ZStack构建“正确的”云平台存储

2017221302006 周玉川

ZStack 有一条宗旨，“向客户交付稳定、可靠、高性能的云平台”，这条宗旨在前几年让他们一直聚焦云平台本身，包括虚拟化、云网络、云编排、存储管理等等这些功能。

在这里面难度最大的就是存储管理。

考虑到存储对业务的无比的重要性，以及他们作为一家创业公司的支持能力，他们一开始一直是基于一些开源的存储方案对客户提供服务：

1. XFS，作为 RHEL 默认的本地文件系统，他们原本一直对 XFS 是比较信任的，但实际上 XFS 在使用过程中问题多多，他们帮客户绕过了很多坑，也在考虑别的替代方案；

2. NFS，NFS 是一个对云平台很简单的方案，因为它屏蔽了很多存储的复杂性，用文件系统的方式提供了共享存储，使得他们可以用类似本地文件系统的管理方式管理共享存储，既简单又支持热迁移等高级功能，看似完美，但实际上 NFS 几乎是他们最不推荐的生产用存储方案之一；

3. OCFS2，当用户只有 SAN 存储，也无法提供 NFS 接口时，他们的选择并不多，此时 Oracle 的 OCFS2 成为一个值得青睐的方案，其优点是在小规模使用时基本上很稳定，部署后也可以使用文件系统的方式使用，但在性能、大规模的扩展性和部分功能（例如文件锁）上支持也并不完美；

4. Ceph，基于 Ceph 可以提供很棒的存储方案，但 Ceph 相对复杂的部署运维对部分客户还是比较难接受，特别是在私有云中，很多客户习惯了 SAN 存储带来的性能和安全感，对他们来说也没有超大容量的需求或者随时需要灵活扩容，反而大厂商带来的安全感，或者能够将之前用在VMware 上的 SAN 存储继续用起来才是最重要的。

综合考虑前面的各种存储，NFS、OCFS2 的不完美促使他们提供一个能够管理共享存储的存储方案，这个方案要能达到下面的要求：

1. 部署速度要足够快，ZStack 的部署速度一向是业界前列，他们的标准一直是对于 Linux 有基本理解的人能够在 30 分钟内完成部署，这个时间是包括部署主存储、镜像仓库的时间的。

2. 能够扩展到足够大的规模，根据 SAN 存储的性能，单个集群应该可以接管几十到上百的服务器。

3. 性能能够完整发挥 SAN 存储的性能，IO 模式能够发挥 SAN 存储的 cache性能，对于 OCFS2 他们可以通过调整 block size来优化 OCFS2 性能，但如果在分层 SAN 存储上测试就会发现由于大 block size 带来的 IO pattern 变化，如果测试 4k 小文件随机写，性能并不稳定，无法像直接在物理机上对 LUN 测试前期全部写到高速盘上，带来了测试数据的不理想。

4. 高稳定性，与互联网、公有云业务不同，私有云均部署在客户机房，甚至是一些隔离、保密机房，这意味着他们无法像互联网环境一样执行“反复试错”的策略，他们无法控制用户的升级节奏，无法时刻监控运维存储状态，也无法再客户环境进行灰度测试、镜像验证。

最终，在2018 年他们决定自己开发一个面向共享块存储的存储方法，命名很直接就叫 SharedBlock。整个方案是这样的：

1. 基于块设备，直接基于块设备向虚拟机提供虚拟云盘，通过避免文件系统开销可以明显提升性能和稳定性；

2. 在块设备上基于 Paxos 实现分布式锁来管理块设备的分配和节点的加入、心跳、IO 状态检查；

3. 通过 Qemu 的接口实现对用户磁盘读写状况进行监控；

SharedBlock在推出后，应用在了很多的生产客户上，特别是可以利旧SAN 存储特点让 SharedBlock 快速部署在大量以往使用虚拟化的客户上。

后来随着 5G 和物联网、云端互联的发展，让市场迫切需要一个价格不高、可以简便部署、软硬一体的超融合产品，因此他们就在考虑一个两节点一体机的产品，通过和硬件厂商合作设计，可以实现 2U 的一体机包含足够用户使用的硬盘、独立的模块和双电冗余，他们希望能通过这个产品将客户的原本单节点运行的应用平滑升级到两节点备份，让客户的运行在轨道站点、制造业工厂这些“端”应用既享受到云的便利，又不需要复杂的运维和部署。这就是他们的Mini Storage。

在开发这些存储产品的过程中，他们踩了无数的坑，也收获了很多经验。

关于存储做正确有多难，在今年说这个话题有一个热点事件是避不开的，就是今年的 FOSDEM 19' 上 PostgreSQL 的开发者在会上介绍了 PostgreSQL 开发者发现自己使用 fsync() 调用存在一个十年的 bug。

1. PG使用 writeback 机制，特别是在过去使用机械硬盘的时代，这样可以大大提高速度，但这就需要定时 fsync 来确保把数据刷到磁盘；

2. PG使用了一个单独线程来执行 fsync()，期望当写入错误时能够返回错误；

3.但其实操作系统可能自己会将脏页同步到磁盘，或者可能别的程序调用 fsync()；

4.无论上面的哪种情况，PG 自己的同步线程在 fsync 时都无法收到错误信息；

这样 PG 可能误以为数据已经同步而移动了 journal 的指针，实际上数据并没有同步到磁盘，如果磁盘持续没有修复且突然丢失内存数据就会存在数据丢失的情况。

在这场 session 上 PG 的开发者吐槽了 kernel 开发以及存储开发里的很多问题，很多时候 PG 只是想更好地实现数据库，但却发现经常要为 SAN/NFS 这些存储操心，还要为内核的未文档的行为买单。

从他们一个云厂商的角度看来，虚拟机存储使用 NFS 遇到的问题包括但不限于这几个：

1.部分客户的存储不支持 NFS 4.0 带来一系列性能问题和并发问题，而且 4.0 之前不支持 locking；

2. nfs服务本身会带来安全漏洞；

3.对于在 server 上做一些操作（例如 unshare）带来的神秘行为；

4.使用 async 挂载可能会带来一些不一致问题，在虚拟化这种 IO 栈嵌套多层的环境可能会放大这一问题，而使用 sync 挂载会有明显的性能损失；

5. NFS本身的 bug

最终他们的建议就是生产环境、较大的集群的情况下，最起码，少用 NFS 4.0 以前的版本……

另一个出名的文章是发表在 14 年 OSDI 的这篇 All File Systems Are Not Created Equal，作者测试了数个文件系统和文件应用，在大量系统中找到了不乏丢数据的 Bug， 在此之后诸如 FSE'16 的 Crash consistency validation made easy 又找到了gmake、atom 等软件的各种丢数据或导致结果不正确的问题：

上面他们举了很多软件、文件系统的例子，这些都是一些单点问题或者局部问题，如果放在云平台的存储系统上的话，复杂度就会更高：

1.首先，私有云面临的是一个离散碎片的环境，他们都知道 Android 开发者往往有比 iOS 开发者有更高的适配成本，这个和私有云是类似的，因为客户有：

1）不同厂商的设备

2）不同的多路径软件

3）不同的服务器硬件、HBA 卡

2. 由于他们是产品化的私有云，产品化就意味着整套系统不可能是托管运维，也不会提供驻场运维，这样就会明显受客户参差不齐的运维环境和运维水平限制。

3. 漫长的存储路径，对于平台来说，他们不仅要操心 IO 路径——Device Mapper、多路径、SCSI、HBA 这些，还要操心虚拟化的部分——virtio 驱动、virtio-scsi、qcow2…… 还要操心存储的控制平面——快照、热迁移、存储迁移、备份…… 很多存储的正确性验证只涉及选举、IO 这部分，而对存储管理并没有做足够的关注，而根据他们的经验，控制面板一旦有 Bug，破坏力可能比数据面更大。

虽然难处很多，但他们仍然解决了，提高了存储的正确性。

在一定程度上可以说 ZStack 很早就开始在用混沌工程的思想测试系统的稳定性，首先他们有三个关键性的外部整体测试：

1.MTBF，这个概念一般见于硬件设备，指的是系统的正常运行的时间，对他们来说会在系统上根据用户场景反复操作存储（创建、删除虚拟机，创建、删除快照，写入、删除数据等）在此之上引入故障检查正确性；

2.DPMO，这个是一个测试界很老的概念，偏向于单个操作的反复操作，例如重启 1000 次物理机，添加删除 10000 次镜像等等，在这之上再考虑同时引入故障来考察功能的正确性；

3.Woodpecker，这是 ZStack 从最开始就实现的测试框架，代码和原理都是开源的，它会智能的组合ZStack 的上千个 API自动找到可以持续下去的一条路径，根据资源当前的状态判断资源可以执行的 API，这样一天下来可以组合执行数万次乃至上百万次，与此同时再考虑引入错误。

上面这些方法，在大量调用 API、测试 IO 之外，很重要的一点就是注入错误，例如强制关闭虚拟机、物理机，通过可编程 PDU 模拟断电等等，但是这些方法有一些缺陷：

1.复杂场景的模拟能力有限，例如有些客户存储并不是一直 IO 很慢，而是呈现波峰波谷的波浪型，这种情况和 IO 始终有明显 delay 是有比较大的区别的。

2.不够灵活，例如有的客户存储随机 IO 很差但顺序 IO 性能却还可以，也不是简单的降低 IO 性能就可以模拟的。总之大部分混沌工程所提供的手段（随机关闭节点、随机杀进程、通过 tc 增加延时和 iproute2、iptables改变网络等等）并不能满足 ZStack 的完全模拟用户场景的需求。

在这种情况下，他们将扩展手段放在了几个方向上：

1.libfiu，libfiu 可以通过 LD\_PRELOAD 来控制应用调用 POSIX API 的结果，可以让应用申请内存失败、打开文件失败，或者执行 open 失败。使用 fiurun + fiuctl 可以对某个应用在需要的时刻控制系统调用。fiu对注入 libaio 没有直接提供支持，但好在 fio 扩展和编译都极为简单，因此他们可以轻松的根据自己的需求增加 module。

2. systemtap，systemtap 是系统界的经典利器了，可以对内核函数的返回值根据需求进行修改，对内核理解很清晰的话，systemtap 会很好用，如果是对存储进行错误注入，可以重点搜 scsi 相关的函数。

3. device-mapper，device-mapper 提供了 dm-flakey、dm-dust、dm-delay，当然你也可以写自己的 target，然后可以搭配 lio 等工具就可以模拟一个 faulty 的共享存储，得益于 device-mapper 的动态加载，他们可以动态的修改 target 和参数，从而更真实的模拟用户场景下的状态。

4. nbd，nbd 的 plugin 机制非常便捷，他们可以利用这一点来修改每个 IO 的行为，从而实现出一些特殊的 IO pattern，举例来说，他们就用 nbd 模拟过用户的顺序写很快但随机写异常慢的存储设备。

5. 此外，还有 scsi\_debug 等 debug 工具。