Jepsen：测试PostgreSQL、Redis、MongoDB以及Riak的分区容忍性

作者 Kyle Kingsbury，译者 邵思华 发布于 2013年12月18日 | 1 讨论

分布式系统的特性是能够在高延迟或不可靠的传输条件下进行状态交换。如果要保证系统的操作的可靠性，必须保证在节点及网络两方面故障发生时的健壮性，但并非所有系统都能够满足我们所要求的安全能力。在本文中，我们将探索分布式数据库在设计上的一些考虑，以及它们是如何对网络分区的情况作出响应。

在两个节点间发送消息时，IP网络可能会任意地删除、延迟、重新排序或复制消息，因此许多分布式系统都使用TCP以防止消息的重新排序与复制。但TCP/IP在本质上仍然是异步的：网络会任意地延迟消息，连接可能会被随时切断。此外，对故障的诊断也并不可靠：要判断某个节点是否出现故障、网络连接是否被切断、或者操作是否比预计中运行得慢也许是不可能实现的。

消息被任意地延迟或切断的这种故障叫做网络分区。分区可能出于多种原因发生在生产环境的网络中：垃圾回收（GC）的压力、网卡（NIC）故障、交换机故障、配置出错、网络拥塞等等。由于分区的发生，使CAP定理限制了分布式系统能够达到的最大担保能力。当消息被切断时，“一致的”（CP）系统会拒绝某些节点的请求，以保持线性一致性。“可用的”（AP）系统虽然能够处理所有节点上的请求，但必须牺牲线性一致性，因为不同的节点对于操作的顺序可能会产生不同意见。当网络情况良好时，系统可以保证一致性与可用性，但由于真实的网络中总会产生分区，因此不存在能够完全做到“一致且可用”（CA）的系统。

另外值得一提的是，CAP定理不仅仅对数据库的整体有效，对其中的各种子系统，例如表、键、列，甚至是各种操作也有效。举例来说，数据库可以为每个键独立地提供一致性，但不能保证键之间的一致性。建立这一折衷的策略可以允许系统在分区发生时处理更多的请求数量。许多数据库能够调节单个读写操作的一致性级别，其代价就是性能与正确性。

测试分区

理论界定了一个设计空间，但真实的软件未必能够达到这种范围。我们需要测试某个系统的表现，以真实地了解它的表现情况。

首先，我们需要准备一组节点以进行测试，我在一台Linux计算机上搭建了5个LXC节点，但你也可以选择使用Solaris zone、VM、EC2节点以及物理硬件等等。你需要这些节点共享某种网络，在我的例子中使用了一个单独的虚拟网桥（virtual bridge interface），我将这些节点命名为n1、n2、…n5，并且在它们之间建立了DNS以及主机操作系统。

为了创建一个分区，你需要某种方式以切断或延迟消息，例如防火墙规则。在Linux上，你可以使用**iptables -A INPUT -s some-peer -j DROP**命令以造成一种单向的分区，使某些节点传给当前节点的消息会被切断。当你在多台主机上应用了这些规则之后，就建立了一个网络数据丢失的人为的模式。

在多台主机上重复地运行这些命令需要你花费一些精力，我使用了一个我自己编写的工具，名为Salticid。但你也可以使用CSSH或其它任意一种集群自动化系统。关键的因素是延迟——你需要能够快速地启动和终止某个分布，因此像Chef或其它慢收敛系统可能没有太大用途。

接下来你需要在这些节点上搭建分布式系统，并设计一个应用程序以进行测试。我写了一个简单的测试：这是一个Clojure程序，它运行在集群的外部，用多线程模拟五个隔离的客户端。客户端会并发地将N个整数加入到分布式系统中的某个集合中，一个节点写入0、5、10……，另一个节点写入1、6、11……等等。无论成功或

失败、每个客户端都会将它的写操作记录在日志中。当所有写操作都完成后，程序会等待集群进行收敛，随后检查客户端的日志是否与数据库中的实际状态相符。这是一种简单的一致性检查，但可以用来测试多种不同的数据模型。

本示例中的客户端与配置自动化，包括模拟分区与创建数据库的脚本都可以免费下载。请单击这里以获取代码与说明。

PostgreSQL

单节点的PostgreSQL实例是一个CP系统，它能够为事务提供可串行的一致性，其代价是当节点故障时便不可用。不过，这个分布式系统会选择为服务器进行妥协，某个客户端并不能保证一致性。

Postgres的提交协议是两阶段提交（2PC）的一个特例，在第一阶段，客户端提交（或撤消）当前事务，并将该消息发给服务器。服务器将检查它的一致性约束是否允许处理该事务，如果允许的话就处理该提交。它将该事务写入存储系统之后，就会通知客户端该提交已经处理完成了（或者在本例中也有可能失败）。现在客户端与服务器对该事务的执行结果意见一致。

而如果确认了该提交的消息在到达客户端之前就被切断了，又会发生什么情况呢？此时客户端就不知道该提交到底是成功了还是失败了！2PC协议规定了节点必须等待确认消息到来，以判断事务的结果。如果消息没有到达，2PC就不能正确地完成。因此这不是一种分区容忍协议。真实的系统不可能无限制地等待，在某个点上客户端会产生超时，这使得提交协议停留在一个中间状态。

如果我人为造成这种分区的发生，JDBC Postgres客户端就会抛出类似以下类型的异常：

```
217 An I/O error occurred while sending to the backend.
Failure to execute query with SQL:
INSERT INTO "set_app" ("element") VALUES (?)  ::  [219]
PSQLException:
Message: An I/O error occured while sending to the backend.
SQLState: 08006
Error Code: 0
218 An I/O error occured while sending to the backend.
```

我们或许会将其解读为“写操作217和218的事务失败了”。但是当测试应用去查询数据库，以查找有哪些成功的写操作事务时，它会发现这两个“失败的”写操作也出现在结果中：

```
1000 total
950 acknowledged
952 survivors
2 unacknowledged writes found! ヽ(´ー`)ノ
(215 218)
0.95 ack rate
0.0 loss rate
0.002105263 unacknowledged but successful rate
```

在1000次写操作中，有950次被成功地确认了，而且这950条写记录都出现在了结果集中。但是，215与218这两条写操作也成功了，尽管它们抛出了异常！请注意，这里的异常并不确保这个写操作是成功了还是失败了：217号操作在发送时也抛出了一个I/O错误，但因为客户端的提交消息在达到服务器之前，连接就已经被切断，因此该事务没有执行成功。

没有任何方法能够从客户端可靠地区分这几种情形。网络分区，或者说其实大多数网络错误并不代表失败，它只是意味着信息的缺失。如果没有一种分区容忍的提交协议，例如扩展的3阶段提交协议，是没有办法断定这些写操作的状态的。

而如果将你的操作修改为幂等（idempotent）的，并且不断地进行重试，就可以处理这种不确定性。或者也可以将事务的ID作为事务的一部分，并在分区恢复后去查询该ID。

Redis

Redis是一个数据结构服务器，它通常会部署在一个共享的堆之内。由于它运行在一个单线程的服务器内，因此它默认就提供了线性一致性，所有操作都会以一个单一的、定义良好的顺序进行执行。

Redis还提供了异步式的主-从分发能力。某一台服务器会被选为主节点，它能够接受写操作，随后将状态的改变分发给各个从节点。在这一上下文中，异步方式意味着当主节点分发某个写操作时，客户端请求不会被阻塞，因为写操作“最终”会达到所有从节点。

为处理节点发现（discovery）、选择主结点及故障转移等操作，Redis包含了一个附加的系统：Redis哨兵（Sentinel）。哨兵节点会不断交流它们访问的各个Redis服务器的状态，并且尝试对节点进行升级与降级，以维持一个单一的权威主节点。在这次测试中，我在所有5个节点上安装了Redis和Redis哨兵。起初所有5个客户端都会从主节点n1以及从结点n2至n5上读取数据，接下来我将n1与n2与其它节点进行分区。

如果Redis是一个CP系统，那么在分区发生时n1和n2就会变得不可用，而其它重要组件（n3、n4、n5）中的某一个便会被选为主节点。但事实并非如此，实际上写操作依然会在n1上成功执行，几秒钟之后，哨兵节点会检测到分区的发生，并且将另一个节点（假设是n5）选为新的主节点。

在分区发生的这段时间，系统中存在着两个主结点，系统的每个组件中各有一个，并且两者会独立地接受写操作。这是一种典型的脑分裂场景，并且违反了CP中的C（一致性）。这种状态下的写（以及读）操作不是可串行的，因为根据客户端所连接的节点的不同，它们所观察到的数据库状态也不一样。

当分区恢复之后会发生什么情况呢？Redis之前的做法是将两个主节点永久地保留，这样每次分区都会提升一个新的主节点，并造成永久性的脑分裂。这一状况在2013年4月30日发布的Redis 2.6.13版本中发生了改变，现在哨兵将会让原始的主节点降级为从节点，并销毁进程中有可能超越界定范围的写冲突以解决冲突，举例如下：

```
2000 total
1998 acknowledged
872 survivors 1126 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
50 51 52 53 54 55 ... 1671 1675 1676 1680 1681 1685
0.999 ack rate
0.5635636 loss rate
0.0 unacknowledged but successful rate
```

在总计2000次写操作中，Redis宣称其中的1998次成功完成了。但是在最后的结果集中只有872个数字，Redis将它所声明成功的写操作其中的56%都丢弃了。

这里存在两个问题。首先，我们注意到所有客户端在分区开始后的写操作全部丢失了：（50、51、52、53、……），这是因为当网络分区时它们都将数据写入n1中，而由于n1在之后被降级，因此在这期间发生的所有写操作都被销毁了。

第二个问题是由于脑分裂所造成的：在分区情况恢复之前，n1与n5两者都作为主节点存在。根据客户端所连接

的节点的不同，有些客户端的写操作被保留，而另外的一些客户端的写操作则丢失了。该数据集合中的最后几个数字除以5取模之后都是0或1，这就是来自于那些始终使用n1作为主节点的客户端的操作。

在任何分发模式下，一旦产生故障转移，Redis就不能提供高可用性或是一致性。因此请将Redis作为一种“尽力而为”的缓存与共享堆使用，并且保证如产生任何数据丢失或数据错误都是可接受的。

MongoDB

MongoDB是一个面向文档的数据库，它的分布式设计类似于Redis。在一个分发集中，存在着一个单独的主节点，它接受写操作并异步地将(“oplog”)的操作日志分布给N个从节点。不过，它与Redis有一些关键的不同之处。

首先，Mongo创建了自己的主结点选择与分发的状态机。不存在一个外部的独立系统去尝试观察某个分发集以决定系统的表现。分发集会在内部自行决定哪个节点该成为主节点，何时进行降级，如何进行分发等等。这种方式在操作上更简便，并且避免了各种拓扑问题。

其次，Mongo允许你询问主节点，让它从自身的磁盘日志或者是从节点的响应中确认某个写操作的分发是否成功。虽然付出了增加延迟的代价，但我们就能够更加确保证某个写操作是否成功了。

为了测试，我建立了一个5个节点的分发集，并将n1设为主节点。客户端通过compare-and-set方式对某个单一的文档（MongoDB一致性的基本单元）进行写操作。接下来我将主节点n1与n2从集群的其它部分中分区出去，迫使Mongo在主要组件之中选择一个新的主节点，并将n1节点降级。我让系统在分区状态下运行了一段时间，然后重新连接各个节点，让Mongo在测试结束之前进行重新聚合。

对MongoDB的操作有着数种一致性级别，称为“写关注”（write concern）。目前为止的默认级别是完全不检查任何类型的失败。Java客户端的调用参数是WriteConcern.UNACKNOWLEDGED。很自然，这种方式在分区期间会丢失任意数量的“成功”写操作：

```
6000 total
5700 acknowledged
3319 survivors
2381 acknowledged writes lost! (ﾉ°□°)ﾉ︵ ┻━┻
469 474 479 484 489 494 ... 3166 3168 3171 3173 3178 3183
0.95 ack rate
0.4177193 loss rate
0.0 unacknowledged but successful rate
```

这次尝试中，3183次写操作中的469次，即42%的数据丢失了。

但是，即使选择能够确认数据已经成功地提交到主节点的WriteConcern.SAFE级别，它仍然会丢失大量的写操作：

```
6000 total
5900 acknowledged
3692 survivors
2208 acknowledged writes lost! (ﾉ°□°)ﾉ︵ ┻━┻
458 463 468 473 478 483 ... 3075 3080 3085 3090 3095 3100 0.98333335 ack rate
0.3742373 loss rate
0.0 unacknowledged but successful rate
```

由于分发协议是异步的，写操作会持续地在n1上操作成功，即使n1无法将这些写操作分发给集群中的其它节点。当n3稍后被选为主节点时，它的数据已经不是最新的数据了，因为它和n1的各种写操作没有进行对接。这两个节点会不一致地并存一段时间，直到n1意识到它必须降级为止。

当分区情况恢复后，Mongo会尝试判断哪个节点拥有权威性，很显然，没有一个权威节点的存在，因为两个节点在分区时都接受了写操作。随后，MongoDB会尝试寻找拥有最高optime（每个节点的oplog的时间戳）的节点，然后它迫使旧的主节点（n1）回滚到最近的一次数据相同的时间点，随后重新应用在n3上发生的各种操作。

在回滚过程中，MongoDB会将冲突对象的当前状态的快照写入磁盘上的一个BSON文件中，某个操作者可以在稍后尝试重建文档的适合状态。

这个系统存在着多个问题。首先，在选择主节点的代码中有个缺陷：MongoDB可能会将分发集中某个并没有最高的optime的节点升级为主节点。其次，在回滚的代码中也有个缺陷。在我的测试中，回滚大约只有10%的次数是成功的。在大多数情况下，MongoDB会完全丢弃冲突数据。此外：在回滚过程中并非所有的对象类型都会被完整地记录下来：例如按照设计，超过上限的集合就会丢弃所有的冲突。第三，即使系统运行正常，回滚日志也不足以恢复线性一致性。由于回滚的版本与oplog并没有共享一个定义良好的因果次序，因此通常只有无次序的合并操作（例如CRDT）才能重建文档的正确状态。

线性一致性的缺失也同样发生在FSYNC_SAFE、JOURNAL_SAFE、甚至是REPLICAS_SAFE等级别中，虽然它们能够确保在请求成功之前已经有两个从节点确认了本次写操作。

```
6000 total
5695 acknowledged
3768 survivors
1927 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
712 717 722 727 732 737 ... 2794 2799 2804 2809 2814 2819
0.94916666 ack rate
0.338367 loss rate
0.0 unacknowledged but successful rate
```

在MongoDB的模型中恢复线性一致性的唯一方法就是等待某一数量的仲裁节点返回响应。但是WriteConcern.MAJORITY级别依然不是一致的，它会丢失已确认的写操作并恢复已失败的写操作。

```
6000 total
5700 acknowledged
5701 survivors
2 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
(596 598)
3 unacknowledged writes found! ヽ(´ー`)ノ
(  562 653 3818)
0.95 ack rate
1.754386E-4 loss rate
5.2631577E-4 unacknowledged but successful rate
```

UNSAFE、SAFE以及REPLICAS_SAFE等级别在分区时会丢失任意数量或全部写操作，而MAJORITY级别只会在分区开始时丢失未完成的写操作。当主节点降级时它会为所有WriteConcern请求签名，将每个回复的OK设为TRUE，无论WriteConcern是否已被满足。

此外，MongoDB会产生任意数量的漏报（false negative）。在本次尝试中，有3个未被确认的写操作实际上已

经在最后的数据集中恢复了。至少到2.4.1版本为止，无论在哪一个一致性级别上，都没有办法在分区时防止数据丢失。

如果你需要MongoDB实现线性一致性，请使用WriteConcern.MAJORITY级别。它不会带来真正的一致性，但至少大量减少了写操作的丢失。

Riak

作为Dynamo的一个克隆项目，Riak对分区容忍采取了AP方式，它能够检查到因故产生分歧的历史数据，无论是由于分区还是普通的并发写操作，并且将某个对象的所有有分歧的拷贝都展现给客户端，由客户端决定如何合并这些冲突。

Riak中的默认合并功能是后来者获胜的方式。每个写操作都包含一个时间戳，合并时只保留时间戳最大的版本。如果各节点的时钟是完全同步的，那么该方式能够确保Riak始终使用最近的数据。

即使没有分区及时钟不同步的情况，并发写操作也意味着后来者获胜的策略会导致成功的写操作被默默地忽略掉：

```
2000 total
2000 acknowledged
566 survivors
1434 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
1 2 3 4 6 8 ... 1990 1991 1992 1995 1996 1997
1.0 ack rate
0.717 loss rate
```

在这一示例中，一个情况良好的集群也会丢失71%的操作结果，因为两个客户端几乎是在同一时间对某个值进行写入的，Riak只会简单地选择时间戳更高的那一个，而忽略其它所有操作，而其中就有可能加入了新的值。

人们经常会试图添加一个锁服务以避免并发操作，从而解决这一问题。由于锁必须是线性的，CAP定理告诉我们分布式加锁系统在分区中并非完全可用，但即使它们能够做到完全可用，也无法避免数据的丢失。这里有一个Riak集群，包括读写和仲裁节点，所有的客户端会使用一个互斥体（mutex）自动进行读、写操作。在当分区发生时，Riak丢失了91%的成功的写操作：

```
2000 total
1985 acknowledged
176 survivors
1815 acknowledged writes lost! (ノ°□°)ノ︵ ┻━┻
85 90 95 100 105 106 ... 1994 1995 1996 1997 1998 1999
6 unacknowledged writes found! ヽ(´ー`)ノ
(203 204 218 234 262 277)
0.9925 ack rate
0.91435766 loss rate
0.00302267 unacknowledged but successful rate
```

实际上，LWW会造成边界外的数据丢失，包括在分区发生前写入的信息。这可能是因为Dynamo（在设计上）允许草率的仲裁，因而分区两边的替代结点都能够满足读与写的要求。

我们可以让Riak使用一个严格的仲裁方案，其中使用了PR和PW，只有在某个原始节点集中的一个仲裁节点承认了该操作之后，整个读写才能成功。但如果发生了分区，仍然会有边界外的数据会丢失：

```
2000 total
1971 acknowledged
170 survivors
1807 acknowledged writes lost! (ノ°□°)ノ﹵ ┻┻
86 91 95 96 100 101 ... 1994 1995 1996 1997 1998 1999
6 unacknowledged writes found! ヽ(´ー`)ノ
(193 208 219 237 249 252)
0.9855 ack rate 0.9167935 loss rate
0.00304414 unacknowledged but successful rate
```

Dynamo的设计初衷就是尽量保留所有的写操作。即使当某个节点无法将某个键分发至主节点，它可能会返回“PW值未被满足”的信息，它仍然有可能能够写入某个主节点，或者任意数目的替代结点。在读取修复过程中，这些值仍然会被进行交换、被视为冲突值，以及用以丢弃旧值的时间戳——这意味着所有的写操作都来自于集群的一边。

这也意味着少数组件的“失败”写操作会破坏所有多数组件的成功的写操作。

在AP系统中是有可能使用CRDT来保留数据的。如果我们将分发集作为数据源，将联合操作作为合并方法，我们就能够在即使遇到人为的分区时也能够保留所有的写操作了。

```
2000 total
1948 acknowledged
2000 survivors All
2000 writes succeeded. :-D
```

这种方式并非串行一致，而且也不是所有数据结构都可以表示为CRDT的。 此外它也不能够防止漏报。Riak会产生超时或者报告错误，但它能够保证对已承认的写操作进行安全收敛。

如果你正在使用Riak，最好使用CRDT或者使用一个你自己编写的合并函数。LWW只在某些情况下是合适的，在其它情况下最好避免使用它。

评估结果

我们已经看到分布式系统在分区情况下会产生一些预计之外的表现，但这些预计之外的错误本质上是由许多事实共同决定的。数据丢失与不一致性发生的可能性取决于你的应用程序、网络、客户端拓扑结构、运行时间、错误的本质等等。我在这里不打算告诉你去选择哪一个具体的数据库，而是鼓励你仔细考虑一下你所需要的各种不变性和你能够承受的风险，然后再进行相应的系统设计。

设计的一个关键部分是对其进行评估。首先为系统建立边界，即它与用户、网络以及其它服务进行交互的边界，并决定这些边界中有哪些需求是必须保证的。

接下来编写一段程序，它从边界之外对系统发起请求，并对外部观察到的结果进行评测。记录下HTTP请求是否返回200或503等代码，或者是在每个点上的post操作所返回的一系列反馈。在运行该程序的时候有意造成故障：例如干掉某个进程、卸载某个磁盘或将某个节点用防火墙进行隔离。

最后，检查日志以确认系统的各种保证已满足。比方说，已接受的聊天信息至少应该有一次被传送至它的接收房，因此请检查信息是否已被正确的传输了。

最后的结果可能会令人感到惊讶，请根据该结果对你的系统设计、实现以及各种依赖进行进一步的调查研究。对系统的设计应该持续地对它的关键安全保证进行评估，就像你在进行性能检测一样。

心得

即使你不打算使用MongoDB或Riak，从以上示例中你也能够学习到一些东西。

首先，客户端是分布式系统的重要组成部分，而不是客观的观察者。而网络错误代表了“我不清楚”，而不是代表“失败了“。在你的代码与API中请明确地区分成功、失败与不确定的状态。考虑将一致性算法扩展到你的系统边界之外：例如检查TCP客户端的Etag或者使用Vector Clocks算法，或者将CRDT扩展到你的浏览器上。

即使是一些著名的算法，例如两阶段提交，也会存在漏报等问题。SQL事务性一致性产生在几个级别上，如果你使用了更强的一致性级别，请记住冲突处理是关键所在。

某些问题是难以良好地处理的，例如当故障转移时维护一个权威的主节点。一致性是数据的属性，而不是节点的属性。在系统中要避免将节点状态的一致假设为其代表数据的一致性。

挂钟（wall clock）只能够保证在死锁发生时的响应性，即使如此，它对正确性也没有产生积极的保证性。在这些测试当中，所有的时钟都与NTP进行了同步，但仍然发生了数据丢失的情况。而如果某个时钟没有被同步、或者某个节点被暂停了一段时间，会发生更加糟糕的事情。请在你的数据中使用逻辑时钟。对那些依赖于系统时间的系统需要保持一定的怀疑性，除非你的节点中使用了GPS或者原子性时钟。另外请一定要评测你的时钟准确性。

如果正确性非常重要，那请你使用某种经过实际检验与论文审阅的技术。在理论性正确的算法与实际的软件之间存在着巨大的鸿沟——尤其是考虑到延迟的问题。基于某个正确算法的实现，哪怕有缺陷存在，通常也比那些基于某个糟糕算法的完美实现来得更好。因此缺陷是可以修复的，但重新调整设计则要难得多。

请为你的问题空间选择正确的设计。你的架构中的某些部分会对一致性要求很高，而某些部分可以对串行性稍作牺牲，只要保证正确性就行了，例如CRDT就是如此。有些时候，即使是失去整块数据对你来说也是可以接受的。性能与正确性的选择经常要进行各种折衷：需要你进行思考、测试并找到最合适的方式。

用某些规则对你的系统进行限制能够更容易地实现安全性。不可变性就是一种非常有用的属性，并且它可以和某个可变的CP数据存储系统共同使用，以实现一个强大的混合型系统。尽量使用幂等的操作：它允许你进行各种队列操作与重试。如果可行的话，请把步子迈得大一点，去完全的使用CRDT的功能。

在使用类似于MongoDB这样的数据库时，如果要防止写操作丢失，只能以严重的延迟作为代价。如果使用Postgres说不定还会快一点。有时候，采购更可靠的网络以及强大的基础设施比起横向扩展来说会更便宜，但有时也并非如此。

分布式的状态是个难以解决的问题，但只要稍稍投入一些精力就能够极大的增加系统的可靠性。关于网络分区所带来的影响的更详细说明，包括各种产生环境中的故障，请查看这里。

关于作者

Kyle Kingsbury是一位来自于[Factual]的工程师，他的个人网站请见此处。此外，他也是[Riemann]的作者，这是一个开源的事件驱动的监护系统。以及[Timelike]的作者，这是一个在网络情况模拟的实验项目。他来自于加利福尼亚州的旧金山市。对于计算机是如何运行的，他至今毫无头绪。

查看英文原文： Jepsen: Testing the Partition Tolerance of PostgreSQL, Redis, MongoDB and Riak