

ZStack 技术白皮书精选

如何构建“正确的”云存储

扫一扫二维码，获取更多技术干货吧



 ZStack中国社区@二群
扫一扫二维码，加入群聊。



长按扫码，关注ZStack官微

版权声明

本白皮书版权属于上海云轴信息科技有限公司，并受法律保护。转载、摘编或利用其它方式使用本调查报告文字或者观点的，应注明来源。违反上述声明者，将追究其相关法律责任。

摘要

大道至简·极速部署，ZStack 致力于产品化私有云和混合云。

ZStack 是新一代创新开源的云计算 IaaS 软件，由英特尔、微软、CloudStack 等世界上最早一批虚拟化工程师创建，拥有 KVM、Xen、Hyper-V 等成熟的技术背景。

ZStack 创新提出了云计算 4S 理念，即 Simple（简单）、Strong（健壮）、Smart（智能）、Scalable（弹性），通过全异步架构，无状态服务架构，无锁架构等核心技术，完美解决云计算执行效率低，系统不稳定，不能支撑高并发等问题，实现 HA 和轻量化管理。

ZStack 发起并维护着国内最大的自主开源 IaaS 社区——zstack.io，吸引了 6000 多名社区用户，对外公开的 API 超过 1000 个。基于这 1000 多个 API，用户可以自由组装出自己的私有云、混合云，甚至利用 ZStack 搭建公有云对外提供服务。

ZStack 拥有充足的知识产权储备，积极申报多项软著和专利，参与业内标准、白皮书的撰写，入选云计算行业方案目录，还通过了工信部云服务能力认证和信通院可信云认证。ZStack 面向企业用户提供基于 IaaS 的私有云和混合云，是业内唯一一家实现产品化，并领先业内首家推出同时打通数据面和控制面无缝混合云的云服务商。选择 ZStack，用户可以官网直接下载、1 台 PC 也可上云、30 分钟完成从裸机的安装部署。

目前已有 1000 多家企业用户选择了 ZStack 云平台。

如何构建“正确的”云平台存储

作者：ZStack 王为

从 2015 年到现在，ZStack 有一条宗旨一直没有变过，就是向客户交付稳定、可靠、高性能的云平台，这条宗旨在前几年让我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。

在这里面最让我们头痛的，即使不是第一也能进前三的存在，就是存储管理。

ZStack 的存储之路

- ZStack 是一个极致产品化、高性能、智能的私有云平台
- 在做云平台的前几年，我们一直借助开源存储：OCFS2、XFS、NFS……
- 因为所有做过基础架构的人都会因存储诡异的报错信息、可怕的调试难度、惊人的破坏力而敬而远之

考虑到存储对业务的无比的重要性，以及我们作为一家创业公司的支持能力，我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，我们原本一直对 XFS 是比较信任的，但实际

上 XFS 在使用过程中问题多多，我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，使得我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 几乎是我们最不推荐的生产用存储方案之一，细节将在后面讨论；
3. OCFS2, 当用户只有 SAN 存储，也无法提供 NFS 接口时，我们的选择并不多，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 的不完美促使我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，我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。

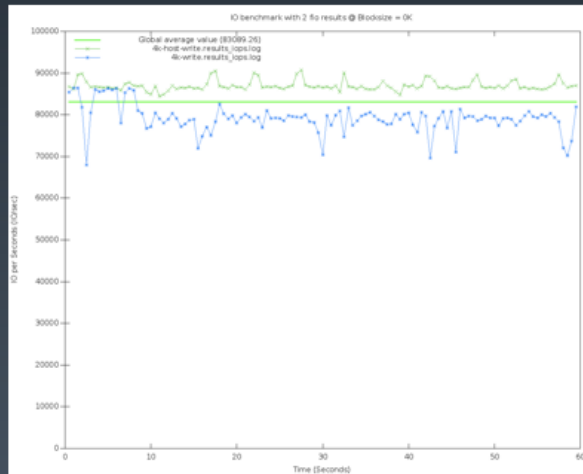
2. 能够扩展到足够大的规模，根据 SAN 存储的性能，单个集群应该可以接管几十到上百的服务器（因为一般来说单个 SAN 存储能支撑的服务器数量有限）。
3. 性能能够完整发挥 SAN 存储的性能，IO 模式能够发挥 SAN 存储的 cache 性能，对于 OCFS2 我们可以通过调整 block size 来优化 OCFS2 性能，但如果在分层 SAN 存储上测试就会发现由于大 block size 带来的 IO pattern 变化，如果测试 4k 小文件随机写，性能并不稳定，无法像直接在物理机上对 LUN 测试前期全部写到高速盘上，带来了测试数据的不理想。
4. 高稳定性，与互联网、公有云业务不同，私有云均部署在客户机房，甚至是一些隔离、保密机房，这意味着我们无法像互联网环境一样执行“反复试错”的策略，我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 年我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

- 完整发挥物理性能
- 极低延迟
- 快速部署
- 对 SAN 厂家、品牌无要求

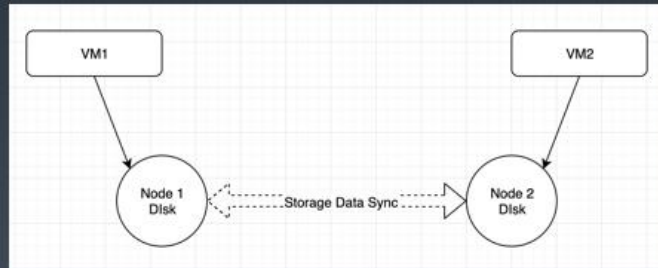


SharedBlock 在推出后，应用在了很多的生产客户上，特别是可以利旧 SAN 存储特点让 SharedBlock 快速部署在大量以往使用虚拟化的客户上。

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，因此我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，我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。这就是我们的 Mini Storage。

Mini Storage

- 近乎完整物理性能
- 低成本
- 快速部署
- 高稳定性



在开发这些存储产品的过程中，我们踩了无数的坑，也收获了很多经验。

下面先说说将存储做正确有多难，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。

PostgreSQL vs fsync()

- PG 使用 writeback 的机制，这样系统可能在后台默默 writeback 时出错
- 此时 IO layer/XFS 会对脏页做 AS_EIO 标记，调用 fsync() 时返回 EIO
- 但 fsync() 实际上存在一个未文档化的、clear-error-and-continue 的机制
- 也就是你下一次再调用 fsync() 时如果没有新的标记的脏页，可能就返回成功了！

https://archive.fosdem.org/2019/schedule/event/postgresql_fsync/

这里说到 NFS，不得不多提两句，在 Google 上搜索 "nfs bug" 可以看到五百万个结果，其中不乏 Gitlab 之类的知名厂商踩坑，也不乏 Redhat 之类的操作系统尝试提供遇到 NFS 问题的建议：



nfs bug



[All](#) [Images](#) [Videos](#) [News](#) [Shopping](#) [More](#) [Settings](#) [Tools](#)

About 5,650,000 results (0.30 seconds)

How we spent two weeks hunting an NFS bug in the ... - GitLab

<https://about.gitlab.com> › [blog](#) › 2018/11/14 › [how-we-spent-two-weeks-h...](#) ▼

Nov 14, 2018 - Thus launched an investigation into the inner workings of Git and the Network File System (NFS). The investigation uncovered a **bug** with the ...

How we spent two weeks hunting an NFS bug in the Linux ...

<https://news.ycombinator.com> › [item](#) ▼

I spend my days chasing **bugs** like this in the FreeBSD kernel, and make heavy use of dtrace. I expect that using something like bpftrace(1) might have ...

RHEL mount hangs: nfs: server [...] not responding, still trying ...

<https://access.redhat.com> › [solutions](#) ▼

Aug 9, 2019 - For example, the **NFS** Client networking misconfiguration, NIC driver or firmware **bug** causing **NFS** requests to be dropped, **NFS** Client firewall ...

Reporting bugs - Linux NFS

<https://wiki.linux-nfs.org> › [wiki](#) › [index.php](#) › [Reporting_bugs](#) ▼

Jul 23, 2011 - Places to report **bugs**. The official place to report **bugs** or make feature requests is linux-nfs@vger.kernel.org. Alternatively, the linux kernel ...

Top 10 NFS Issues and Solutions - Knowledgebase - NetApp

<https://kb.netapp.com> › [app](#) › [answers](#) › [answer_view](#) › [a_id](#) › [top-10-nfs-i...](#) ▼

Nov 6, 2018 - **Error** message: not found; **Error** message: access denied; **Error** message: Permission denied; Cannot mount / access an **NFS** volume or file ...

Bug #1603719 “[regression] NFS client: access problems after ...

<https://bugs.launchpad.net> › [bugs](#) ▼

Jul 17, 2016 - SRU Justification Impact: A regression in xenial causes automatic mounting of exported submounts of an **nfs** mount to fail. Fix: Change **nfs** to ...

Started work: 2016-07-25

Bug #214041 “NFS mounting hangs instead of returning with ...

<https://bugs.launchpad.net> › [bugs](#) ▼

从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

最终我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:



All File Systems Are Not Created Equal

- 上层开发者往往认为崩溃一致性是最基础的保证
- 实际上崩溃一致难度也是很高的
- 从文件系统到数据库, 已经被大家找出无数 Bug

<https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-pillai.pdf>

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open [†]
GDBM	2*	3*			
HSQldb	2	3	5		
SQLite-Roll	1*				
SQLite-WAL					
PostgreSQL			1 [†]		
Git	1*	3*	5*		3#*
Mercurial	2*	1*	6*	5	dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	5	12	25	17	9

(b) Failure Consequences.

Type	Application	LOC	Language	Version	Bug# (tracker)	Amp.	Consequence	d (bytes)
Utility	GNU make	39.0K	C	4.1	46193 (savannah)		Incorrect build	7.33K
	GNU zip	47.3K	C	1.6	22770 (debbugs)		Data loss	5.04K
	bzip2	8.12K	C	1.0.6	N/A (email)		Data loss	8.56K
	GNU coreutils sort	4.65K	C	8.21	22769 (debbugs)	✓	Data loss	23.9K
	Perl	801K	C	5.22	127663 (perlbug)		Data loss	17.4K
Productivity	Shelve	0.23K	Python	2.7.11	25442 (bug tracker)		Corruption	907
	Gimp	522K	C	2.8.14	763124 (bugzilla)	✓	Data loss	188K
	CuteMarkEd	21.8K	C++	0.11.2	285 (github)	✓	Data loss	5.61K
	T _E Xmaker	46.7K	C++	4.5	1553361 (launchpad)	✓	Data loss	1.61K
	T _E Xstudio	139.6K	C++	2.10.8	1693 (sourceforge)	✓	Data loss	1.61K
	Ted	3.7K	Java	1.0	45 (github)		Data loss	4.10K
	jEdit	188K	Java	5.1.0	3952 (sourceforge)		Data loss	1.61K
	GitHub Atom	55.8K	Node.js	1.5.3	10609 (github)	✓	Data loss	1.61K
	Adobe Brackets	117K	Node.js	1.5.0	12103 (github)	✓	Data loss	1.61K

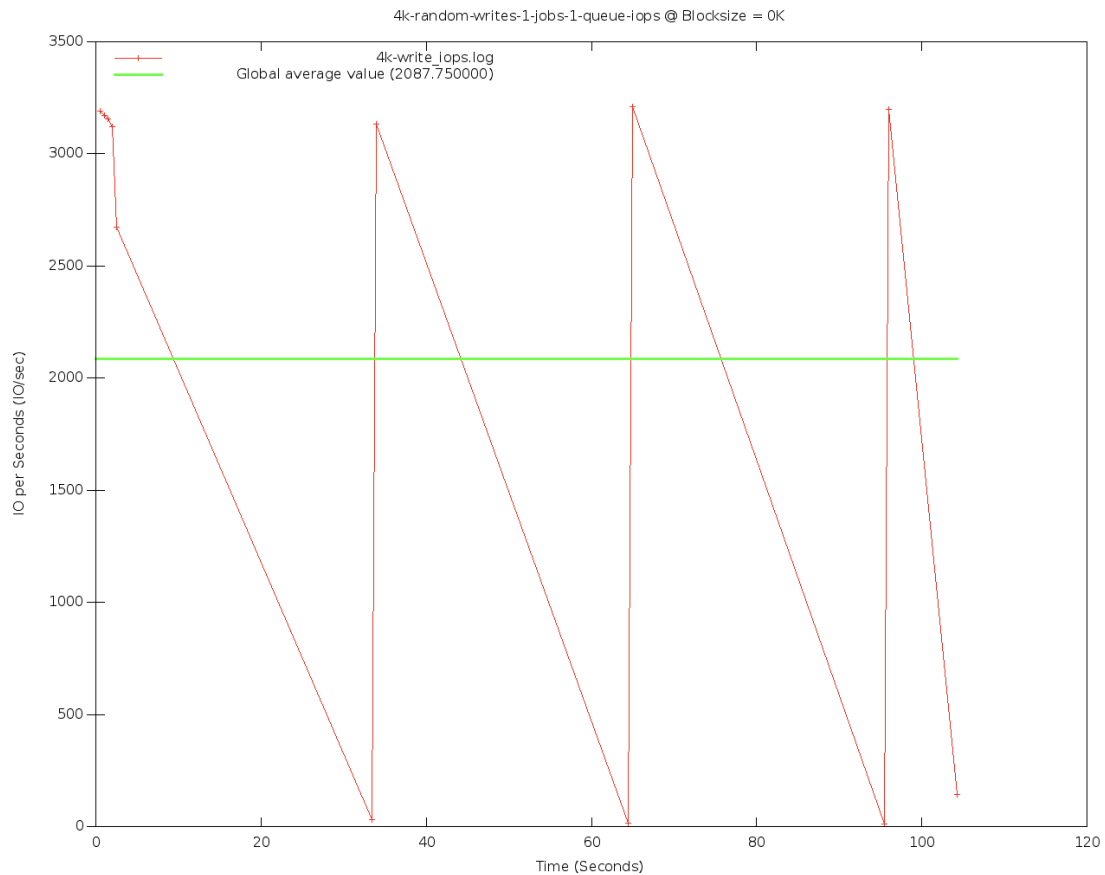
上面我们举了很多软件、文件系统的例子，这些都是些单点问题或者局部问题，如果放在云平台的存储系统上的话，复杂度就会更高：

1. 首先，私有云面临的是一个离散碎片的环境，我们都知道 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；

虽然 SCSI 指令是通用的，但实际上对 IO 出错、路径切换、缓存使用这些问题上，不同的存储+多路径+HBA 可以组成不同的行为，是最容易出现难以调试的问题地方，例如有的存储配合特定 HBA 就会产生下面的 IO 曲线：



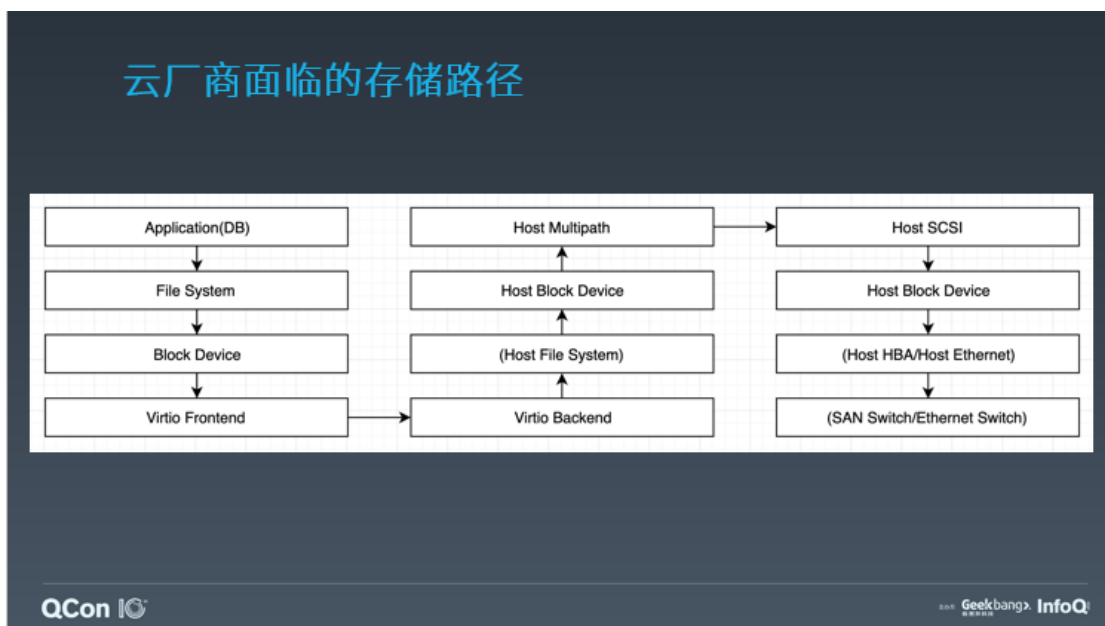
2. 由于我们是产品化的私有云，产品化就意味着整套系统不可能是托管运维，也不会提供驻场运维，这样就会明显受客户参差不齐的运维环境和运维水平限制：

1) 升级条件不同，有的用户希望一旦部署完就再也不要升级不要动了，这就要求我们发布的版本一定要是稳定可靠的，因为发出去可能就没有升级的机会了，这点和互联网场景有明显的区别；

2) 联网条件不同，一般来说，来自生产环境的数据和日志是至关重要的，但对产品化的厂商来说，这些数据却是弥足珍贵，因为有的客户机房不仅不允许连接外网，甚至我们的客户工程师进机房的时候手机也不允许携带；

3) 运维水平不同, 对于一个平台系统, 如果运维水平不同, 那么能发挥的作用也是不同的, 比如同样是硬件故障, 对于运维水平高的客户团队可能很快能够确认问题并找硬件厂商解决, 而有的客户就需要我们先帮忙定位分析问题甚至帮助和硬件厂商交涉, 就需要消耗我们很多精力。

3. 漫长的存储路径, 对于平台来说, 我们不仅要操心 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, 而根据我们的经验, 控制面板一旦有 Bug, 破坏力可能比数据面更大。



说了这么多难处, 我们来说说怎么解决。提到存储的正确性, 接触过分布式系统的同学可能会说 TLA+, 我们先对不熟悉 TLA+ 的同学简单介绍下 TLA+。

2002 Lamport 写了一本书《Specifying Systems》基本上算是 TLA+ 比较正式的第一本书，了解的朋友可能知道在此之前 Lamport 在分布式系统和计算科学就很出名了——LaTeX、Lamport clock、PAXOS 等等，TLA+ 刚开始的时候没有特别受重视，他的出名是来自 AWS 15 年发表在 ACM 会刊的《How Amazon Web Services Uses Formal Methods》。

从本质上讲，形式化验证并不是新东西，大概在上世纪就有了相关的概念，TLA+ 的优势在于它特别适合验证分布式系统的算法设计。因为对于一个可验证的算法来说，核心是将系统时刻的状态确定化，并确定状态变化的条件和结果，这样 TLA+ 可以通过穷举+剪枝检查当有并发操作时会不会有违反要求（TLA+ 称之为 invariant）的地方——例如账户余额小于 0，系统中存在了多个 leader 等等。

TLA+ 的发展之路

- 2002 年 *Specifying Systems*
- 2015 年 *How Amazon Web Services Uses Formal Methods*
- 2018 年 TLA Workshop
- 被 MongoDB, Elasticsearch 等应用

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
Internal distributed lock manager	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

看最近的几场 TLA Community Meeting，可以看到 Elasticserach、MongoDB 都有应用。

那么既然这个东西这么好，为什么在国内开发界似乎并没有特别流行呢？我们在内部也尝试

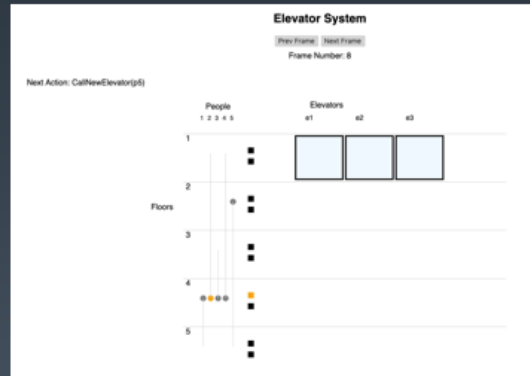
应用了一段时间, 在 Mini Storage 上做了一些验证, 感觉如果 TLA+ 想应用更广泛的话, 可能还是有几个问题需要优化:

1. 状态爆炸, 因为 TLA+ 的验证方式决定了状态数量要经过精心的抽象和仔细的检查, 如果一味地增加状态就可能遇到状态爆炸的问题;
2. TLA+ Spec 是无法直接转换成代码的, 反过来, 代码也无法直接转换成 Spec。那么换句话说, 无论是从代码到 Spec 还是从 Spec 到代码都有出错的可能, 轻则有 Bug, 重则可能导致你信心满满的算法其实与你的实现根本不同;
3. 外部依赖的正确性, 这一点可能有点要求过高, 但却也是可靠系统的重要部分, 因为用户是不管产品里是否用到了开源组件, 不论是 qemu 的问题还是 Linux 内核的问题, 客户只会认为是你的问题, 而我们不太可能分析验证每个依赖。

当然了, 涉及到算法的正确性证明, 形式化证明依然是不可替代的, 但不得不说目前阶段在云平台存储上应用, 还没做到全部覆盖, 当然了我们也看到 TLA+ 也在不断进步——

1. 可视化
2. 增强可读性
3. Spec 的可执行

未来的形式化验证：可视化



https://github.com/will62794/tlaplus_animation

未来的形式化验证：易读

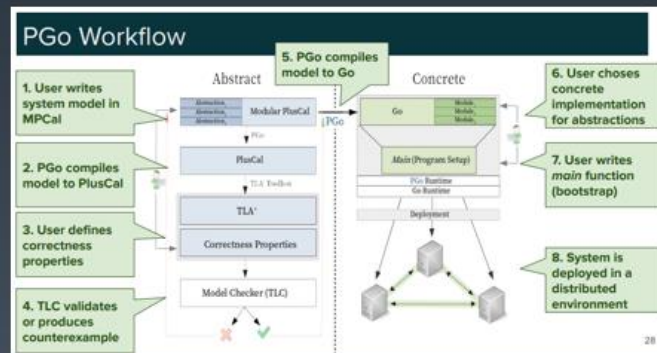
```

Original algorithm in DistAlgo
1 def setup(s):
2   self.s := s           # set of all other processes
3   self.q := {}         # set of pending requests with logical clock
4
5 def mutex(task):       # for doing task() in critical section
6   self.t := logical_time() # rule 1
7   send('request', t, self) to s
8   q.add('request', t, self)
9   await each ('request', t2, p2) in q | (t2, p2) != (t, self) implies (t, self) < (t2, p2) # rule 5
10  task()
11  send('release', t, self) to s # critical section
12
13 receive ('request', t2, p2): # rule 2
14   q.add('request', t2, p2)
15   send('ack', logical_time(), self) to p2
16
17 receive ('release', .. p2): # rule 4
18   q.del('request', .., p2)
19

```

<http://tda2018.loria.fr/contrib/jiu.pdf>

未来的形式化验证：可执行



<https://github.com/UBC-NSS/pgo>

这里特别是第三点，如果我们的 Spec 能够被转换成代码，那么我们就可以将核心代码的算法部分抽象出来，做成一个单独的库，直接使用被 Spec 证明过的代码。

分布式系统的测试和验证，这几年还有一个很热门的词汇，就是混沌工程。

混沌工程对大多数人来说并不是一个新鲜词汇，可以说它是在单机应用转向集群应用，面向系统编程转向到面向服务编程的必然结果，我们已经看到很多互联网应用声称在混沌工程的帮助下提高了系统的稳定性如何如何，那么对于基础架构软件呢？

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，首先我们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，对我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, 我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,因此我们可以轻松的根据自己的需求增加 module。

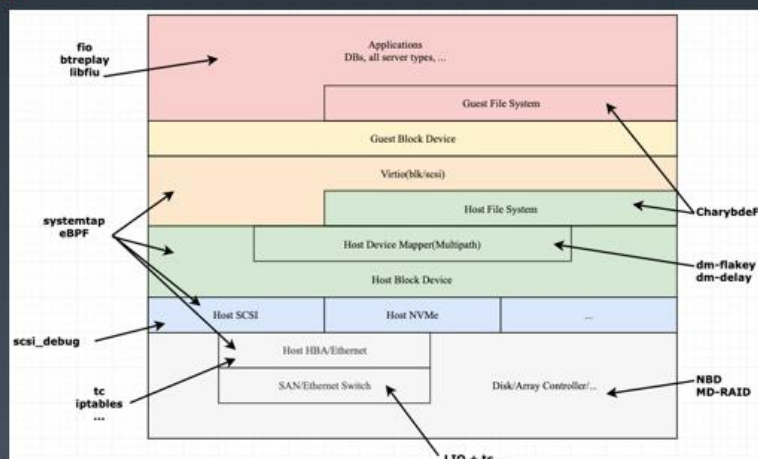
2. systemtap, systemtap 是系统界的经典利器了,可以对内核函数的返回值根据需求进行修改,对内核理解很清晰的话,systemtap 会很好用,如果是对存储进行错误注入,可以重点搜 scsi 相关的函数,以及参考这里: Kernel Fault injection framework using SystemTap
3. device-mapper, device-mapper 提供了 dm-flakey、dm-dust、dm-delay,当然你也可以写自己的 target,然后可以搭配 lio 等工具就可以模拟一个 faulty 的共享存储,得益于 device-mapper 的动态加载,我们可以动态的修改 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, 我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, 我们就用 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, 但这些比较面向特定问题, 就不细说了。

总结

手段	层面	便利程度	灵活程度	模拟特定模式
libfiu	用户态	很方便	限制较多	不可以
scsi_debug	内核态	比较方便	限制较多	不可以
systemtap	内核态	比较复杂	比较灵活	可以
device mapper	内核态	比较方便	限制较多	不可以
nbd	-	复杂	非常灵活	可以

总结



上面两张图对这些错误注入手段做了一些总结,从系统角度来看,如果我们在设计阶段能够验证算法的正确性,在开发时注意开发可测试的代码,通过海量测试和错误注入将路径完整覆盖,对遇到的各种 IO 异常通过测试 case 固化下来,我们的存储系统一定会是越来越稳定,持续的走在“正确”的道路上的。