LEC 1:Introduction

什么是分布式系统?

多台协同工作的计算机。

大型网站的存储，MapReduce，P2P 文件交换系统(peer-to-peer sharing),&c,DNS 域名解析。

许多关键的基础设施是分布式的。

为何选择分布式架构?

- 联通物理上分散的节点。
- 通过各个节点的资源隔离保证安全。
- 通过备份实现高可用(通过复制实现容错)，高可用(high availability)通常来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性。计算机系统的可用性可用平均无故障时间(MTTF)来度量，即计算机系统平均能够正常运行多长时间，才发生一次故障。可用性越高，平均无故障时间越长。
- 通过并行的 CPU/mem/disk/net 来达到横向扩展,扩容(来提高吞吐量，吞吐量是指在单位时间内中央处理器(CPU)从设备读取->处理->存储信息的量，高吞吐意味着系统可以同时承载大量的用户使用。高并发是高吞吐的延伸需求。)

分布式系统要求在不同的机器上进行调用，网络通信时间明显大于单机服务，那为什么说能提高吞吐量呢？为什么通过并行的 CPU/mem/disk/net 这些能提高吞吐量呢?

只有当单个节点的处理能力无法满足日益增长的计算，存储任务，且硬件的提升(加内存，加磁盘，使用更好的 CPU)高昂到得不偿失时候，应用程序也不能进一步优化的时候，我们才需要考虑分布式系统。那么一台服务器 CPU/mem/disk/net 等单位时间的吞吐量显然是小于分布式系统，因为分布式系统明显是有很多台服务器的。所以当单台服务器运行不了的时候(假设其耗时为 N)，将其任务分别包给多台机器，运算完再返回其总体时间是要小于 N 的。值得注意的事，单个可执行任务的单机吞吐量是大于分布式系统的，因为要考虑通信消耗，但是我们需要考虑的是一整个系统的吞吐量

但是分布式系统实现很复杂，需要解决各个层次上的并发(多个并发部分)，肯定会出现部分节点失效的情况(必须处理部分失败的情况)，还需要有很强的系统性能优化能力，即难以实现的性能潜力(操作系统、文件系统、网络 Lan->Wan、数据库等底层的优化使用)。

主题

这是一门被应用程序使用的基础架构课程。它会对分布式系统的复杂性进行抽象，其包括下面三个抽象：存储，通讯，计算(图表:用户、应用服务器、存储服务器)

这三个领域也是体系结构的老问题，这三个领域中关于分布式系统工程实现都有一些共性需要去解决的问题，也是我们的主题，也将反复出现

实现(implementation)

RPC 机制、线程机制、并发控制等如何高效实现

性能(performance)

理想:

可伸缩的吞吐量。通过购买更多的机器处理更高的负载。

扩展变得越来越困难：

负载不均衡

straggler(Some node is much more slower than others. 慢节点)

共享资源形成瓶颈等情况如何处理，例如网络

部分逻辑无法并发

不可并发代码：初始化、交互(initialization,interaction)

请注意，一些性能问题不容易通过扩展来解决，例如减少单个用户请求的响应时间，以及一些算法问题，即比起增加更多的机器，倒不如聘请一个算法工程师使代码运行所占内存更小，运行更快。

容错(fault tolerance)

上千的服务器，复杂的网络->总是会有东西出错，我们期望从应用程序中隐藏这些错误。

我们经常希望：

Availability(可用性):即时出错系统也可以继续使用

Durability(耐用性):当故障修复后，应用数据可以恢复

重要理念：

复制服务器。如果一个服务器故障了，客户端可以使用连接别的服务器

一致性(consistency)

通用的基础架构需求定义明确的行为。例如：Get(k)获取到的值应该是最近的 Put(k,v)设置的(这里的 put 是指物理上最近的机器上获取，还是指近期获取的缓存中获取?)。

实现良好的行为是很困难的!因为“副本”服务器很难保持一致；客户端可能在多步更新的中途崩溃；服务器可能会在“执行之后回复之前”等一些

尴尬的时刻崩溃；网络可能会让还存活的服务器(需要即时通信的服务器)看起来像挂掉一样；存在“脑裂”的风险。

一致性和性能不能兼得，一致性需要沟通,如获取最新的 Put()；“强一致性”经常使得系统缓慢(带有严格同步语义的系统往往是缓慢的。)；高性能通常会给应用程序带来“弱一致性”。那么如何做到性能与一致性之间的设计平衡是工程师应该研究的地方

工程师在设计一个分布式系统时，应当充分考虑到上面的要点，根据实际情况作出相应的设计。

让我们以 MapReduce 为例看看这个架构如何碰到上面的这些问题，又是如何解决的，同时也是 lab01 的关注点。

MapReduce 概要

- 背景：严格来讲，MapReduce 是一种分布式计算模型，用于解决大于 1TB 数据量的大数据计算处理。在 TB 级别的数据集上需要很多个小时才能完成计算，例如爬取网页后分析其图形结构只有在 1000 台计算机的情况下才可行，而这通常不是由分布式系统开发专家开发，一旦发生错误就会非常痛苦。著名的开源项目 Hadoop 和 Spark 在计算方面都实现的是 MapReduce 模型。从论文中可以看到花了不少篇幅在讲解这个模型的原理和运行过程。
- 总体目标：非分布式专家的程序员可以轻松的在合理的效率下解决的巨大的数据处理问题。程序员定义 Map 函数和 Reduce 函数、顺序代码一般都比较简单。MR 在成千的机器上面运行处理大量的数据输入，隐藏全部分布式的细节。

MapReduce 的抽象视图

input is divided into M files //输入被分割成 M 个文件

[diagram: maps generate rows of K-V pairs, reduces consume columns]

```
Input1 -> Map -> a,1 b,1 c,1
Input2 -> Map ->      b,1
Input3 -> Map -> a,1        c,1
                          |    |    |
                          |    |    -> Reduce -> c,2
                          |    -----> Reduce -> b,2
                          ---------> Reduce -> a,2
```

//数字是出现的次数,Reduce 是合并出现的次数,减少 key

MR calls Map() for each input file, produces set of k2,v2

"intermediate" data
each Map() call is a "task"
MR gathers all intermediate v2's for a given k2,
and passes them to a Reduce call
final output is set of <k2,v3> pairs from Reduce()
stored in R output files
[diagram: MapReduce API --
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2) -> list(k2, v3)]

例子：word count

input is thousands of text files
Map(k, v)
split v into words
for each word w
emit(w, "1")
Reduce(k, v)
emit(len(v))//因为是字符串所有如果有 3 个 w,reduce 后为”111”,len(v)=3

MapReduce 隐藏了很多令人痛苦的细节：①start s/w on servers(在服务器上运行软件)②跟踪完成了哪些任务③数据传送④从故障中恢复。

MapReduce 的模型设计很容易进行水平横向扩展以加强系统的能力，基本分为两种任务：map 和 reduce，通过 map 任务完成程序逻辑的并发，通过 reduce 任务完成并发结果的归约和收集，使用这个框架的开发者的任务就是把自己的业务逻辑先分为这两种任务，然后丢给 MapReduce 模型去运行。设计上，执行这两种任务的 worker 可以运行在普通的 PC 机器上，不需要使用太多资源。当系统整体能力不足时，通过增加 worker 即可解决。

详细说一下 MapReduce 的容易扩展性质：N 台电脑可以具有 Nx 的吞吐量(N 台计算机可以同时执行 nx 个 Map 函数和 Reduce 函数)，假设 M 和 R 大于等于 N，Map 函数不需要相互等待或者共享数据，完全可以并行的执行，对于 reduce 而言也是一样的。Map 和 reduce 唯一的交互是在”shuffle”。 在一定程度上，你可以通过购买更多的计算机来获取更大的吞吐量。而不是每个应用程序专用的高效并行。电脑是比程序员更便宜!

哪些将会成为性能的限制?

我们关心的就是我们需要优化的。CPU?内存? 硬盘? 网络? 在 2004 年这篇文章问世的时候回答还是”网络带宽“最受限，在论文中作者想方设法的减少数据

在系统内的搬运与传输，请注意，在 Map->Reduce shuffle 期间所有的数据都是通过网络传输的。论文的 root 交换机，1800 台机器传输速度在 100 到 200 千兆/秒，所有每台机器 55 兆/秒，这是很小的，比当时的磁盘霍尔 RAM 速度小的多。所以他们关心最小化网络上的数据传输。而到如今数据中心的内网速度要比当时快多了，因此如今更可能的答案恐怕就是磁盘了，新的架构会减少数据持久化到磁盘的次数，更多的利用内存甚至网络（这正是 Spark 的设计理念）。

更多细节(论文的 Figure 1):

master:给 workers 分配工作，记得运行时输出的中间结果是 M 个 Map 任务产生的，R 个 Reduce 任务输入存储在 GFS，每个 Map 输入文件拷贝三份，全部电脑运行 GFS 和 MR workers, 输入的任务(分片?)远远多于 worker 的数量, master 在每台机器上面执行 Map 任务，当原来的任务完成之后 map 会处理新的任务。

Map worker 将输出按 key 散列映射输出到 R 分区保存在本地磁盘上。

问题：有没好的数据结构可以实现这这个设计?

直到所有的 Maps 完成后 Reduce 再开始调用。

master 告诉 Reduce 处理者们获取从 Map workers 中产生的中间数据分区集合。Reduce workers 把最终的输出写入 GF(一个文件减少一个任务)。

如何设计可以降低网速慢带来的影响?

Map 的输入是从本地硬盘的 GFS 备份中读取，而不要通过网络来读取。

中间数据仅在网络中传输一次。Map worker 将数据写入本地磁盘,而不是 GFS。

中间数据通过 key 被划分到多个文件，”大网络传输“更加有效。Question:为什么不将 records 以 stream 形式传输到 reducer(通过 TCP), 因为它们是由 mappers 生成的?

参考论文 3.4 节减少网络带宽资源的浪费，都尽量让输入数据保存在构成集群机器的本地硬盘上，并通过使用分布式文件系统 GFS 进行本地磁盘的管理。尝试分配 map 任务到尽量靠近这个任务的输入数据库的机器上执行, 这样从 GFS 读时大部分还是在本地磁盘读出来。中间数据传输（map 到 reduce）经过网络一次，但是分多个 key 并行执行

他们是如何处理好负载均衡问题?

吞吐量是关键(Critical to scaling):某个 task 运行时间比较其他 N-1 个都长，大家都必须等其结束那就尴尬了，因此参考论文 3.5 节、3.6 节系统设计保证 task 比 worker 数量要多，做的快的 worker 可以继续先执行其他 task，减少等待。（框架的任务调度后来发现更值得研究），但是有些 reduce 可能就是比其他任务需要更长的时间。

[diagram: packing variable-length tasks into workers]

比 worker 多的多的任务的解决方案:

- Master 不断的将新的任务分配给那些已经完成之前任务的 worker。
- 希望没有任何一个任务是超级巨大以至于被其控制了(影响)完成时间。
- 同时速度更快的服务器将会处理更多的工作， 最后一起完成。

what about fault tolerance?

即如果服务器在 MR job 期间崩溃了怎么办?参考论文 3.3 节重新执行那些失败的 MR 任务即可，因此需要保证 MR 任务本身是幂等且无状态的。隐藏失败对于编程的易写性是很重要的一部分。

Question:为什么不重新开始整个 job 呢?

MR 仅重新运行那些失败的 Map 和 Reduce 任务。MR 需要他们是一些纯粹的函数:①它们不用在调用过程中保持状态②除了 MR 的 inputs/outputs，它们不用读或者写文件③任务之间没有隐藏的交流。

与其他并行编程方案相比，对纯粹函数的要求是 MR 的一个主要限制。但这对 MR 的简单性至关重要。

Details of worker crash recovery(MR 怎么应对 worker 崩溃)

Map Worker 崩溃:

master 看到 worker 不再对 pings 响应时就知道 work 崩溃了。已崩溃的 Map workers 产生的中间数据已丢失，但是每个 Reduce 任务都可能会需要它。

基于 GFS 的其他副本的输入输出传播任务 master 重新执行。

有些 Reduce workers 也许在读取中间数据的时候就已经失败，我们依赖于功能和确定性的 Map 函数。

如果 Reduces 已经获取全部的中间数据，那么 master 不需要重启 Map 函数;如果 Reduce 崩溃那么必须等待 Map 再次运行。Reduce worker 在输出结果前崩溃,master 必须在其他 worker 上面重新开始该任务。

Reduce worker crashes:

完成的任务可以存储在 GFS 中(带有副本)。

master 将未完成的任务交给其他的 workers。

Reduce worker 在输出结果的过程中崩溃:

GFS 会自动重命名输出，然后使其保持不可见直到 Reduce 完成，所以 master 在其他地方再次运行 Reduce worker 将会是安全的。

其他错误/问题:

如果 master 分配给两个 worker 同样的 Map()任务怎么办?

也许 master 错误的认为另一个 worker 挂掉了。

仅会告诉 Reduce workers 其中的一个。

如果 master 分配给两个 worker 相同的 Reduce()任务怎么办?

他们将会尝试将同样的输出文件写入到 GFS 中！GFS 的文件名不会重名，一个完整的文件将会被看到。

如果单个 master 非常慢--一个“散兵游勇”的计算机怎么办?

可能是因为机器的硬件不行了。master 开始最后几个未完成任务的副本。

如果 worker 计算的结果是不正确的，是因为软件还是硬件问题?

MR assumes “fail-stop” cpus and software

如果 master 崩溃了怎么办?

从 check-point 恢复或者放弃任务。

那些应用程序不适合用 MapReduce

不是所有的任务都适合使用 map/shuffle/reduce 的模式。

小数据，因为管理成本太高，如非网站后端。

大数据中的小更新，比如添加一些 document 到大的索引。

不可预知的读(Map 和 Reduce 都不能选择输入)。

Multiple shuffles, e.g. page-rank (can use multiple MR but not very efficient)。

多数灵活的系统允许 MR，但是使用非常复杂的模型。

现实世界的互联网公司是如何使用 MapReduce 的?

一家运营猫的社交网络互联网企业需要这样做:

1) 建立一个搜索索引，以便用户能够检索到其他人养的猫。
2) 分析不同猫的受欢迎程度，决定广告价值。
3) 检测狗，并删除它们的档案。

可以将 MapReduce 用于所有这些目的！--每天晚上对所有配置文件运行大量批处理作业

1) 建立倒序索引，让用户可以检索到其他用户的猫
2) 统计主页浏览次数：map(web logs) -> (cat_id, "1")
   reduce(cat_id, list("1")) -> list(cat_id, count)
3) 过滤档案:map(profile image) -> img analysis -> (cat_id, "dog!")
   reduce(cat_id, list("dog!")) -> list(cat_id)

结论

因为 MapReduce 的出现而使得计算机集群技术流行起来。但 MR 不是最有效或者最灵活的，但它具有良好的扩展性，并且易于编程，并且对开发者隐去了数据传输和容错的麻烦。其实还有部分工程问题，这篇文章中并没有讨论，可能

因为这些更偏重工程实践，比如：task任务的状态如何监控、数据如何移动、worker故障后如何恢复等。

最后总结一下MapReduce，这是个非常成功的分布式系统模型设计，尽管它可能不是某个问题的最佳解决方案，但是它是最通用化的解决方法（有点类似集装箱，不一定可以装最多，但是最容易标准化）。利用它你可以很轻松的将程序的逻辑进行标准化并放到多节点上并行执行。这种标准化模型的横向扩展性很强，同时因为标准化也解决了分布式系统中需要处理的种种问题，成功简化了分布式应用的开发，使得大数据处理程序得以工业级流水线生产，普通开发人员即可胜任，可谓是开启大数据时代的发明。它在工程设计上各个特性的取舍实践也很有学习的价值。我将在后续看到更高级的继任者。