分布式系统常见问题

一、概述

市场上的分布式系统越来越多，大概分为几类：

分布式存储：hdfs，ceph

分布式中间件：kafka，pulsar

分布式数据库：Cassandra，HBase，TiDB

分布式基础设施：k8s，zookeeper，etcd

上面这些系统都是“天然的分布式”系统，那么什么样的系统叫分布式，从几个点定义：

a. 运行在多台机器上，集群服务器数量从3台到几千台；
b. 要处理数据，所以内在是“有状态”系统；

常见的挑战：

- 多副本的一致性
- 各节点的不可靠需要的故障应对措施
- 网络延迟带来的不一致

综上，当多台服务器做数据存储时，会有如下几种出问题的方式：

1.进程崩溃

- 系统管理员常规维护，进程可能会被关闭
- 进行IO操作，但磁盘空间不足，且异常没有被正确处理，会导致进程被kill
- 云环境下，可能更加复杂，一些无关的原因也可能导致系统宕机

重要的是，如果进程负责存储数据，则必须将它们设计为能够对存储在服务器上的数据提供持久性保证。 即使进程突然崩溃，它也应保留所有成功给过用户ack确认消息的数据。 根据访问模式，不同的存储引擎具有不同的存储结构，范围从简单的hash表到复杂的图结构都有。 将数据刷新到磁盘是最耗时的操作之一，因此无法将每次插入或更新都立即刷新到磁盘。 因此，大多数数据库具有内存存储结构，内存中的数据定期刷新到磁盘。 如果进程突然崩溃，可能会丢失所有这些数据。

Write-Ahead Log的技术就是用来解决这个问题的。服务器将每个状态更改作为命令存储在硬盘上的仅附加(append-only)文件中。 append操作通常是非常快的，因此可以在不影响性能的情况下完成。 顺序写单个日志文件，以存储每个更新。 在服务器启动时，可以重播日志以再次建立内存状态。

这提供了持久性的保证。 即使服务器突然崩溃，然后重新启动，数据也不会丢失。 但是，在服务器恢复之前，客户端将无法获取或存储任何数据。 因此，如果服务器发生故障，我们会缺乏可用性。

一种显而易见的解决方案是将数据存储在多个服务器上。 因此，我们可以在多个服务器上复制预写日志。

但当涉及多个服务器时，还有更多的故障情况需要考虑。

2.网络延迟

在TCP / IP协议栈中，通过网络传输消息时所引起的延迟没有上限。 它会根据网络上的负载而变化。 例如，一个1 Gbps的网络连接可能会被定时触发的大数据作业淹没，从而填满网络缓冲区，并且可能导致某些消息在不确定的延迟后才到达服务器。

在典型的数据中心中，服务器打包在机架中，并且通过机架式交换机连接多个机架。 可能会有一棵交换机树将数据中心的一部分连接到另一部分。 在某些情况下，一组服务器可以相互通信，但与另一组服务器的连接是断开的。 这种情况称为网络分区。 服务器通过网络进行通信的基本问题之一是，何时能知道特定服务器发生了故障。

这里有两个问题要解决。

- 一台特定的服务器不能通过无限期的等待，去知道另一台服务器已经宕机。
- 不应有两套服务器，每套服务器都认为另一套服务器发生了故障，因此继续为不同的客户端提供服务。 这称为脑裂(split brain)。

为了解决第一个问题，每台服务器都会定期向其他服务器发送心跳消息。如果没有收到心跳，则将对应的服务器视为已崩溃。心跳间隔足够短，能够确保不需要花费很长时间就能检测服务器故障。在最坏的情况下，服务器可能在正常运行，但是被集群整体认为已经宕机，并继续运行。心跳这种方式可以确保提供给客户端的服务不会中断。

第二个问题是脑裂。在脑裂的情况下，如果两组服务器独立接受更新，则不同的客户端可以获取和设置不同的数据，当脑裂的问题被解决后，是不可能自动解决冲突的。

为了解决脑裂的问题，我们必须确保彼此断开连接的两组服务器不能独立地正常运行下去。为确保这一点，服务器执行动作时，只有大多数服务器可以确认这个动作，该才将动作视为成功。如果服务器无法选出来一个”大多数“，则它们将无法提供服务，并且某些客户端组可能无法接收该服务。但是集群中的服务器将始终处于一致状态。占多数的服务器数量称为Quorum。如何确定Quorum？这是根据集群可以容忍的故障数决定的。比如，我们有五个节点的群集，则需要三个quorum。通常，如果我们要容忍f个故障，则需要2f + 1的集群大小。

Quorum确保我们有足够的数据副本以承受某些服务器故障。但是，仅向客户提供强大的一致性保证是不够的。假设客户端在quorum上启动了写操作，但是该写操作仅在一台服务器上成功。Quorum中的其他服务器仍具有旧值。当客户端从Quorum读取值时，如果具有最新值的服务器可用，则它可能会获得最新值。但是，如果仅当客户端开始读取值时，具有最新值的服务器不可用，它就可以获得旧值。为了避免这种情况，需要有人跟踪判断quorum是否同意特定的操作，并且仅将值发送给保证在所有服务器上都可用的客户端。在这种情况下使用主从模式(master&slave)。其中一台服务器当选为master，其他服务器充当slave。Master控制并协调对slave的复制。Master需要确定哪些更改应对客户可见。高水位标记(High-water mark)用于跟踪预写日志中已成功复制到足够的slave中的条目。客

户端可以看到所有高水位之前的条目。Master还将高水位标记发送给slave。因此，如果Master失败并且slave之一成为新master，那么客户看到的内容就不会出现不一致之处。

3.进程暂停

即使有了Quorums和Master and Slave，仍然需要解决一个棘手的问题。 Master进程随时可能暂停。 进程暂停的原因很多。 对于支持垃圾回收的语言来说，可能会有很长的垃圾回收暂停。 具有较长垃圾收集暂停时间的Master可能会与slave断开连接，并在暂停结束后继续向slave发送消息。 同时，如果slave没有收到master的任何心跳信号，他们可能选择了新的master并接受了客户端的更新。 如果旧master的请求按原样处理，则它们可能会覆盖某些更新。 因此，我们需要一种机制来检测过时master的请求。 Generation Clock用于标记和检测来自老master的请求。 世代(generation)是单调增加的数字。

4.时钟不同步及顺序消息

从较新的消息中检测较旧的master消息的问题是如何保证消息顺序。看起来我们可以使用系统时间来给一组消息排序，但实际上，是不行的。主要原因是我们无法保证跨服务器的系统时钟是同步的。计算机中的一天中的时钟由石英晶体管理，并根据晶体的振荡来测量时间。

这种机制容易出错，因为晶体可以更快或更慢地振荡，因此不同的服务器可能具有截然不同的时间。一组服务器上的时钟通过称为NTP的服务进行同步。该服务会定期检查一组全局时间服务器，并相应地调整计算机时钟。

由于这是通过网络上的通信发生的，并且网络延迟可能会如以上各节中所述发生变化，所以由于网络问题，时钟同步可能会延迟。这可能导致服务器时钟彼此之间漂移，并且在NTP同步发生后甚至会倒退。由于计算机时钟存在这些问题，因此通常不将一天中的时间用于顺序事件。取而代之的是使用一种称为Lamport’s timestamp的简单技术。Generation Clock就是一个例子。

5.一致性

分布式一致性是分布式系统实现的一种特例，它提供了最强的一致性保证。 比如流行的企业架构中常见的Zookeeper，etcd和Consul。 他们实现了zab和Raft等一致性算法，以提供复制和强大的一致性。 还有其他流行的算法可以实现一致性，Google的Chubby中用于锁定服务的Paxos。

6.实现一致性的模式列表

一致性的实现中，使用 state machine replication来实现容错。在state machine replication中，存储服务，比如键值存储，在所有服务器上复制，而且用户的输入也会在每个服务器上以相同顺序执行。用于实现此目的的关键实现技术是在所有服务器上复制Write-Ahead log以具有“ Replicated Wal”。

我们可以将这些模式放在一起以实现Replicated Wal，如下所示。

为了提供持久性保证，使用Write-Ahead log。使用Segmented Log将预写日志分为多个段。这有助于Low-Water Mark进行日志清理。通过在多个服务器上复制预写日志来提供容错能力。服务器之间

的复制通过使用主从模式进行管理。更新High-Water Mark时，使用Quorum，以确定哪些值对客户可见。通过使用Single Update Quest，使所有请求均按严格顺序处理。使用Single Socket Channel, 将master的请求发送给slave时，事件将得到维护。为了优化Single Socket Channel上的吞吐量和延迟，使用了Request Pipeline。Slavet通过HeartBeat确定Master的可用性。如果Master由于网络分区而暂时从集群断开连接，则可以使用“Generation Clock”来检测它。

分布式系统是一个巨大的话题。 这里涵盖的模式集只是一小部分，涵盖了不同的类别，以展示模式方法如何帮助理解和设计分布式系统。 我将继续向这个合集添加下边这些问题。

- Group Membership and Failure Detection
- Partitioning
- Replication and Consistency
- Storage
- Processing