基于 Paoxs 算法的分布式数据库开发
以树洞为例

算法设计与分析小班 **project** 报告

孙睿阳[1]，孙佳琪[2]，伊清扬[2], and 吴芊染[3]

[1] 地球与空间科学学院
[2] 信息与科学技术学院
[3] 元培学院

Abstract

Paxos 算法由 Leslie Lamport 于 1998 年提出, 其核心为通过分布式节点之间的消息传递, 在可能存在网络延迟、重复、丢失的情况下快速且正确地就某一个值达成共识. 该算法以高度抽象的方式描述了分布式系统中实现一致性的方法, 但其具体的实现方式存在差异性. 本文价绍了我们以 Lamport 最初设计的，没有其他优化 paxos 算法建立的分布式数据库系统，同时我们提出了一个高效可行的多轮 paxos 算法，并进行了仿真实验，验证了我们数据库系统的可行性和效率.

1 介绍

Paxos 算法得名于其原始论文中作者 Leslie Lamport 借以比喻算法运行过程的希腊小岛 Paxos. 当此文于1998年面世时, 由于其叙述方式为将此算法中多个进程的运行过程以 Paxos 小岛中议会决议的方式比喻写出, 相对较为晦涩, 因此并未得到广泛的理解. 在此之后, 于 2001 年在 PODC 会议上, 作者应邀将此算法以更易懂的方式写出 (Lamport,2001), 使得这一算法的影响面显著扩大.

随着 21 世纪开始互联网的迅速发展, 单一服务器已经无法满足多数情况下的访问需求, 因此中等规模以上的网站都需要采用集群的方式响应用户请求, 使得分布式一致性算法成为必不可少的需求. 应这一需求, 出现了多种基于 Paxos 算法的工程实践, 包括谷歌的分布式系统 Chubby(Burrows,2006), 开源的 Zookeeper(Hunt et al.,2010) 和腾讯用于微信消息存储的 PaxosStore(Zheng et al.,Zheng et al.) 等. 除此之外, 由于 Paxos 算法尚存在一些问题, 如状态数量多, 达成共识的核心过程存在较强耦合以及不能处理恶意节点等, 一些基于 Paxos 的改进算法也被提出并得到了广泛的使用, 如 Raft 算法 (Ongaro and Ousterhout) 和 Zab 算法 (Medeiros,2012). 同时, 在 Paxos 之外, 近

年还出现了一些以完全去中心化为主要需求的算法, 其核心为区块链技术, 如以工作量保证一致性的比特币 (Nakamoto,2008)、以质押的方式保证一致性的 PP 币 (King and Nadal) 等.

本文以 Leslie Lamport 于 (2001) 年撰写的简化叙述 [6] 为基础, 结合关于 Paxos 算法核心内容的简单工程实践, 对 Paxos 算法的运行过程以及处理网络延迟、错序及丢失的方式进行阐述, 并对 Paxos 算法所不能较好处理的问题进行了进一步的讨论.

我们在本文中首先先在小节 2介绍了分布式数据库的相关工作，在小节 3介绍了我们系统的基本架构，在小节 4介绍了我们如何实现这一系统，在小节 5重点介绍了我们设计的多轮 paxos 算法，在小节 6部分介绍了我们如何对于系统性能的衡量进行实验和实验结果，在小节 7介绍了该项目未来可以继续探索的方向，在小节 8对我们的工作进行了总结。

2 相关工作

Paxos 算法运行的核心为三种类型的节点, 即 proposer、acceptor 和 learner. 其中, proposer 负责产生新的提案内容, acceptor 接受提案值并进行多数投票表决, 而 learner 则计票并产生共识. 当产生一个共识时, 则将其记录, 并广播通知所有 proposer 和 acceptor. 作者论文中并未给出这一共识值的细节, 但是一般来讲这一共识值为数据库中的操作原语, 例如写入、删除、修改等 [2].

Paxos 运行的过程简单分为三部分, 提案、投票和表决 [7]:

1. proposer 选择一个包含唯一标识的提案编号 n, 并通过 prepare 广播. 接收到这个信息的 acceptor 若已经响应编号更大的信息则选择拒绝, 若已经对先前一个编号更小且尚未达成共识的提案作出响应则回复 prepared(n',v'), 否则回复 prepared(0, null).

2. 当多数 acceptor 均接受这一信息, 则进入第二阶段. 如果收到的回复为 prepared(n',v'), 则将提案值修改为 v', 否则随便确定一个 v. 之后广播 propose(n,v), 接收到此信息的 acceptor 选择接受或者拒绝.

3. 如果多数派接受这一 propse 信息, 则形成共识, learner 记录下 value 并广播达成一致的 decided 信息.

这一算法为理想情况下分布式系统的工作方式, 但是实际情况下需要考虑多种其他因素. 关于这一方面, 已经有相关工作如下:

分布式服务器的一个重要需求为不停机热更新 [7]. 在这一情况下, 会有处于任意状态的节点退出表决系统, 并有新的节点以刚完成初始化的形式加入. 此处的一个隐患为假如正在进行表决的旧节点已经退出, 而新的节点还没有收到 decide 信息, 因此无法完成表决 [7]. 此处的解决方式为借助状态机的确定性, 即两个相同的状态机以相同的状态开始, 并接受相同的请求序列, 则会进入相同的最终状态, 因此可以通过 RPC(Remote

Procedure Call) 实现状态复制. 此处为保证确定性, 在可能出现分歧状态的时候 (如文件系统的时间戳), 让一个节点确定这一值, 并让其他节点按照这一值执行.

腾讯的开源 Paxos 库 PhxPaxos 则解决了原始的 Paxos 算法中一次提案过程中只能产生一个值, 工程上实用性不高的问题. 其解决方案为在一个物理节点上可以同时运行多个 Paxos 算法的实例, 以同时确定多个确定值 [11]. 这一库所解决的另一个问题为 Paxos 中确定的值无法进行修改, 因此难以用于实现服务. 这里由于大部分存储系统, 如 LevelDB, 都是以 AppendLog 的形式, 确定一个操作系列, 而后需要恢复存储的时候都可以通过这个操作系列来恢复, 而这个操作系列, 正是确定之后就永远不会被修改的. 因此将操作序列作为真正确定的值可以使得 Paxos 算法得以应用于服务.

目前最广泛使用的 Paxos 库为由谷歌公司开发的 Chubby 数据库系统 [2], 结构如图 1所示. 其实现方式大量考虑了生产环境中可能出现的突发状况如磁盘故障等. 其解决磁盘故障的方式需要在文件中存储文件目录的校验和, 如果校验和变化则表示磁盘出现故障, 其修复主要方法是参与 Paxos 但是不投票, 利用 catch-up 机制复制文件到本机, 不给与 promise 或者 ack 回复, 直到完成了所有文件的修复为止. 此外, 在 multi-paxos 中为避免活锁问题加入了主节点 master, 但是由于网络波动的原因 master 有可能无响应, 需要选举新的 master, 选举新的 master 的序列大于旧的序列, 旧序列即使连上了, master 也会自我解除权限, 这就避免了多个 master 造成不一致的问题. Chubby 中的这一实现方式与原始论文相反, 因为断线的 master 更有可能在未来的时间点再次断线, 此时更为保险的办法为自我解除, 而非在重新连接之后再次成为 master.

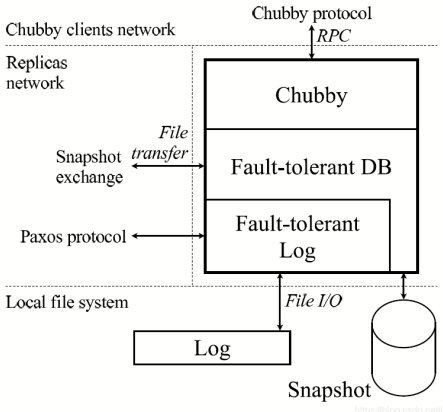

图 1: 谷歌的 Chubby 数据库系统结构. 注意到其同样使用 Paxos 算法记录操作序列而非操作本身

Paxos 算法的工程实践中所需要解决的问题远远不止这些, 还包括很多性能上的考虑因素, 如减少磁盘读写量、处理创建/销毁快照、减少内存消耗、状态机序列销毁等, 而关于其中每一个问题都有大量的工作致力于寻求高效率、低延迟的解决方案. 受篇幅所限, 本文只对上述三个工程实践内容进行了简要介绍.

3 系统架构

3.1 节点间的架构

我们假设了我们的系统的不同节点可以分布在世界各地的多个数据中心，以此来为不同地区的用户提供尽可能低延迟的服务。当然这并不是我们项目中的业务目标，也就是树洞需要达成的，但如果能确保这一点，将意味着我们的数据库系统在大尺度的地域分布时仍有不错的性能和容灾能力。

图 2展示了存在三个数据中心时，节点的结构情况。每一个数据中心之间的关系都是平等的，它们之间会通过 paxos 算法达成每一条命令的共识。客户端会选择一个相对于客户端而言延迟最小的数据中心进行访问，因为 paxos 算法达成的强一致性，客户端可以从它所连接的数据中心中任意的读取数据，而不用担心因为数据中心的不一致导致数据不一致。同时客户端也可以给它所连接的数据中心发送写入数据的指令，该数据中新则会通过 paxos 算法与所有的其他数据中心达成共识后，执行该指令。

在同一数据中心下，为了进一步增加数据的安全性和访问效率，还应当可以设置存在数据中心内网下的集群，该集群下应当可以实现数据冗余和数据分区存储，但由于时间原因，这一部分我们并没有实现，具体会在小节 7未来工作中具体介绍。

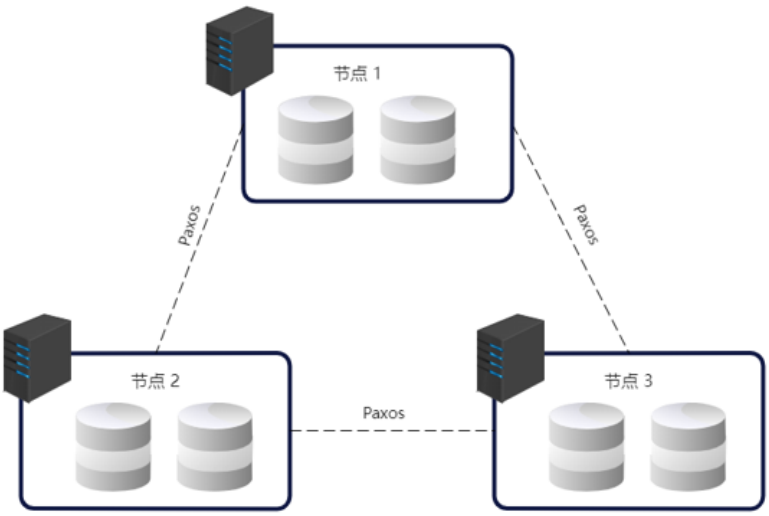

图 2: **节点间的架构**. 三个数据中心时的节点架构情况，每个节点的地位平等，客户端可以发给某个节点读或者写的指令，对于读的指令，节点会当场返回结果；对于写的指令，节点会通过 paxos 算法与其他节点达成共识

3.2 节点内部的结构

图 3展现了我们系统的单个节点的内部架构，它包含了四层。网络层（Net Layer）暴露在外面处理与客户端和其他节点的通信，它会对不同的消息类型进行分流之后交给下一层进行处理。控制层（Control Layer）会进一步处理网络层传递的消息，并控制

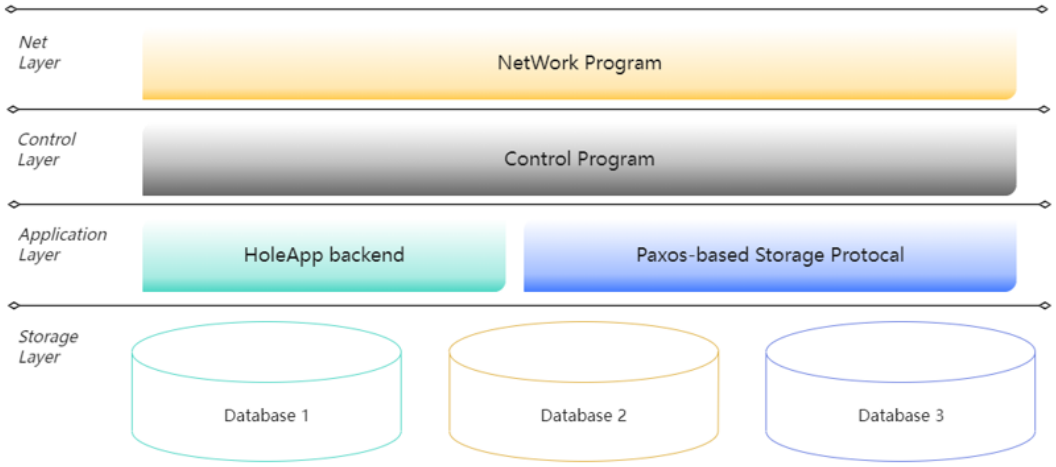

图 3: **节点内的架构** 这是节点内部的系统架构，它分为网络、控制、应用和存储四层

Paxos 算法的开始。应用层包（Application Layer）含两个部分，其中树洞后端（HoleApp Backend）会处理读的指令，它会从下一层（存储层）中的数据库读出数据后再通过网络层想客户端返回数据。另一部分是对于基于 Paoxs 算法的存储协议的实现。最底层是存储层（Storage Layer），由数据库的存储引擎组成。

工程角度上看，这样设计系统的架构的缺点是树洞的后端和存储部分并不是解耦的，这并不利于将我们的系统移植到其他的项目上。但事实上应用层中的树洞后端需要编写的核心代码只是处理读指令然后返回数据。因此在实际应用中，应用后端的设计有两种思路，一种是在树洞后端的部分仅仅完成一个忠实的对于所有读指令都能正确执行的代码，将业务逻辑的部分放到一个独立的后端中，而这个后端会作为我们系统的节点的客户端。这样的好处是后端的业务逻辑代码会和分布式数据库的代码完全解耦，但缺点是增加了一层消息传递，用户访问数据的速度可能会下降。另一种思路。是将业务逻辑代码放到树洞后端编写，但这样就要求在新的项目上，开发者需要重写几个我们提供的抽象接口。[1]

4 实现

4.1 基于 Http 的消息传递机制

我们在具体实现系统时，使用的消息传递协议是 http 协议。节点之间会使用 http 协议进行通信。这样的好处有二，首先 http 协议的使用可以分为两部分，发送端可以使用 python 的 requests 包一行代码完成以此消息传递，接收端可以使用成熟 nginx+uwsgi 的方案搭建 http 服务器，省去了重复造轮子的麻烦。其二，http 的下层的传输层协议 TCP，具有首部校验、丢包重传、接收确认的机制，这一可以用来保证消息传递的非拜占庭性质。

[1] 为了保证通用性，我们在小节 4介绍的实现采取的是第一种思路

处理节点与客户端的通信时，我们给客户端提供了可供调用的网络接口，客户端试图读取一条数据时，可以通过 GET 方法，向节点发送指定参数。当客户端需要发送一条写入指令时，可以通过 POST 方法向节点发送数据 json 格式的指令，节点会在之后的某一轮 paxos 算法中通过该指令，并执行。

处理节点与节点之间的消息时，我们的系统了采取 json 的数据格式，节点与节点之间的通信通过 POST 方法完成，我们为 paxos 传递的消息定制了适用的消息格式。代码4.1展现了我们定制的消息格式的细节。

```
class NodeMessage:
    type = None  # 消息类型 有 'prepareRequest', '
        prepareRespond', 'acceptRequest', 'acceptRespond',
         'BroadcastAccept'
    source = None  # 发送者的地址
    target = None  # 接收者的地址
    targetAgent = None  # 需要处理该消息的角色
    turn = None  # 当前处理的轮次
    number = None  # 提案编号
    value = None  # 提案内容，应该是 Ins 的 Dict 格式 or
        json 格式
    promise = None  # acceptor 做出的 promise
    accept = None  # acceptor 是否接受提案
```

Listing 1: 节点间通讯消息格式

4.2 基于 redis 的消息队列和分布式锁

我们的系统在实际设计中，存在大量的并行策略，实际运行中会有大量的进程、线程，甚至不同的部分都会拥有不同的程序入口。因此保证系统的高效运行和线程安全，我们使用 redis 作为单个节点中实现进程同步的重要组件。

redis 提供的 List 和 key-value 的数据类型很适合用来实现消息队列和分布式锁。使用 redis 作为中间件的优点有三，其一是上面提到了，作为拥有多个程序入口和多个进程、线程的系统而言，python 或其他语言提供的消息队列和锁是不够用的；其二，redis 是及其高效的内存形数据库，读取写入性能对提高系统效率也有帮助；其三，因为不依赖程序语言本身提供的线程同步工具，使得系统的某些部分也可以用其他语言实现，并直接替换之间的部分，这可以保证系统的可移植性。

4.3 使用 docker 容器的系统实现

我们在实现系统时使用 docker 的容器来运行系统的不同部分，并将一个节点中的不同部分编写到同一个’docker-compose.yml‘文件中，节点支持一条命令（’ docker-compose up ‘）进行启动。

使用 docker 的好处有三。其一，docker 作为一种虚拟化技术，对于性能的要求足够低，支持我们的系统可以在单机上完成测试。其二，docker 的打包能大大提高系统启

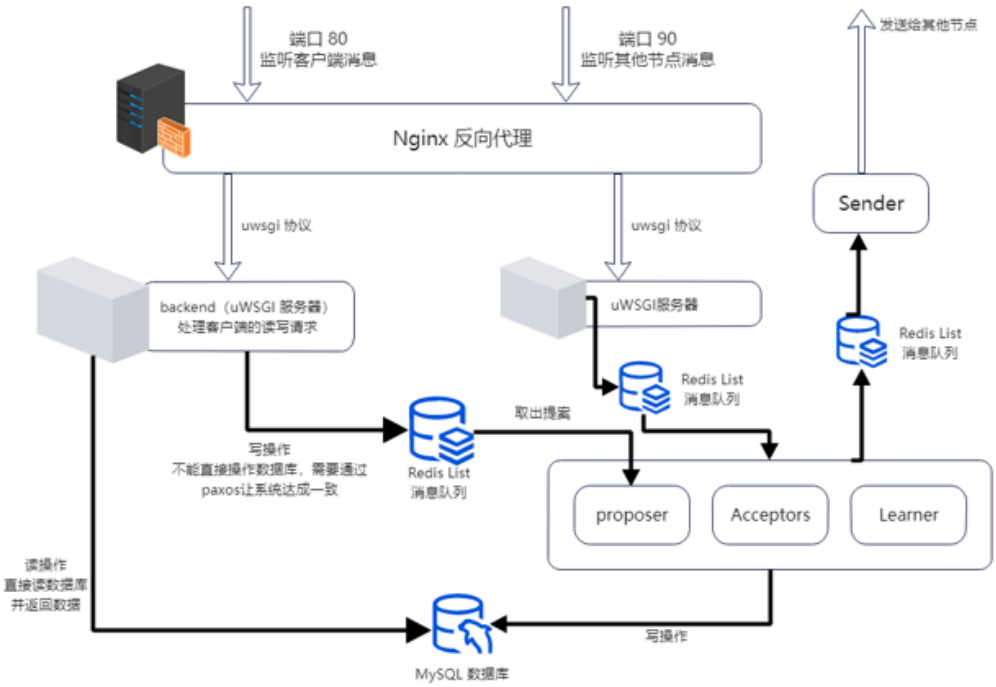

图 4: **系统实现结构**

动的便利，做到即插即用。其三、docker 提供的 network 可以很好隔离与连接不同的容器。

图 4展示了系统实现的具体框架。其中 nginx 服务器会充当网络层，并通过不同的端口隔离开客户端的请求和节点之间的消息。nginx 是支持高并发的 http 服务器组件，消息传递的高并发可以保证系统运行的高效性；对于一个客户端的请求，nginx 服务器会把该请求交给 backend 的 uwsgi 服务器处理，如果该请求是一个读取指令，uwsgi 服务器会直接访问本地的数据库，并把结果返回给客户端，如果该请求是一个写指令，backend 则会把该指令写入到存放指令的消息队列中，交给之后的 paxos 算法进行处理；对于节点之间的消息，nginx 服务其则会将消息传递给处理节点信息的 uwsgi 服务器，uwsgi 服务器会根据消息中的 target 和 targetAgent 字段将消息分发给相应的 agent 进行处理。paxos 的部分会根据指令和消息的消息队列来进行新的 paxos 轮次或者使得旧的 paxos 轮次继续运行，完成的 paxos 轮次会将得到的指令按照轮次的顺序以此执行。对于 paxos 中的 agent 需要传递消息，我们为 agent 设计提供了 sender 的消息队列，当某个 agent 需要向其他节点发送消息时，它可以在 sender 的消息队列写入消息，之后 sender 会采用多进程的策略并发的将消息发送出去。

我们的系统实现中，为消息传递进行并行的处理，这是因为 http 协议会维护一个长连接，只有当消息传递完成后，长连接才会断开。因此消息传递如果使用依次发送的非并行策略，则会使得系统的大量时间花在等待消息传递完成上。采用并行的策略无疑会大幅度的提高系统的运行效率。

5 多轮 Paxos 算法

5.1 概念

在多轮 paxos 算法中，每个节点需要处理多个轮次的指令选取。对于所有节点的同一轮次，每个节点都同时负责 proposer，acceptor 和 learner 三个角色。同一节点间不同轮次没有交互，不同结点间只有同一轮次的进程之间有交互并形成一个完整的 paxos 算法。

每个轮次的 paxos 算法都会分别选择一条指令作为该轮次的结果。所有轮次的 paxos 算法选中的指令按照轮次号排序得到最终的指令序列。这个指令序列应该记录在每个节点上。

5.2 整体结构

在我们实现的多轮 paxos 算法中，每个节点维护一个进程池。进程池中的每个进程对应一个轮次的 paxos 算法中的一个节点。每个进程又开由 3 个线程组成，分别处理 proposer，acceptor 和 learner 的事项。

信号传输在节点之间发生，具体的信号处理由每个进程下的线程执行。

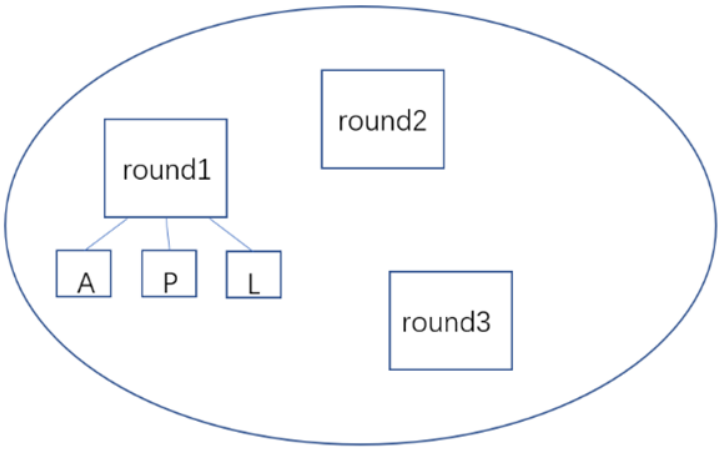

图 5: 多轮 paxos 算法整体架构

5.3 信息处理

多轮 paxos 算法的信息处理有以下几种方式：1.proposer 向 acceptor 发送请求。当节点收到另一个节点发来的 proposer 请求后，会从自己的进程池中查找对应轮次的进程，并把请求交给该进程处理。该进程收到指令后，会以 acceptor 的身份处理信号，给出答复；2.acceptor 向 proposer 回复。acceptor 所在节点会将信息发给 proposer 所在节点，proposer 所在节点根据轮次号找到对应进程，发送接收回复的指令，该进程下的 proposer 线程处理指令；3.learner 广播。当一个节点的某一轮次进程确定了该轮指令

的具体内容，这个进程下的 learner 线程会把指令广播给所有节点。对于每个接收到指令的节点，都会通知进程池内对应轮次的进程，该进程的任务完成，从进程池内剔除。同时，这条被确定的指令会写入到每个节点对应的数据库中。此时进程池会扩充更高轮次的进程以处理后面的指令序列（如果有）

5.4 paxos 算法

在 paxos 算法中，每个提案都需要给出提案编号，当编号不够大时会采用更高的编号。我们的具体实现是将客户端指令的时间戳的相反数作为提案编号。这样做可以保证提案编号是递增的，同时越靠前的提案就越有机会被选中。在我们的算法设计中，客户端可能连接任何一个节点作为 proposer 进行 paxos 算法。因此可能存在两个客户端的指令在同一轮 paxos 算法中。由于 paxos 算法只会选择一条指令，另一条指令会被保留至后续的轮次执行。由于越靠前的指令的时间戳的相反数越大，被拒绝次数更多的指令更有可能被选择，这样做的目的是防止一条指令一直不被选中。

由于节点间通信时间有差异，paxos 算法不一定会选择最早被提出的指令。这有可能导致先写入再删除这一需要严格时序先后的指令出现问题。我们的设计是删除指令需要先查找对应数据是否存在，因此如果删除指令在写入指令之前被 paxos 算法选中，会返回“不存在这条数据”的错误，这样可以避免数据库错误。

5.5 可能存在的问题

上诉多轮 paxos 算法仍存在部分问题：考虑一个节点，如果节点由于某些错误下线，后续重新上线时，他的进程池中某一轮次号的 paxos 可能已经被其他节点决定而抛出进程池，而该进程因为没有接收到 learner 的广播而一直卡住。

对此，朴素的解决方案是让每一个节点间隔固定时间向其他节点询问某一轮次的进程是否已结束，如果是这样，那么需要从其他节点处备份最新的指令序列，并更新自身进程池。由于我们设计的框架的可用性，无法保证算法强一致性，在部分时间访问不同节点可能得到不同数据。

6 评估

6.1 实验装置

我们通过在一台主机上建立多个 docker 容器来运行仿真实验。实验中的主机配置是 Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz，携带 32GB DDR4 内存和足够的存储。实验中，最多同时运行 5 个节点，并在主机上检测资源占用，确保系统资源并没有饱和。

同时为了尽可能模拟显示的网络情况，我们编写了一个代理网络。每个节点会向代理网络发送消息，而不是向目标节点发送消息。代理网络会根据我们的延时设置和丢包模拟设置进行相应的操作之后，再把消息转发给目标节点。

6.2 数据库效率测试

我们为测试数据库效率定义了一个读写效率的物理量：每 100 次读/写所需时间。效率测试分为两个部分，即读测试和写测试。

因为读取数据时，客户端只需要访问某一个节点的数据库，因此读性能一定与节点数量无关。我们编写了一个测试脚本，将向某一个节点无间隔的发送 100 次读取数据的指令。记时将从发送指令开始，到取回所有数据结束。考虑到客户端和节点之间的通讯应当存在延时，我们通过手动设置代理网络的延时时间，来进行测试。

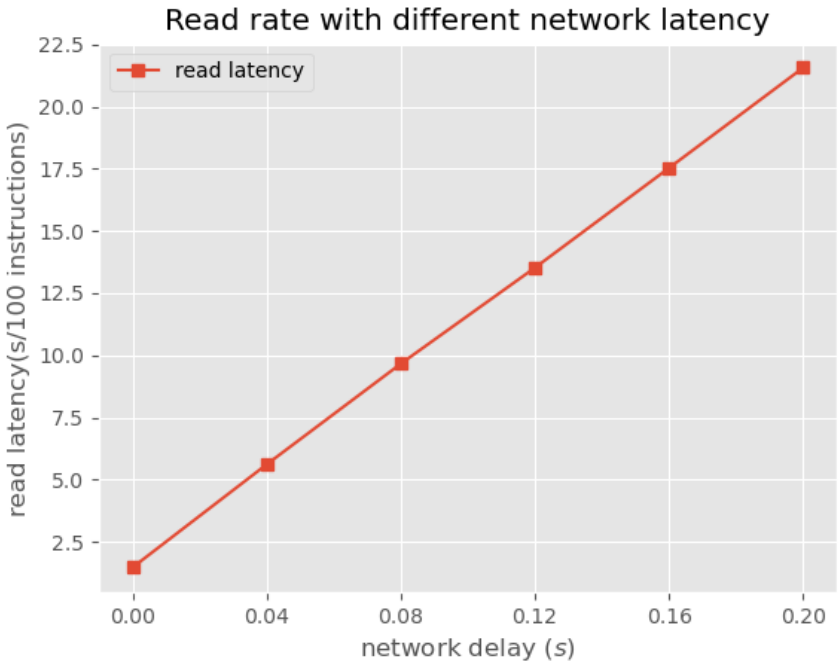

图 6:

图 6展示了不同的代理网络延迟下的读取性能。如果不使用代理网络增加延迟，读取性能可以达到 $1.45s/100insinstructions$，如果客户端是用来完成业务逻辑的程序后端，那么这个速度会比较接近真实情况下的速度。如果客户端是用户的前端部分，可能比较接近正式情况的延迟是 40 ms 到 80 ms，这样的读取性能可以达到 $5s/100insinstructions$ 到 $10s/100insinstructions$，相对而言这是一个比较慢的读取速度。但是其主要愿意是测试脚本对于读取数据没有并行处理，测试脚本会在完成一条数据读取之后才会开始下一条数据的读取。因此，如果我们能为我们系统的设计并行读取的存储的引擎的话，读取性能会有很高的提升。

对于写入性能，则要考虑到节点数量的影响。因为对于每一个写入指令，节点之间需要通过 paxos 算法达成共识。我们分别测试了，在代理网络不设置延迟的情况和模拟数据中心在较大地域尺度上分布的情况（也就是每个节点之间会有 40 毫秒的延迟），系统的写入速度与节点数量的关系。

需要明确的是我们采取的测试方式是编写一个测试脚本，该脚本会随机生成 100 条假数据，然后随机向不同的节点发送写入请求，我们会在系统收到第 1 条写入指令之后开始计时，到完成 100 条指令写入之后停止计时，以该时间衡量系统的写入性能。

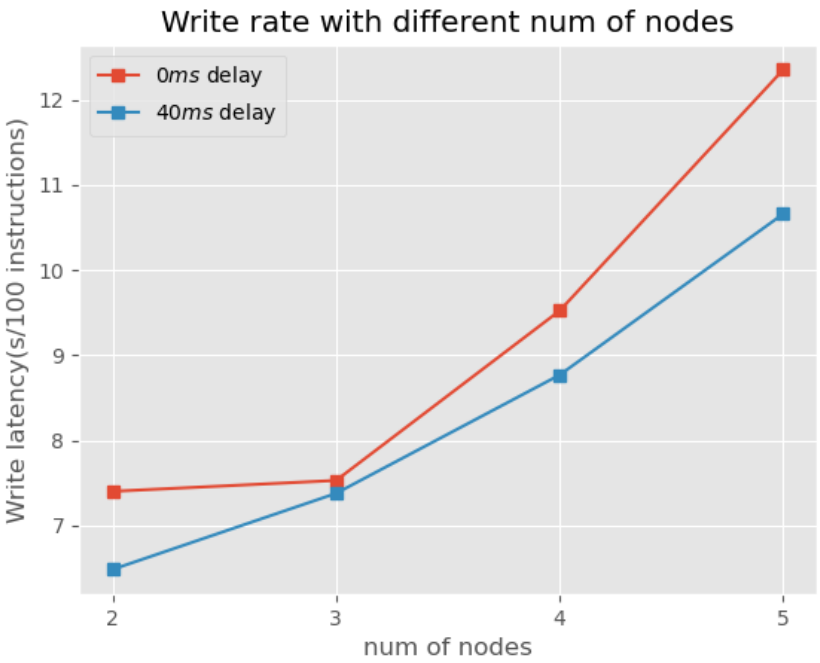

图 7: 不同延时下，系统的写入效率随节点数的变化。可以看到随节点数的增加，写入效率确实会降低。这是因为节点之间达成共识时，节点数越多，消息传递的次数就越多。

图 7展示了不同延时下，系统的写入效率随节点数的变化。可以看到随节点数的增加，写入效率确实会降低。这是因为节点之间达成共识时，节点数越多，消息传递的次数就越多。明显的，节点数与消息传递数量的关系应该比线性增长的要快。而在我们的实验中并没有明显体现这种比线性更快的增长，说明的我们设计的并行消息发送机制的有效性。

同时也要考虑的是代理网络的模拟延迟对系统写入性能的影响。我们以三个节点的情况为例，测试了在不同的延迟下系统的写入性能。图 8展示了系统有三个节点，设置不同的代理网络延时时，系统的写入性能。对于该数据的解释是，当延时较小的时候，系统的耗时依赖于 paxos 算法的计算时间；而当延时较大时，系统的耗时基本由网络情况限制。结果上，系统即使在 200 ms 的延迟下，系统也可以达到 20 s 完成 100 次数据写入。

6.3 容错率测试

paxos 算法是在非拜占庭情况下有效的共识算法，也就是消息可以丢失或多次发送，但是不会出现消息发送错误的情况。因此我们让代理网络可以以某个较小的概率不转发某一条消息，来模拟真实网络当中存在的丢包情况。

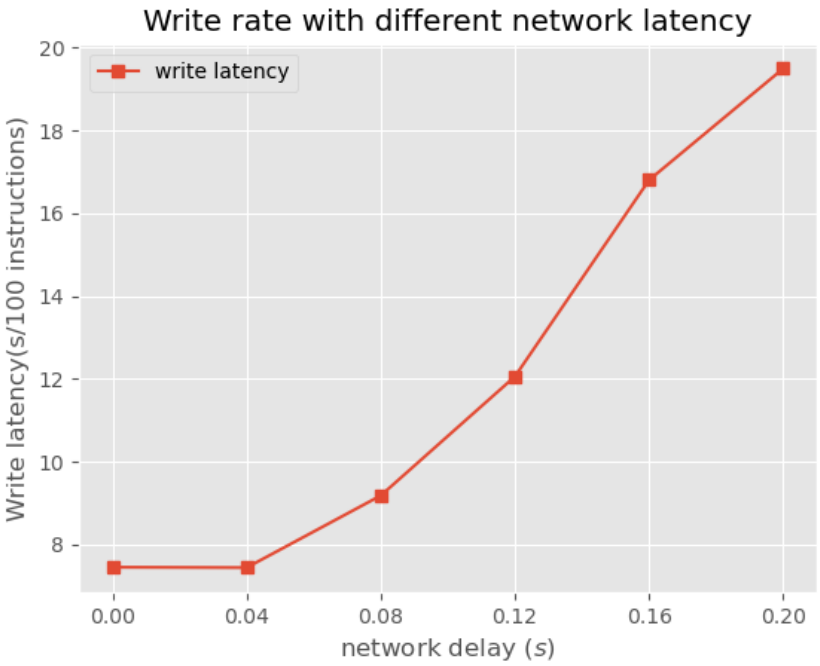

图 8: 系统有三个节点，设置不同的代理网络延时时，系统的写入性能。对于该数据的解释是，当延时较小的时候，系统的耗时依赖于 paxos 算法的计算时间；而当延时较大时，系统的耗时基本由网络情况限制。结果上，系统即使在 200 ms 的延迟下，系统也可以达到 20 s 完成 100 次数据写入。

我们的测试结果表明，系统对于消息发送错误的容忍度，随着节点数量的上升而降低。甚至在只有两个节点的情况下，系统的容错率极低。这是容易解释的，因为 paxos 算法达成共识，极度依赖于在系统的节点中的大多数节点同意某一个提案，而对于两个节点的情况，它们本身才能构成最小的大多数节点，因此 paxos 算法会退化成两次握手的网络协议算法，这就会导致某一条消息的发送失败，使系统在某个 paxos 轮次不能达到最终的结果。而当节点数量增加时，最小的大多数节点需要的节点数量占比降低，消息冗余度提高，某一条消息的发送失败使系统失败的概率也会降低。

7 未来工作

至此，我们已经完成了一个基于多轮 paxos 共识算法的的树洞后端数据库的设计和测试。这个数据库系统可以同时接收多个 client 向不同服务器发送的消息，这些服务器之间通过 paxos 共识算法同步数据库，从而在设备运行稳定、网络环境良好的情况下满足一致性、可用性。如此一来，相较于单一存储的树洞后端，我们的系统具有更大的访问带宽、更高的安全性（safety）和更高的效率。当然，我们的系统作为具有分层结构的后端，也有很好的扩展性，在未来可以进一步优化空间。

首先是一致性的保证。事实上，这个系统仍然有可能不满足强一致性。我们很容易假想一个例子：在第 n 轮 paxos 的进程组达成共识的时候，一个服务器 s 掉线了（可能是由于服务器故障，也可能是由于网络分区等原因）。在过了一段时间后，这个服务器重新启动，而其他服务器在得知达成共识后都已经结束了第 n 轮的进程。因此，服务器 s 不会再收到第 n 轮的任何消息，从而无法得知第 n 轮的结果，其数据库和其他正常结束轮次 n 的服务器之间产生差别：第 n 个指令没有被执行。

因此在改进的过程中，我们必须考虑在服务器出现故障时，从其他服务器获取信息（在实际使用中，这些服务器大部分情况下是一致的），更新数据库。在上述例子中，根据 paxos 最原始的设计，如果这个服务器是 leader，那么它可以让 proposer 发送一个 prepare request，来确定有没有达成共识。而此时我们也采用类似的思路，当服务器通过某种方式得知自己丢失了某些轮次的消息时，它有一些可能的恢复数据的方式：1) 向其他服务器直接拷贝数据库中的数据，2) 向其他服务器寻求缺失轮次的指令。我们的理想状态是，这种恢复可以独立于当前正在执行的 paxos 算法。也即是，在服务器 s 恢复数据的过程中，它仍然可以参与更高轮次的 paxos 共识算法中，得到对数据库的操作指令。这样可以确保这个服务器仍然可以正常运行，而不是不断地从其他服务器直接拷贝数据或指令（这并非共识算法的初衷）。

第二项未来的工作是增加安全性。我们采用了商业 SQL 数据库，而在 SQL 数据库中有一种攻击方式，称为 SQL 注入攻击。而言之，是通过输入特定的字符串作为 SQL 语句的参数，使得 SQL 语句在编译过程中，执行设计者预期以外的命令，从而获取表头信息、表格内容等关键数据。这种注入是对安全性的很大挑战。在我们的系统中，只对前端的特定字符进行了识别。事实上还有更多攻击的模式，需要有针对性地进行防御。

更进一步地，我们的数据库可以广泛应用于其他场景。“树洞”实质上只作为贴合同学们校园生活的一种数据结构载体。而事实上，有更多的应用场景，如记录交易、聊天室等，都可以用这种数据管理系统提高数据库性能。我们预期这将需要一些前端的设计来配合后端，从而适用于不同的应用场景。

8 总结

在学习了 Lamport 对于 paxos 的最初设计之后，我们按照 paxos 算法最原始的设计理念，将 paxos 共识算法应用到数据库的设计之中，以树洞的数据结构为载体，构建基于多轮 paxos 进程池的后端数据库。

从 paxos 算法上，我们提出了构建多轮 paxos 进程池，来处理不同轮次消息共识，保证指令的有序执行。并在性能表现上上，我们的系统也表现了完全可用的读写性能。

在未来，我们希望对数据库的性能进行改进，包括但不限于各个节点的一致性、数据库安全性、算法应用的适应性等。

References

[1] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. $\{$ZooKeeper$\}$: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[4] S. King and S. Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake.

[5] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998. ISSN 0734-2071. doi: 10.1145/279227.279229.

[6] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001. URL `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[7] D. Mazieres. Paxos made practical. 08 2009.

[8] A. Medeiros. ZooKeeper' s atomic broadcast protocol: Theory and practice. *Aalto University School of Science*, 20, 2012.

[9] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[10] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319.

[11] J. Zheng, Q. Lin, J. Xu, C. Wei, C. Zeng, P. Yang, and Y. Zhang. PaxosStore: high-availability storage made practical in WeChat. *Proceedings of the VLDB Endowment*, 10(12):1730–1741, 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137778.