分资源，增强系统扩展性

由于各类服务特征的不同，例如离线服务大都在夜间启动运行，对资源需求高，而传统应用服务则在日间活跃，导致系统对资源的规划很难一次性准确或稳定不变。因此，需要此类系统能够支持对已部署服务资源的动态调整与切分。增强系统的扩展性。

### Apache Yarn

Yarn是开源项目Apache Hadoop的一个资源管理系统，最初设计是为了解决Hadoop中MapReduce[18]计算框架的资源管理问题，但经过不断演化，它已经是一个更加通用的资源管理系统，可将MapReduce计算框架作为一个应用程序运行在Yarn系统之上，通过Yarn来管理资源。如果你的应用程序也需要借助Yarn的资源管理功能，你也可以实现Yarn提供的编程API，将你的应用程序运行于Yarn之上，将资源的分配与回收统一交给Yarn去管理，从而大大简化资源管理功能的开发。目前，已有诸如Storm，Spark，Tez，HBase[9]分布式计算框架或存储系统成功运行与Yarn之上。Yarn生态圈如图1-1所示：

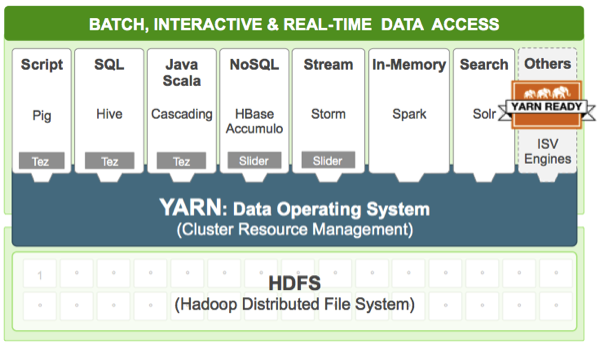


图 1‑1 Yarn生态圈环境[49]

Figure 1‑1 Ecosystem for Yarn related systems[49]

Yarn是基于Master/Slave模式的分布式架构，具体架构图如图1-2所示：

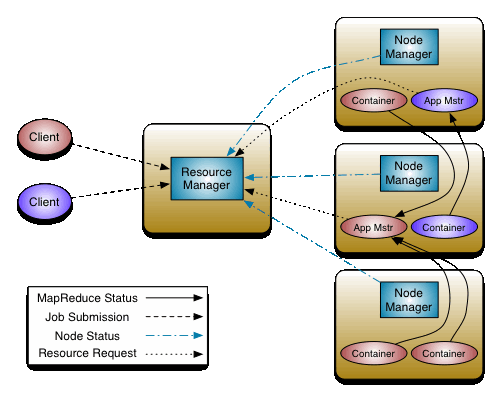


图 1‑2 Yarn架构图[50]

Figure 1‑2 Architecture of Yarn[50]

上图，从逻辑上定义了Yarn系统的核心组件和主要交互流程，各个组件说明如下：

1）Yarn Client：

Yarn Client提交Application给RM，它会首先创建一个Application上下文对象，并设置AM必须的资源请求信息，然后提交给RM。Yarn Client可以与RM通信，从而获得一个已经提交并运行的Application的状态信息。

2）Resource Manager（RM）：

RM是Yarn集群的Master角色，负责管理整个集群的资源，并对Application进行资源分配。RM作为集群资源的管理和调度角色，如果出现单点故障的情况，则整个集群的资源都无法使用。（这一问题已由2.4.0版本中新增的RM HA方案解决）

3）Node Manager（NM）：

NM是Yarn集群中的Slave角色，是集群中拥有实际资源的工作节点。我们提交Application之后，会将组成Application的多个Component调度到对应的NM上执行。

4）Container

Container是Yarn集群中资源的抽象，将NM上的资源进行量化，根据需要组装成一个个Container，然后服务于已授权资源的计算任务。计算任务在完成计算后，系统会回收资源，以供后续计算任务申请使用。目前，Container包含两种资源：内存和CPU，后续版本中会陆续增加硬盘、网络等更多的资源选项。

5）Application Master（AM）

AM主要管理和监控部署在Yarn集群上的Application，以MapReduce为例，MapReduce Application是一个用来处理MapReduce计算的服务框架程序，为用户编写的MapReduce程序提供运行时支持。通常我们在编写一个MapReduce程序中可能包含多个Map Task和Reduce Task，而各个Task的运行管理与监控都是由这个MapReduce Application来负责，比如运行Task的资源申请，由AM向RM申请；启动/停止NM上某Task的对应的Container，都有AM向NM请求来完成。

在对Yarn进行全面的分析之后，Yarn目前善于处理不需要与外部系统或用户交互的批量任务。它接收一个任务，并决定诸如Mapper、Reducer之类的模块在集群中何处运行，并将这些模块部署到相应的节点，令其启动，并完成其所被安排的任务。但当诸如Hbase, Storm, Spark等长时间运行的分布式计算框架出现时，Yarn难以告诉外部系统这些服务存在于哪里，以及怎样与这些计算框架进行交互。 除此之外，当Yarn中一个节点失效之后，Yarn所做的只是简单的将那个节点上所存在的模块调度到另一个节点之上并再次启动。这样的解决方案在处理map-reduce模型的任务时是可行的，因为所有map-reduce任务的模块都是无状态的，并且不需要直接与模型中其他模块进行通讯。但Storm这一类分布式计算框架出现时，问题变得不再那么简单。当Storm中Nimbus模块失效后，Yarn会将该模块调度到另一节点。然而由于Yarn是随机重调度失效节点的，Storm中所有需要与Nimbus节点进行交互的节点将找不到该节点，从而导致整个Storm的不可用。在Yarn中运行长服务的另一个问题在于日志管理，长服务通常都会随着运行时间的增长而产生大量的日志，而这些日志信息通常对于调试系统问题，以及记录系统状态起着重要的作用。因此Yarn需要提供一种有效的汇聚这一类日志以及在日志不再被需要时快速删除的解决方案。最后，Yarn的容器资源可伸缩性对于运行长服务同样重要。试想这样一种场景，Yarn上运行着一个tomcat的集群，但当整个集群的吞吐量上升时，需要动态提高这些服务所占用的资源以提高集群的吞吐性。

### Apache Mesos

Mesos是Apache下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos最初是由加州大学伯克利分校的AMPLab开发的，后在Twitter、Airbnb、Netflix等公司得到广泛使用。

**Mesos架构：**

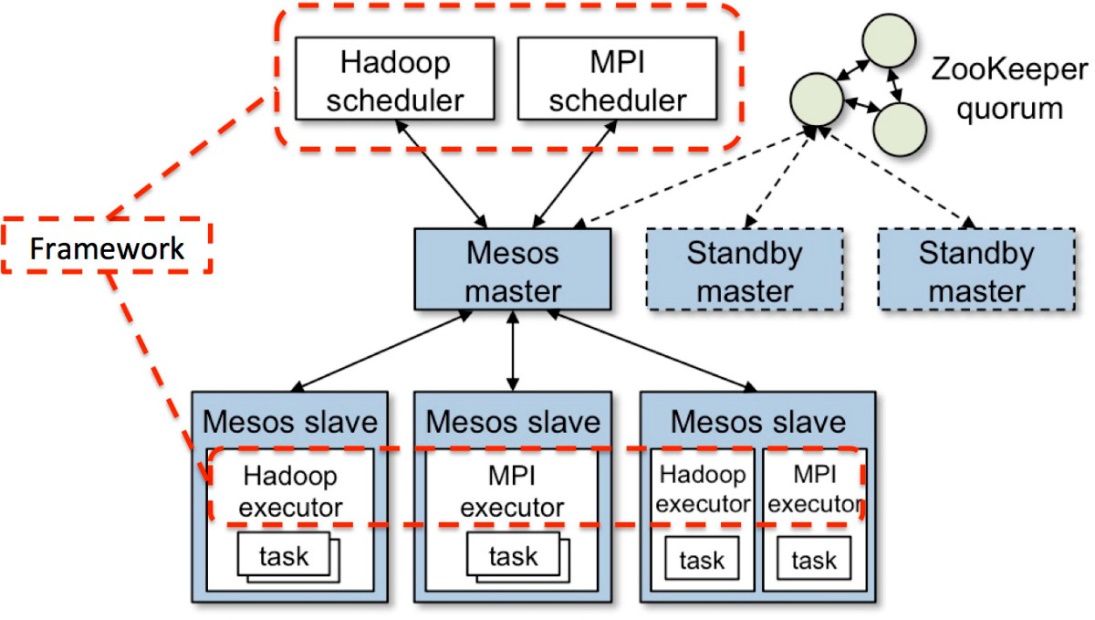


图 1‑3 Mesos架构图[51]

Figure 1‑3 Architecture of Apache Mesos[51]

如图1-3所示，Mesos实现了两级调度架构，它可以管理多种类型的应用程序，第一级调度是Master的守护进程，管理Mesos集群中所有节点上运行的Slave守护进程。集群由物理服务器或虚拟服务器组成，用于运行应用程序的任务，比如Hadoop和MPI作业。第二级调度由被称作Framework的“组件”组成。Framework包括调度器（Scheduler）和执行器（Executor）进程，其中每个节点上都会运行执行器。Mesos能和不同类型的Framework通信，每种Framework由相应的应用集群管理。上图仅展示了Hadoop和MPI[45]两种Framework。

Mesos Master协调全部的Slave，并确定每个节点的可用资源，聚合计算跨节点的所有可用资源的报告，然后向注册到Master的Framework（作为Master的客户端）发出资源邀约。Framework可以根据应用程序的需求，选择接受或拒绝来自master的资源邀约。一旦接受邀约，Master即协调Framework和Slave，调度参与节点上任务，并在容器中执行，以使多种类型的任务，比如Hadoop和Cassandra，可以在同一个节点上同时运行。

**Mesos资源分配**：

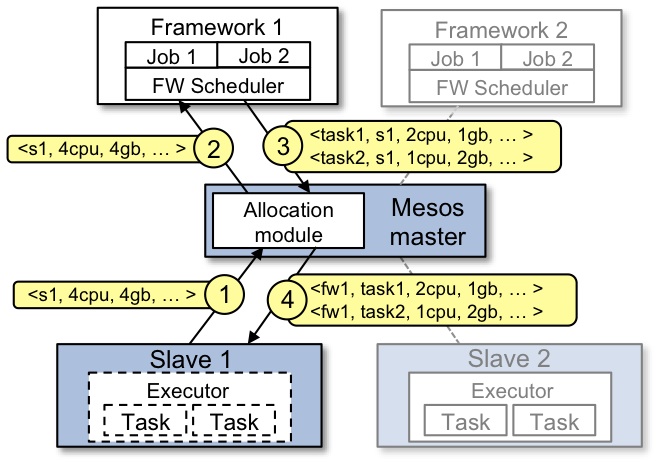


图 1‑4 Mesos资源分配流程[52]

Figure 1‑4 Resource allocation process of Mesos[52]

如图1-4所示，为了实现在同一组Slave节点集合上运行多任务这一目标，Mesos使用了隔离模块，该模块使用了一些应用和进程隔离机制来运行这些任务。虽然可以使用虚拟机隔离实现隔离模块，但是Mesos当前模块支持的是容器隔离。Mesos早在2009年就用上了Linux的容器技术[46]，如cgroups和Solaris Zone，时至今日这些仍然是默认的。然而，Mesos社区增加了Docker作为运行任务的隔离机制。不管使用哪种隔离模块，为运行特定应用程序的任务，都需要将执行器全部打包，并在已经为该任务分配资源的Slave服务器上启动。当任务执行完毕后，容器会被“销毁”，资源会被释放，以便可以执行其他任务。

Mesos资源邀约和分配策略，对管理跨多个Framework和应用的资源，是不可或缺的。 资源邀约：即由Master向注册其上的Framework发送资源邀约。每次资源邀约包含一份Slave节点上可用的CPU、RAM等资源的列表。Master提供这些资源给它的Framework，是基于分配策略的。分配策略对所有的Framework普遍适用，同时适用于特定的Framework。 Framework可以拒绝资源邀约，如果它不满足要求，若此，资源邀约随即可以发给其他Framework。由Mesos管理的应用程序通常运行短周期的任务，因此这样可以快速释放资源，缓解Framework的资源饥饿； Slave定期向Master报告其可用资源，以便Master能够不断产生新的资源邀约。 另外，还可以使用诸如此类的技术， 每个Framework过滤不满足要求的资源邀约、Master主动废除给定周期内一直没有被接受的邀约。

### Google Omega

Google的第一代/第二代集群（资源）管理系统被称为Borg，而于2013年，Google公布了它的下一代集群管理系统Google Omega[3]的设计细节。Omega论文中提出Google经历了三代资源管理系统，如图2-5所示，这三代系统分别如下：

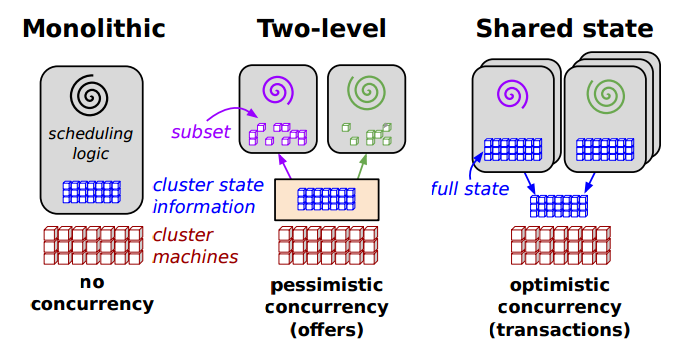


图 2‑5 三代资源调度系统概述

Figure 2‑5 Overview of resource management system

1）中央调度器（Monolithic scheduler）

中央式调度器的特点是，资源的调度和作业的管理功能全部放到一个进程中完成，开源界典型的代表是Hadoop JobTracker的实现。这种设计方式的缺点很明显，扩展性差：首先，集群规模受限，其次，新的调度策略难以融入现有代码中，比如之前仅支持MapReduce作业，现在要支持流式作业，而将流式作业的调度策略嵌入到中央式调度器中是一项很难的工作。

Omega论文中提到了一种对中央式调度器的优化方案：将每种调度策略放到单独一个路径（模块）中，不同的作业由不同的调度策略进行调度。这种方案在作业量和集群规模比较小时，能大大缩短作业相应时间，但由于所有调度策略仍在一个集中式的组件中，整个系统扩展性没有变得更好。

2）双层调度器（Two-level scheduler）

为了解决中央式调度器的不足，双层调度器是一种很容易想到的解决之道（实际上是分而治之策略或者是策略下放机制）。双层调度器仍保留一个经简化的中央式调度器，但调度策略下放到各个应用程序调度器完成。这种调度器的典型代表是Apache Mesos和Hadoop YARN。

双层调度器的特点是，各个框架调度器并不知道整个集群资源使用情况，只是被动的接收资源；Mesos Master仅将可用的资源推送给各个框架，而框架自己选择使用还是拒绝这些资源；一旦框架（比如Hadoop JobTracker）接收到新资源后，再进一步将资源分配给其内部的各个应用程序（各个MapReduce作业），进而实现双层调度。

双层调度器的缺点是：

A）各个框架无法知道整个集群的实时资源使用情况：

很多框架不需要知道整个集群的实时资源使用情况就可以运行的很顺畅，但是对于其他一些应用，为之提供实时资源使用情况可以为之提供潜在的优化空间，比如，当集群非常繁忙时，一个服务失败了，是选择换一个节点重新运行它呢，还是继续在这个节点上运行？通常而言，换一个节点可能会更有利，但是，如果此时集群非常繁忙，所有节点只剩下小于5GB的内存，而这个服务需要10GB内存，那么换一个节点可能意味着长时间等待资源释放，而这个等待时间是无法确定的。

B）采用悲观锁，并发粒度小：

在数据库领域，悲观锁与乐观锁争论一直不休，悲观锁通常采用锁机制控制并发，这会大大降低性能，而乐观锁则采用多版本并发控制(MVCC ,Multi-Version Concurrency Control)，典型代表是MySQL innoDB，这种机制通过多版本方式控制并发，可大大提升性能。在Mesos中，在任意一个时刻，Mesos资源调度器只会将所有资源推送给任意一个框架，等到该框架返回资源使用情况后，才能够将资源推动给其他框架，因此，Mesos资源调度器中实际上有一个全局锁，这大大限制了系统并发性。

3）共享状态调度器（Shared state scheduler）

为了克服双层调度器的以上两个缺点，Google开发了下一代资源管理系统Omega，Omega是一种基于共享状态的调度器，该调度器将双层调度器中的集中式资源调度模块简化成了一些持久化的共享数据（状态）和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。一旦引入共享数据后，共享数据的并发访问方式就成为该系统设计的核心，而Omega则采用了传统数据库中基于多版本的并发访问控制方式（也称为“乐观锁”, MVCC, Multi-Version Concurrency Control），这大大提升了Omega的并发性。

由于Omega不再有集中式的调度模块，因此，不能像Mesos或者YARN那样，在一个统一模块中完成以下功能：对整个集群中的所有资源分组，限制每类应用程序的资源使用量，限制每个用户的资源使用量等，这些全部由各个应用程序调度器自我管理和控制，根据论文所述，Omega只是将优先级这一限制放到了共享数据的验证代码中，即当同时由多个应用程序申请同一份资源时，优先级最高的那个应用程序将获得该资源，其他资源限制全部下放到各个子调度器。

引入多版本并发控制后，限制该机制性能的一个因素是资源访问冲突的次数，冲突次数越多，系统性能下降的越快，而google通过实际负载测试证明，这种方式的冲突次数是完全可以接受的。

## 服务治理

服务治理是分布式系统及面向服务系统架构的核心组成。客户端怎样知道存在于多台主机上的服务的IP以及端口即为服务治理在最初所面临的问题。然而，随着系统中服务的不断增多，在自动或手动扩容、主机失效、新服务部署的情况下，服务的地址变化将变得极其频繁。此时，为了避免服务终端，动态服务注册与发现重要性将大大增加。

### 服务注册与发现

在定位服务地址的问题上，有两个关键性问题：

1）服务注册：服务将自身地址注册到一个中央注册中心当中。通常它将注册自己的IP及端口，验证凭据、使用的协议、版本号或一些使用环境细节信息。

a）自注册模式：

自注册模式意味着服务实例对自身的注册以及注销负责。同时，在必要的情况下需要发送心跳信息至服务中心防止其注册信息过期。图1-6展示了该模式的架构：

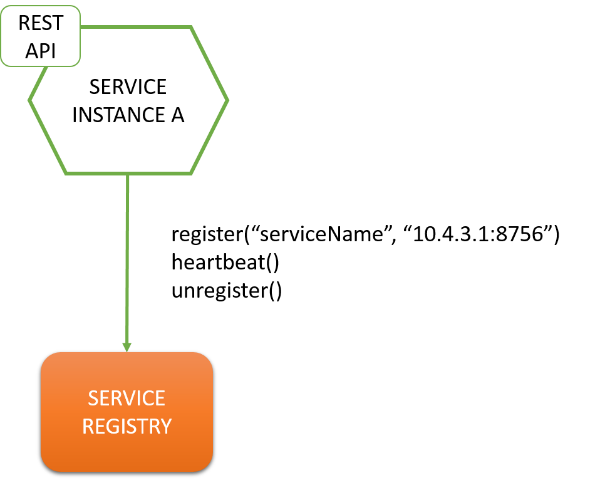


图 1‑6 自注册模式

Figure 1‑6 Self-registration pattern

自注册服务拥有许多的优势与缺陷，其中一个优势在于它的相对简单性，且不依赖于其他的系统组件。然而，其中一个主要的缺陷在于它导致服务实例与服务中心耦合。必须针对每种变成语言或框架实现特定的代码。

b）第三方注册模式：

使用第三方注册模式将服务实例与注册中心解耦，服务实例不再负责自身的注册与注销。反之，一个称为服务注册器的系统组件被用以处理服务注册。服务注册器通过拉取运行信息或订阅时间的方式跟踪一组运行实例，当它发现一个新的可用服务时就将该服务注册至服务中心。与此同时，服务注册器也会处理服务注销的问题。第三方注册模式如图1-7所示：

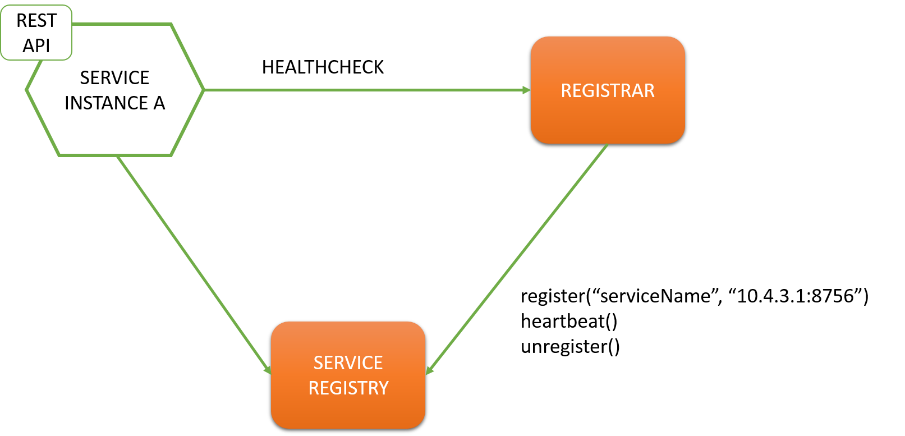


图 1‑7 第三方注册模式

Figure1‑7 Third-party registration pattern

服务发现：客户端程序查询中央注册中心获取服务地址。

a）客户端服务发现模式：

当使用客户端服务发现模式时，客户端负责在可用服务实例的网络地址中进行选择，并在其中进行负载均衡决策。客户端通过查询服务注册中心获取服务信息，并使用一个负载均衡算法来选择可用服务并发送请求。

客户端服务发现模式有自己的优势与缺陷。这个模式相对简单，并且除了注册中心之外，没有其他的系统部件需要移动。同时，由于客户端知道可用的服务实例地址，它能做出诸如全局一致性hash等智能的负载均衡决策。而客户端服务发现模式的一个重大的缺陷在于注册中心与客户端耦合严重，对于每个变成语言或框架都需要实现特定的服务发现注册逻辑。

图1-8详细描述了该模式的架构：

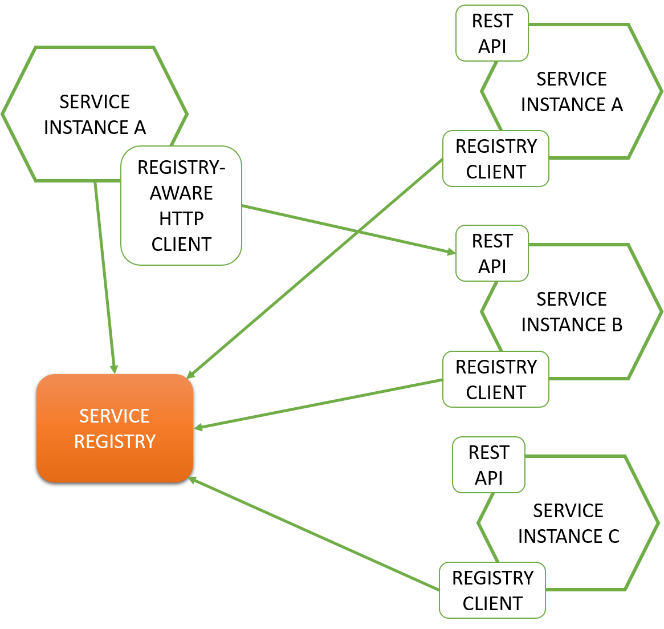


图 2‑8 客户端服务发现模式

Figure 2‑8 Client-side service discovery pattern

b）服务器端服务发现模式：

服务器端服务发现模式如图2-9所示：

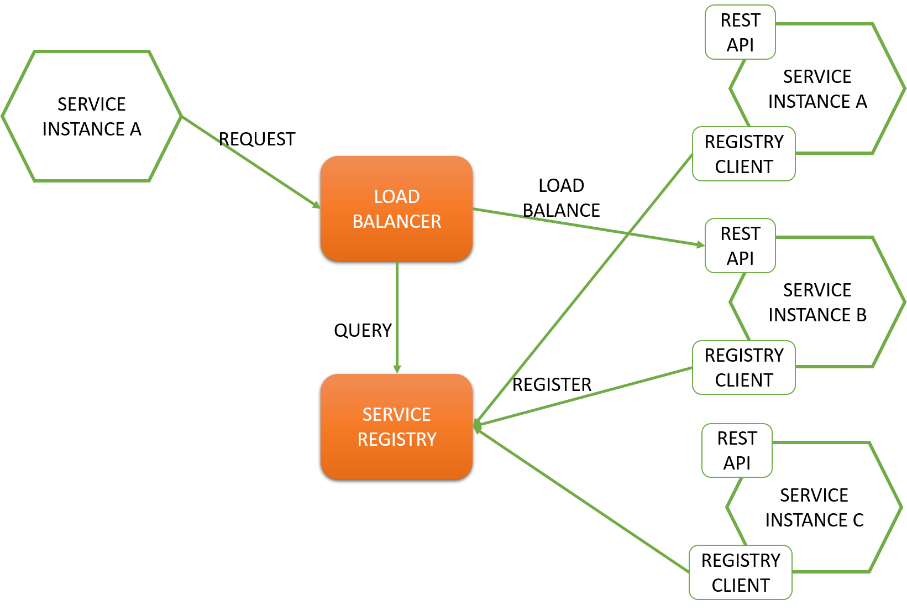


图 2‑9 服务器端服务发现模式

Figure 2‑9 Server-side service discovery pattern

客户端发送服务请求到一个负载均衡器。负载均衡器查询服务注册中心并将每个请求路由到一个可用的服务实例上。与客户端服务发现模式相同，服务实例通过服务注册中心注册与注销。

AWS Elastic Load Balancer（ELB）是服务器端服务发现模式的一个例子。一个ELB通常被用于对外部Internet流量进行负载均衡处理。HTTP服务器如Nginx也可用做服务器端服务发现负载均衡器。而例如Kubernetes[47]和Marathon一类的部署环境在集群中每台主机上运行了一代理。这个代理担任着服务器端服务发现负载均衡器的职责。

### 服务监控与调度

在解决上述两大关键性问题的基础上，通常还需要考虑以下的监控与调度方面的问题：

1）监控：当一个注册的服务失效后应该怎样处理？立即注销、一段时间后注销、利用另一守护进程注销都是可行的解决方案。通常服务被要求实现一个心跳机制来确保自身的可用性，而将处理失效服务的责任交给客户端处理。

2）负载均衡：若一个服务注册了多个实例，客户端怎样在多个实例中平衡分配流量。若服务拥有主服务，客户端怎样发现这个主服务？

3）集成因素：注册中心支持哪些语言的注册绑定？集成需要内嵌的注册发现库还是使用外部进程来实现？

4）运行时依赖：服务需要什么样的运行环境？

5）可用性：在节点失效后服务是否仍然可用？是否可在运行时升级且不会导致服务中断？注册中心变成系统中核心部分后怎样解决其单点失效的问题？

### 国内外产品对比

目前国内外已经存在诸多的用于服务发现、注册的基础服务，表2-1对它们进行了详细的对比：

表格 2‑1 服务发现、注册产品比较

Table 2-1 Comparison between service registration & discovery products

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 类型 | 一致性 | 语言 | 依赖 | 集成 |
| Zookeeper | 通用 | 强一致 | Java | JVM | 客户端集成 |
| Doozer | 通用 | 强一致 | Go |  | 客户端集成 |
| Etcd | 通用 | 混合 | Go |  | 客户端集成/HTTP |
| SmartStack | 专用 | 最终一致 | Ruby | Haproxy、Zookeepr | Sidekick |
| Eureka | 专用 | 最终一致 | Java | JVM | Java Client |
| NSQ | 专用 | 最终一致 | Go |  | 客户端集成 |
| Serf | 专用 | 最终一致 | Go |  | 本地命令行 |
| Spotify | 专用 | 最终一致 | N/A | Bind | DNS库 |
| SkyDNS | 专用 | 混合 | Go |  | HTTP/DNS库 |

## 微服务软件架构风格

微服务是一种软件架构风格，它代表着一种开发多个运行在独立进程、环境中的使用轻量级通信协议进行交互的小型服务来代替传统的单一应用的大型应用的软件架构风格。这些微服务是围绕着业务功能进行划分的，并且可独立部署、升级、管理。

### 微服务架构的优势

为了更好的解释微服务架构，我们将其与传统的单一应用架构进行比较。企业级应用通常包括三个部分：1）客户端接口（通常包括运行在客户浏览器中的HTML页面以及javascript代码），2）数据库（通常为关系型数据库中的多个表集合），3）服务器端应用（处理HTTP请求，执行业务逻辑，获取更新数据）。而服务器端应用通常是一个逻辑上的集合，对于服务器端的任何修改都会导致对整个服务器端应用的重新编译以及部署。单一应用架构是可行的，但随着应用范围以及复杂度的增加，其缺点也显得越来越明显：每次更改都要求整个服务器端重新编译以及部署，在扩容时也只能够对整个应用进行扩容，而不能够只针对资源需求更低的一个模块来进行扩容。微服务架构如图1-10所示

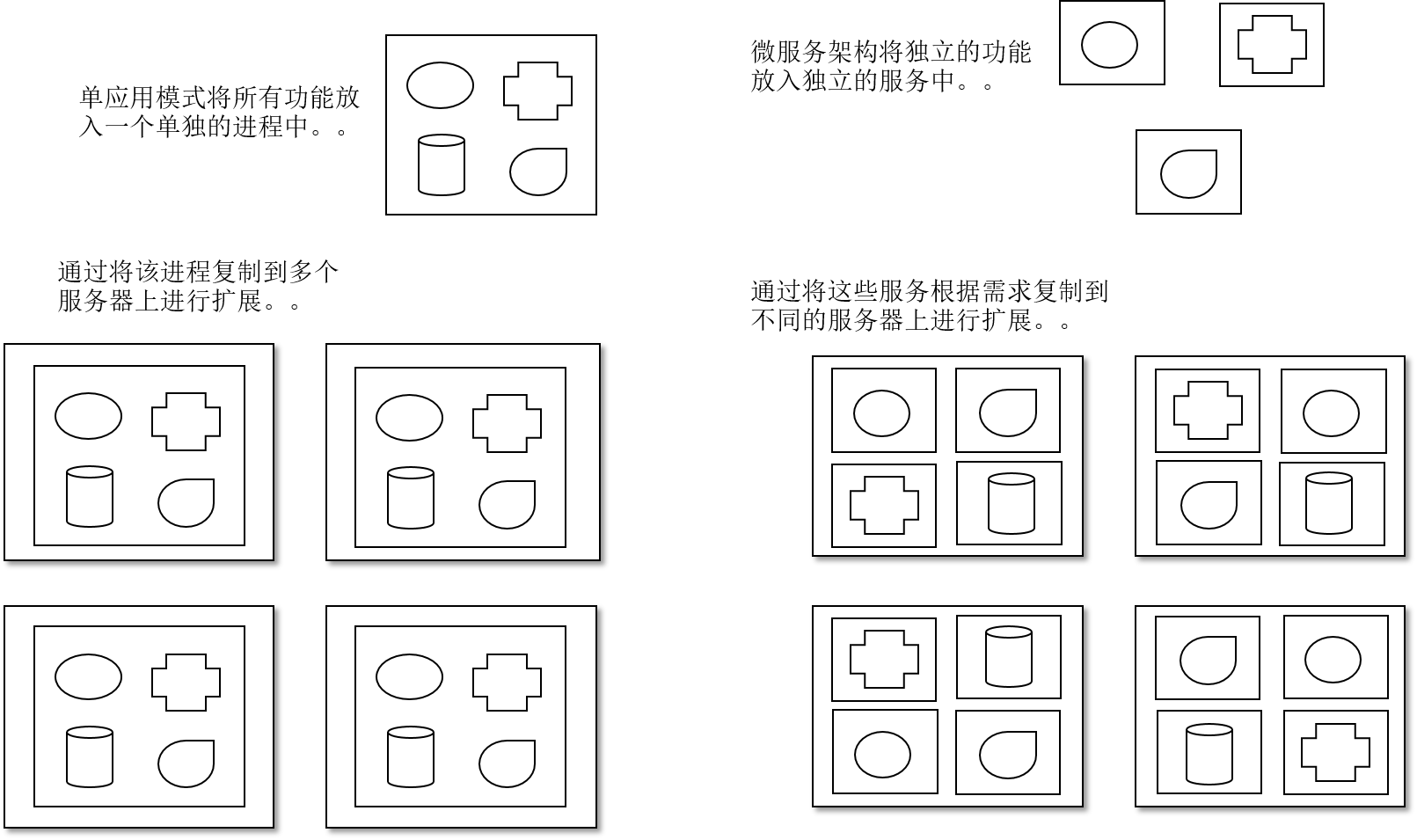


图 1‑10 微服务架构与垂直架构对比

Figure1‑10 Comparison between micro-service and monolithic architecture

上述的问题就引出了一个新的软件架构：微服务架构。微服务架构将传统的单一应用以一组独立的服务来构成，与此同时，这些服务是可以单独部署以及水平扩展的。每一个这样的服务都拥有着严格的模块边界，甚至允许不同的服务使用不同的开发语言来进行开发。

微服务架构会使用依赖包，但其主要通过将软件分割成服务来进行模块化。我们定义依赖包为链接入程序并通过in-memory程序调用使用的程序模块，而服务为使用RPC或者网络请求进行交互的进程。使用服务作为系统模块有两大优势：首先服务可独立部署，这样对一个单独的模块进行更改不会影响到系统中的其他模块，另一个优势则是服务间通过显示的远程调用来进行通讯，从而减小程序间的接口定义难度。

### 微服务架构的缺陷

然而，微服务架构也是有缺点的。首先，远程调用比进程内函数调用开销更大，其次，将程序职责在服务间进行迁移的难度也会更加的复杂。

去中心化治理：中心化治理的会导致整个程序开发使用单一的技术和平台。然而，这样的解决方案是受到很大限制的，不是每个问题都能通过单一的技术来解决。针对具体的问题选择最适合的技术显然是一个更加的做法。微服务架构则将这样的方法带入了软件开发之中。

去中心化数据管理：在对概念模型以及业务逻辑进行去中心化的同时，微服务架构同样对数据进行了去中心化的管理。每个微服务都独自管理自己的数据，如图2-11所示，这意味着微服务可根据具体的业务需求选择合适的数据库（关系型或非关系型），但同时也意味着在服务间数据一致性问题会更加难以处理。

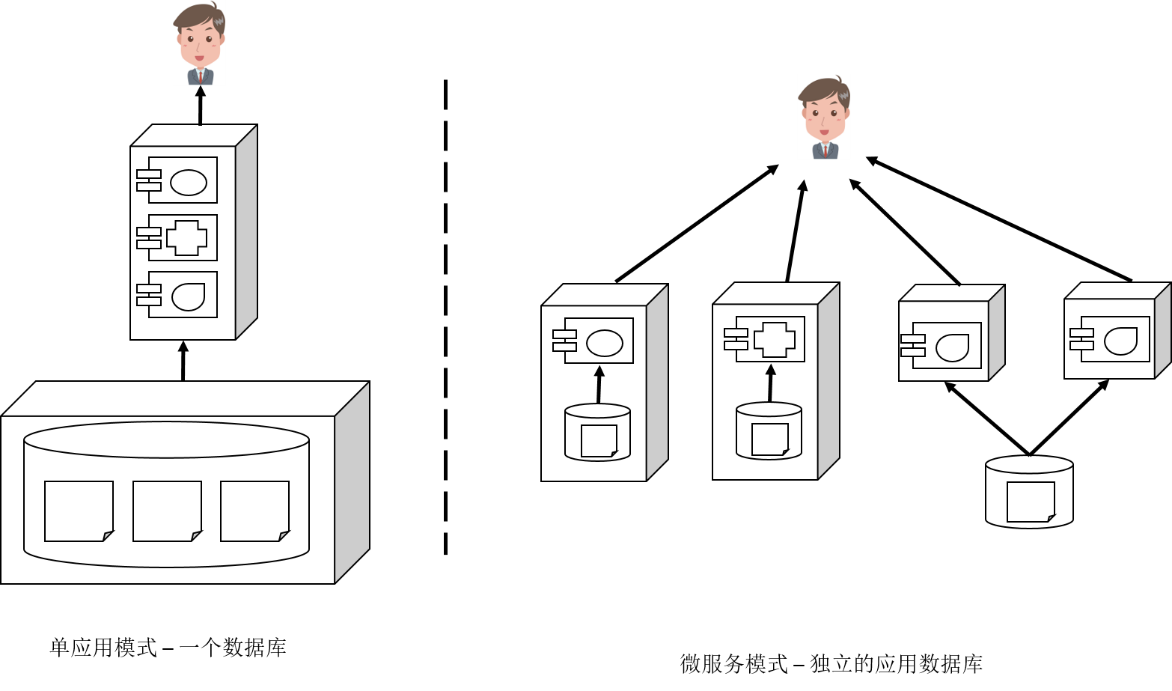


图 2‑11 微服务架构数据管理

Figure 2‑11 Data management for micro-service architecture

测试部署自动化：随着云平台的日渐成熟，测试部署自动化在近年中发展迅速。而云平台大大减少了构建、部署、运行微服务的复杂度。

预防软件错误：微服务架构需要在设计时将服务异常纳入考虑范围，由于任何服务调用都可能因为服务提供者失效或网络失联而失败，因此客户端需要很好地处理这一类情况。

迭代式设计：当尝试将一个系统服务化时，我们将面临一个至关重要的问题：怎样将系统划分成不同的服务？应该使用怎样的原则来进行划分？由于一个服务的关键要素是独立部署升级，因此在系统划分时应该考虑在其需要重构时不会影响到其他的协作者。

# 技术路线

## 关键问题

本课题研究的关键问题包括：

1. 邻里社交子系统：

分析并建立基于实名制的邻里社交模型，对社区中的管理、生活、服务群众的社交模式进行抽象；建立即时通讯系统，支持点对点、群组等社交方式；实现用户状态管理服务，支持对用户或外接设备在线状态的管理；实现应用消息推送服务，支持聊天消息、应用广告等信息推送。

2. 基于微服务的应用服务管理子系统

对智慧社区应用进行抽象划分，微服务化；基于服务治理框架Dubbox，实现服务发现、注册、监控调用统计、容量评估等功能，对服务进行管理；与统一资源管理系统集成，以服务监控信息为依据，按照一定策略对服务进行自动水平扩展。

3. 支持长服务的资源管理子系统：

改进Yarn自身基于jobId的服务管理机制，通过服务发现注册机制扩展服务管理的功能及灵活性。解决由于节点迁移后导致的整体服务不可用的问题，并提供更多的协议支持；改进Yarn对于服务日志的管理机制，通过重写日志管理器及客户端API，支持在服务运行过程中对日志的管理，并支持日志管理策略的配置；实现Yarn中Container资源动态变更功能，使Yarn能够在不重启Container的情况下对Container资源进行变更，从而增加服务的可用性；

## 技术路线

本研究将分为“即时聊天系统建立”，“便民便利服务系统建立”，“安全体系的建立”，“运行与验证”四个部分实现：

1）实名制邻里社交系统：

a) 分析并建立基于实名制的邻里社交模型，对社区中的管理、生活、服务群众的社交模式进行抽象；

b) 建立即时通讯系统，支持点对点、群组等社交方式；

c) 实现用户状态管理服务，支持对用户或外接设备在线状态的管理；

d) 实现应用消息推送服务，支持聊天消息、应用广告等信息推送；

2) 实现基于Apache Yarn的支持“长服务”的统一资源管理系统：

a) 改进Yarn自身基于jobId的服务管理机制，通过服务发现注册机制扩展服务管理的功能及灵活性。解决由于节点迁移后导致的整体服务不可用的问题，并提供更多的协议支持；

b) 改进Yarn对于服务日志的管理机制，通过重写日志管理器及客户端API，支持在服务运行过程中对日志的管理，并支持日志管理策略的配置；

c) 实现Yarn中Container资源动态变更功能，使Yarn能够在不重启Container的情况下对Container资源进行变更，从而增加服务的可用性；

3) 基于微服务架构的社区管理系统：

a) 对智慧社区应用进行抽象划分，微服务化；

b) 基于服务治理框架Dubbox，实现服务发现、注册、监控调用统计、容量评估等功能，对服务进行管理。

c) 与统一资源管理系统集成，以服务监控信息为依据，按照一定策略对服务进行自动水平扩展。

4）运行与验证：搭建整个社区治理服务系统，运行并测试上述改进的可行性及有效性。

# 参考文献

[1] Rajkumar, R.; Lee, C.; Lehoczky, J.P.; Siewiorek, D.P., "Practical solutions for QoS-based resource allocation problems," *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE* , vol., no., pp.296,306, 2-4 Dec 1998

[2] Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.

[3] Toshniwal, Ankit, et al. "Storm@ twitter." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.

[4] Xin, Reynold S., et al. "Graphx: A resilient distributed graph system on spark." First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013.

[5] Zaharia, Matei, et al. "Spark: cluster computing with working sets."Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

[6] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[7] Zaharia, Matei, et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling." Proceedings of the 5th European conference on Computer systems. ACM, 2010.

[8] Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.

[9] Low, Yucheng, et al. "Graphlab: A new framework for parallel machine learning." arXiv preprint arXiv:1006.4990 (2010).

[10] [Hitesh Ballani , Paolo Costa , Thomas Karagiannis , Ant Rowstron, Towards predictable datacenter networks, ACM SIGCOMM Computer Communication Review, v.41 n.4, August 2011](http://dl.acm.org/citation.cfm?id=2018465&CFID=617641976&CFTOKEN=19180186)

[11] [Ronnie Chaiken , Bob Jenkins , Per-Åke Larson , Bill Ramsey , Darren Shakib , Simon Weaver , Jingren Zhou, SCOPE: easy and efficient parallel processing of massive data sets, Proceedings of the VLDB Endowment, v.1 n.2, August 2008](http://dl.acm.org/citation.cfm?id=1454166&CFID=617641976&CFTOKEN=19180186)

[12][Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM, v.51 n.1, January 2008](http://dl.acm.org/citation.cfm?id=1327492&CFID=617641976&CFTOKEN=19180186)

[13] [Michael Isard , Mihai Budiu , Yuan Yu , Andrew Birrell , Dennis Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, March 21-23, 2007, Lisbon, Portugal](http://dl.acm.org/citation.cfm?id=1273005&CFID=617641976&CFTOKEN=19180186)

[14] S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29-43.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, EECS Department, University of California, Berkeley, May 2010.

[16] The Yahoo! Blog: https://developer.yahoo.com/blogs/ydn/storm-yarn-released-open-source-143745133.html

[17] Song, Ying, et al. "A service-oriented priority-based resource scheduling scheme for virtualized utility computing." High Performance Computing-HiPC 2008. Springer Berlin Heidelberg, 2008. 220-231.

[18] Sha, Lui, et al. "Real time scheduling theory: A historical perspective." Real-time systems 28.2-3 (2004): 101-155.

[19] Hunt, Patrick, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." USENIX Annual Technical Conference. Vol. 8. 2010.

[20] S4 Distributed stream computing platform. http://incubator.apache.org/s4/

[21] Spark Streaming. http://spark.incubator.apache.org/docs/latest/streaming-programming-guide.html

[22] Gonzalez, Joseph E., et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." OSDI. Vol. 12. No. 1. 2012.

[23] Facebook Engineering. Under the Hood: Scheduling MapReduce jobs more efficiently with Corona, November 2012.

[24] Schwarzkopf, Malte, et al. "Omega: flexible, scalable schedulers for large compute clusters." Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013.

[25] Xu, Yiqi, Adrian Suarez, and Ming Zhao. "Ibis: interposed big-data i/o scheduler." Proceedings of the 22nd international symposium on High-performance parallel and distributed computing. ACM, 2013.

[26] Wang, Ke, and Ioan Raicu. "Scheduling Data-intensive Many-task Computing Applications in the Cloud."

[27] Narayanamurthy, Shravan, et al. "Towards resource-elastic machine learning." (2013).

[28] Schuh, Hannes. "Collecting Execution Statistics of Scientific Workflow on Hadoop YARN."Dowling, Jim. "Attack of the Borg: Decentralized Resource Management." (2014).

[29] Ivanov, Todor, Nikolaos Korfiatis, and Roberto V. Zicari. "On the inequality of the 3V's of Big Data Architectural Paradigms: A case for heterogeneity." arXiv preprint arXiv:1311.0805 (2013).

[30] Raj, R. Sandeep, and G. Prabhakar Raju. "An Enhanced Approach for Resource Management Optimization in Hadoop."

[31] Kromonov, Ilja, Pelle Jakovits, and Satish Narayana Srirama. "NEWT-A resilient BSP framework for Iterative algorithms on hadoop YARN." High Performance Computing & Simulation (HPCS), 2014 International Conference on. IEEE, 2014.