支付宝 Alipay.com 知托付

如何构建高可用系统

平台技术部-基础数据部-解伦

Agenda

- **介绍篇** 可用性基本介绍
- **设计篇** 如何提高可用性
- **案例篇** case study
- **总结篇** 高可用设计原则

介绍篇

- 可用性的重要性

- 可用性 Vs. 可靠性
  - 如果系统在每小时崩溃1ms，那么它的可用性就超过99.9999%，但是它还是高度不可靠。与之类似，如果一个系统从来不崩溃，但是每年要停机两星期，那么它是高度可靠的，但是可用性只有96%。

如何衡量

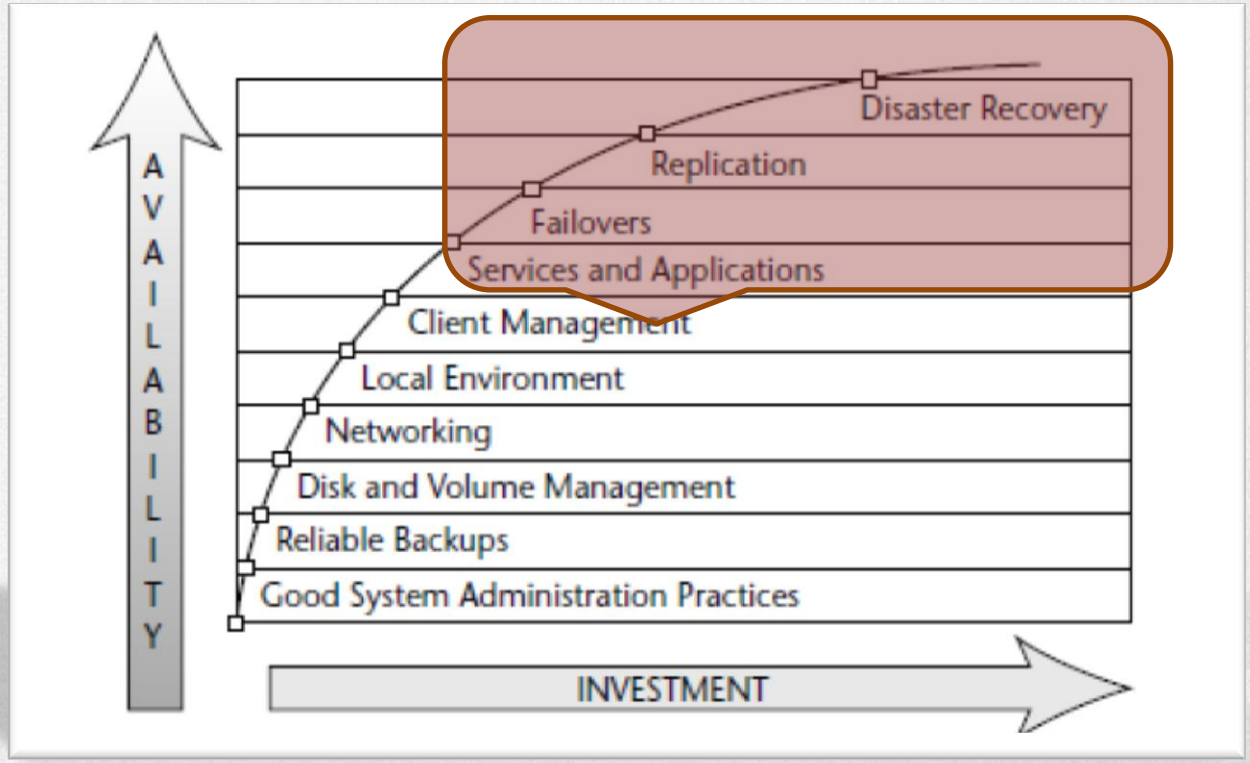
- 平均无故障时间（Mean Time Between Failure, **MTBF**）和平均恢复时间（Mean Time To Repair, **MTTR**）

- 可用性=MTBF/(MTBF + MTTR) * 100%

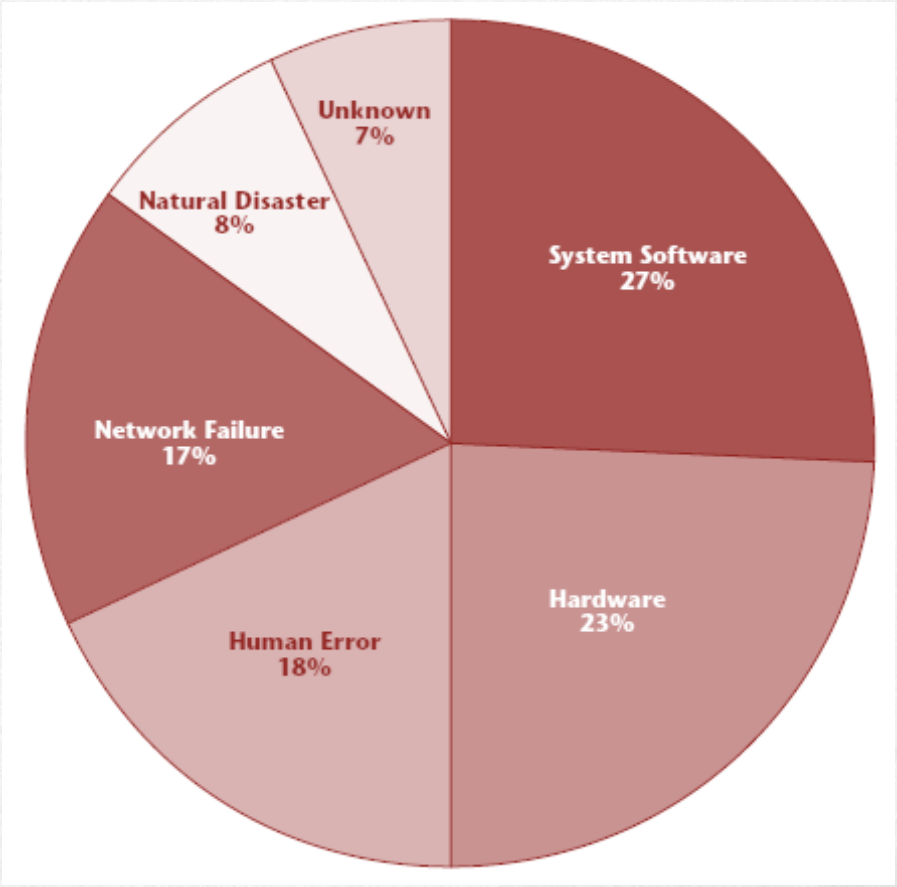
几个“九”的含义

PERCENTAGE UPTIME	PERCENTAGE DOWNTIME	DOWNTIME PER YEAR	DOWNTIME PER WEEK
98%	2%	7.3 days	3 hours, 22 minutes
99%	1%	3.65 days	1 hour, 41 minutes
99.8%	0.2%	17 hours, 30 minutes	20 minutes, 10 seconds
99.9%	0.1%	8 hours, 45 minutes	10 minutes, 5 seconds
99.99%	0.01%	52.5 minutes	1 minute
99.999%	0.001%	5.25 minutes	6 seconds
99.9999% (“six 9s”)	0.0001%	31.5 seconds	0.6 seconds

设计篇

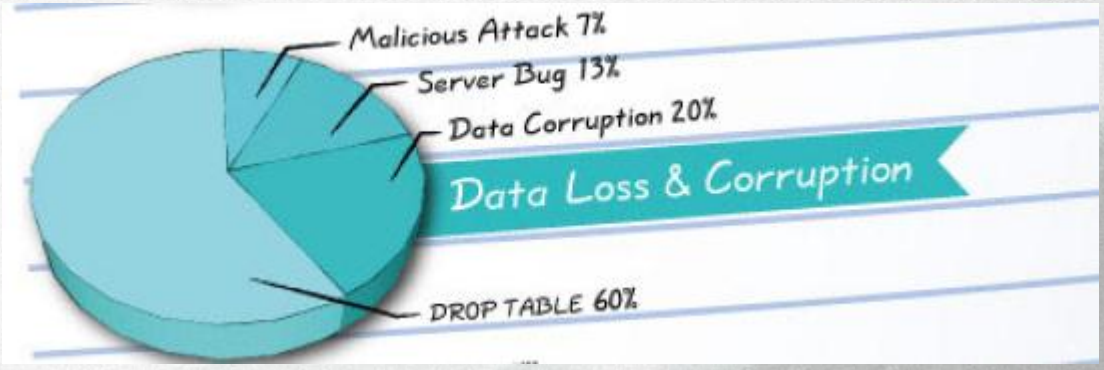

AVAILABILITY
Disaster Recovery
Replication
Failovers
Services and Applications
Client Management
Local Environment
Networking
Disk and Volume Management
Reliable Backups
Good System Administration Practices
INVESTMENT

Causes of Downtime



MySQL不可用原因


Operating Environment 35%
Data Loss and Corruption 10%
Performance 34%
Replication 21%
35%
10%
34%
21%


Malicious Attack 7%
Server Bug 13%
Data Corruption 20%
Data Loss & Corruption
DROP TABLE 60%

高可用三步走

- **减少故障发生的可能**
  - 0 bug 神话
- 减少故障恢复时间
  - Fast recovery
- 降低故障的外部影响
  - 应用服务降级
  - 牺牲部分特性

如何减少故障

- **内：避免单点故障 SPOF**
- 外：容错性设计防止扩散
- 综：有效的监控运维配合

避免单点故障

- 常见的**冗余**设计
  - RAID, Replica Set, Erasure Code
  - Back-Up, Reassign, Retry
  - Master-Slave, Mirror, Data Guard, RAC
    - So Many Replication, log shipping

容错性设计

- 重点关注**远程调用**
  - 减少对外部系统强依赖
    - 结果进行缓存
    - 异步代替同步的方式
  - 对外部依赖不信任原则
    - 结果进行有效性检查
    - 失败情况下的Failover
      - 适度重试原则：控制频率和次数

有效的内部监控

- 尽早发现瓶颈、隐患
  - 问题的扩散方式
  - 发现越早代价越小
  - 结合**内外部**有效监控

高可用三步走

- 减少故障发生的可能
  - 0 bug 神话
- **减少故障恢复时间**
  - Fast recovery
- 降低故障的外部影响
  - 应用服务降级
  - 牺牲部分特性

快速故障恢复

- 坚持无状态设计
  - 状态的持久化：Checkpoint + commit log
- 有效的故障隔离
  - 故障检测，黑白名单，流量分配
    - 虚假“故障”识别，黑名单的恢复机制
- 可运维性设计
  - 事实证明，故障恢复一大利器

高可用三步走

- 减少故障发生的可能
  - 0 bug 神话
- 减少故障恢复时间
  - Fast recovery
- **降低故障的外部影响**
  - 应用服务降级
  - 牺牲部分特性

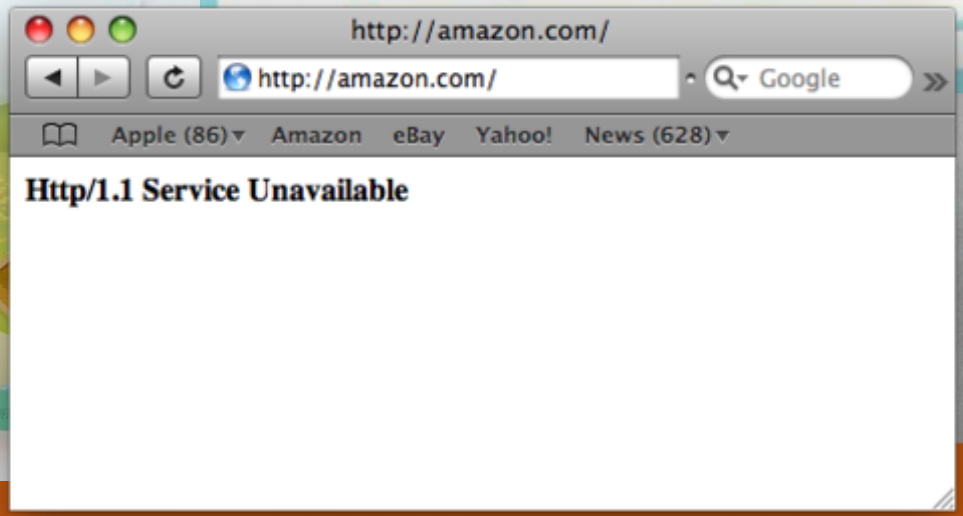
降低故障影响

- 过载保护
  - 异常发现：超时时间、最大并发、SQL识别
  - 资源流量限制：内存、网络、QPS， 队列
- 应用降级
  - 关闭部分不重要功能，柔性可用
  - 权衡用户体验、一致性、完整性

案例篇

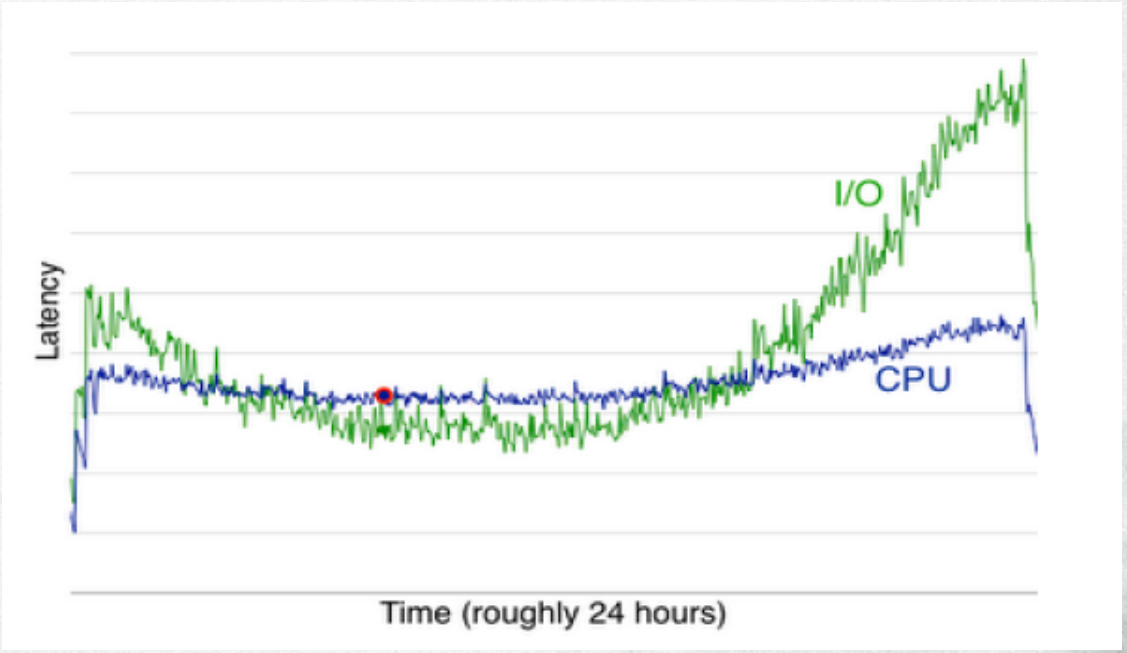

Amazon is currently experiencing a degradation. T
reddit is down.


twitter
Twitter is currently down for Unplanned maintenance.


http://amazon.com/
http://amazon.com/
Google
Apple (86)
Amazon
eBay
Yahoo!
News (628)
Http/1.1 Service Unavailable

Twitter

- 业务调用Memcached接口不检点，应对突发流量情况下，容量规划不足，IO成为瓶颈，性能严重下降，前端超时严重。



Foursquare

- 因为 Mongdb Shard 算法导致的数据分散不均衡，其中一台(Shard0)数据增长到 67GB(另外一台 50GB)，超过了 66GB 的限制，读写部分分散到磁盘上，性能急剧下降。

- 尝试增加第三台 Shard 机器，上线后开始迁移，读取从三台进行，Shard0 的数据迁移到 5% 的时候，但是写操作还是让 Shard0 宕机了。这个时候发现Shard0 存在数据碎片，即使数据迁移走，还是会占用原来的内存。

Amazon

- EBS 节点之间通过两个网络连接，主网络吞吐率较大，用于数据访问，另外一个是备用网络，用于保障节点之间通讯可靠性。
- 事故起因是维护时操作错误，网络流量被全部切换到备用网络，导致备用网络过载。网络不通导致控制系统认为服务器大量宕机，马上开始数据复制以替换“宕机”的服务器上的数据副本，引发了“复制风暴”，而由此增加的数据流量更加剧了网络过载，从而使故障在集群中蔓延，进入恶性循环。

Weibo

- **Cache雪崩**:在大并发存在热点的场景下，当cache失效时，大量并发同时取不到cache，会同一瞬间去访问db并回设cache，可能会给系统带来潜在的超负荷风险。我们曾经在线上系统出现过类似故障。@TimYang

Facebook

- 对一个配置数据的持久副本做了一点修改，让它显示为无效。这意味着每一个客户机都能看到这个无效数据，并且试图修复这个数据。因为修复过程牵涉到对数据库集群的查询，一下子这每秒钟百万次的查询迅速把集群累垮。

- 更糟糕的是，每次一个客户机试图查询数据库失败都认为是有一个无效数据，缓存里的相应的键值会被删除。这意味着即使最初的问题被解决了，请求查询的数据流仍然不会停止。直到数据库无法为其中的某些请求进行查询，而这又会给自己招致更多的查询。进入了一个循环反馈圈，使数据库无法恢复正常。

OceanBase

- 内部保护和防御不足，一次业务高峰期，客户端API 过度failover重试，最终将两个主备Cluster都加入黑名单，死循环产生了**重试风暴**，业务端线程加大起了反作用，最终导致全部不可用；

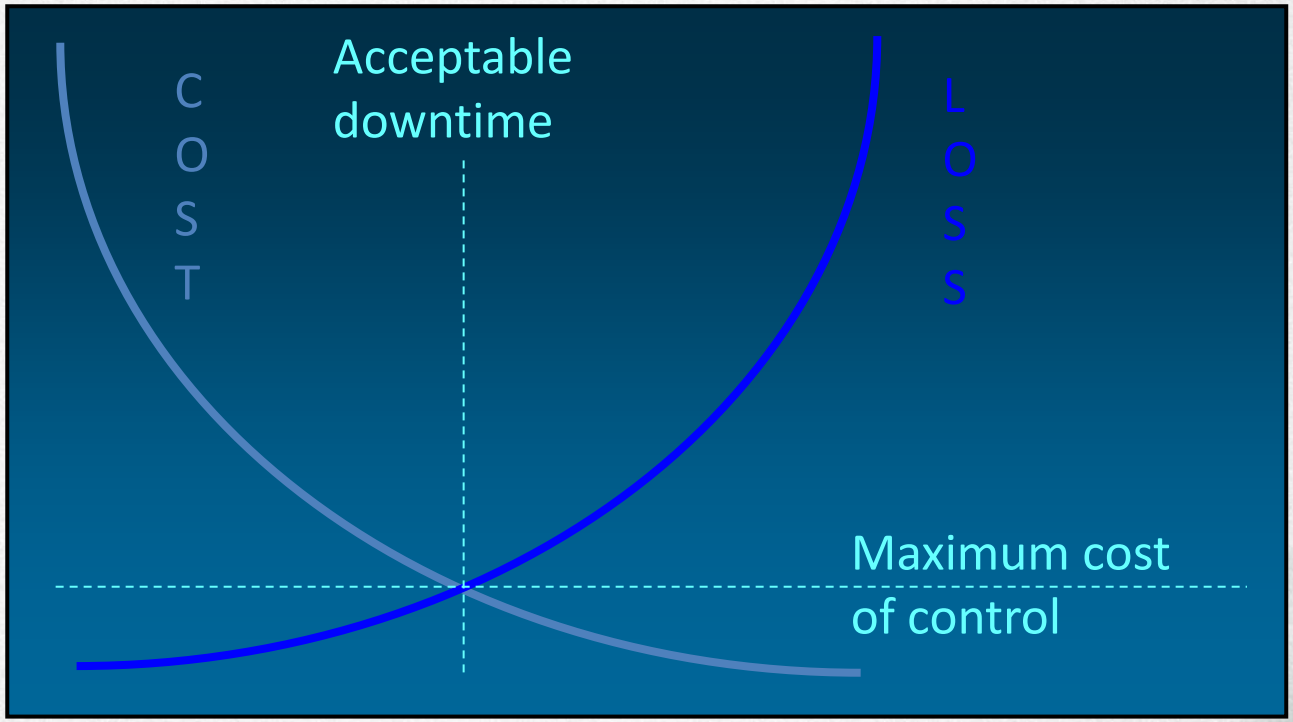
- 一次磁盘故障引发ChunkServer的bug,导致单机的延时加剧，MergeServer 的黑名单和负载均衡策略没有生效，导致应用严重超时；

总结篇

- 可用性的重要性
  - BASE：分布式的优势
  - CAP: No CAP, No CP
- 无处不在，赢在细节

20 Key High Availability Design Principles
#20: Don't Be Cheap
#19: Assume Nothing
#18: Remove Single Points of Failure (SPOFs)
#17: Enforce Security
#16: Consolidate Your Servers
#15: Watch Your Speed
#14: Enforce Change Control
#13: Document Everything
#12: Employ Service Level Agreements
#11: Plan Ahead
#10: Test Everything
#9: Separate Your Environments
#8: Learn from History
#7: Design for Growth
#6: Choose Mature Software
#5: Choose Mature, Reliable Hardware
#4: Reuse Configurations
#3: Exploit External Resources
#2: One Problem, One Solution
#1: K.I.S.S. (Keep It Simple . . .)
Key Points

tradeoff



参考资料

1. http://engineering.twitter.com/2010/02/anatomy-of-whale.html
2. http://www.infoq.com/news/2010/10/4square_mongodb_outage
3. http://aws.amazon.com/cn/message/65648/
4. http://timyang.net/programming/memcache-mutex/
5. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919

Thanks