**区块链时代的拜占庭容错：Tendermint（一）**

分布式共识系统成为现代互联网基础设施中的一个关键组件，正助力于每一个主要的互联网应用。本章节内容介绍了必要的背景材料来理解和探讨这些系统。

**复制状态机（Replicated State Machine）**

最常见的用来研究和实施分布式共识（distributed consensus）的范例的是复制状态机的范例，其中，一个确定的状态机在数个进程（processes）之间被复制，这样不管部分进程是否失败，这些进程看上去像单个状态机。状态机有一些列输入驱动，这些输入被称作交易（transactions），每一个交易根据其是否有效，可能引起一个状态迁移并返回一个结果。更正式的，交易为数据库上的原子操作（atomic operation），意味着其要么完成要么根本就没有发生，不能返回一个中间状态（ intermediate state）。状态交易逻辑（state transition logic）由状态机的状态转移函数决定，这个函数映射了一个交易和目前的状态到一个新的状态和一个返回值。状态转移函数有时也被称为应用逻辑（application logic）。

订购交易（order the transactions）并且将相应的交易日志（ transaction log ）复制到每一个进程是共识协议的责任。使用一个确定的（deterministic）状态转移函数意味着在给定同样的交易日志的情况下，每一个进程将计算出相同的结果。

复制状态机架构如图2.1所示。

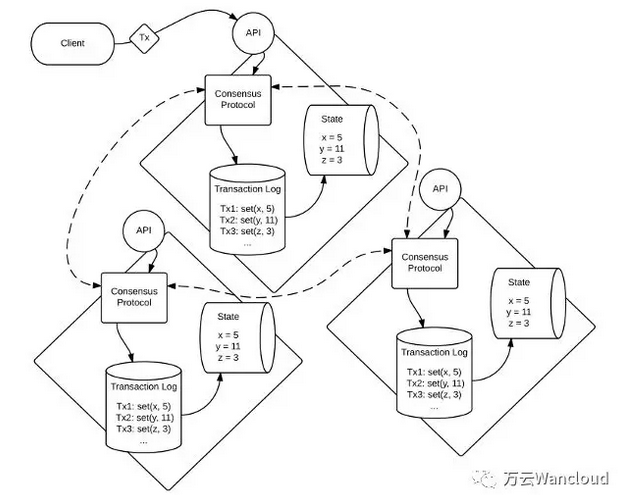


图2.1

图2.1 一个复制状态机在多个机器之间复制了一个交易日志和结果状态。交易从客户端接受，运行了共识协议，在交易日志中订购（ordered），最后执行得到最新状态。在图中，每一个菱形代表了单个机器，其中，虚线代表机器间的通讯用来承载进行订购交易（ ordering transactions）的共识协议。

Tendermint的目标是创建一个通用目的，高性能，安全和健壮的复制状态机。

**不同时性（Asynchrony）**

容错复制状态机（fault-tolerant replicated state machine）的目的是在对上层提供服务的时候，协调网络中的计算机的同步，不管是否存在故障节点。

保持同步意味着成功复制交易日志；提供一个有用的服务意味着在处理新交易的时候保持状态机的可用性。传统上系统的这些方面被各自成为安全性（safety）和可用性（liveness）。通俗地，安全性意味着没有任何坏的事情发生；可用性意味着好的事情最终发生。安全违规（ violation of safety）意味着存在两个或者更多的有效的，竞争的交易日志。可用性违规（violating liveness ）意味着一个无法响应的网络。

通过接受所有的交易可以很容易来满足可用性。通过不接受任何交易可以很容易来满足安全性。因此，状态机复制算法可以被看作在两者之间的一个平衡。一般地，进程在提交一个新的交易在之前，需要对来自于其他进程的信息设立一个阈值。在同步的环境中，我们对网络消息的最大延迟或者处理器时钟的最大速度作出假设，通过轮流坐庄来提议新的交易，进行大多数投票表决，如果提议者在同步假设的区间内并没有产生任何提议，则跳过（skip）提议者的这一回合。

在一个异步的环境中，没有关于网络延迟或者处理器速度的保证的假设，权衡将变得更为困难。事实上，所谓的FLP不可能性结果（FLP impossibility result）证明了在确定的异步进程（单个进程可能会崩溃）之间的分布式共识的不可能性。该证明意味着，因为进程可能失败，存在协议的有效执行，但进程恰好在某一时间崩溃这样就阻止了共识。因此，我们对共识没有任何保证。

一般地，协议中的同步是通过管理某些交易时用到的超时（timeouts）来进行的。在异步环境中，消息能够被任意延迟，依赖同步来确保安全性的话可能导致交易日志的分叉。依赖同步来保证系统的可用性能够引起共识的宕机，并且服务无法响应。前者通常被看作更为严重，因为调解冲突日志可能是一个令人生畏或者不可能的任务。

实际上，仅当消息延迟能够被良好的定义和控制的时候，同步解决方案才会被使用，例如在一架飞机上的控制器之间，或者利用同步的原子时钟的数据中心之间。因此，尽管存在很多高效的同步解决方案，计算机网络的一般化的不可靠性（general unreliability）太大以至于不能实际投入使用而不显著增加额外的成本。

根本上有两种途径来克服FLP不可能性结果。第一个是采用更强的同步假设-甚至相当弱的假设也是足够的，例如，那个唯一的最终崩溃的进程被怀疑崩溃了，正确的进程不受影响。一般地，这个方法利用leaders，其扮演了一个特别的协作的角色，并且在超时并被认为发生故障了以后可以被跳过。实际上，这样的领导选取机制成功运转起来很难。

第二种克服FLP的方法是使用非确定性的-包含随机化元素，这样达成共识的可能性接近为1。尽管第二种方法更智能并且某些高级加密技术近些年来取得了速度上的提高，依赖随机化的方法通常更慢。

**广播和共识**

为了让一个进程复制状态到其它进程上，它必须有基本通讯原语的权限来允许其传播或者传递信息。一个最有用的原语是可靠广播（reliable broadcast）。可靠广播（RBC）是一个广播原语满足如下特性，对消息m，有：

有效性（validity） - 如果一个正确的进程广播m，它最终成功传达了m

一致性（agreement） - 如果一个正确的进程成功传达了m，所有最终所有的进程成功传达m

完整性（integrity） - m只传递一次，并且是以广播的形式被发送者发送出去

本质上，可靠广播使得消息最终到达所有的进程一次。

另外，更有用的原语是原子广播（ atomic broadcast（ABC）），其满足可靠广播（RBC）和另外的一个属性：

总的顺序（total order） - 如果正确的进程p和q分别传递出m和m'，p传达m在m'之前，那么q传达m在m'之前

原子广播是一个可靠的广播，其中值(values)以相同的顺序被发送到每个机器上。注意到这实际上复制交易日志的问题。通俗地讲，该问题可以被称作共识，共识原语的标准定义满足以下条件：

终止性 - 每个正确的进程最终能做出决定

完整性 - 每个正确的进程最多只做出决定一次

一致性 - 如果一个进程做出了v1的决定， 并且另外一个进程做出了v2的决定，那么v1=v2

有效性 - 如果一个正确的进程做出了v的决定，至少一个进程提议了v

直观地，共识和原子广播看上去十分类似，主要的差异在于，原子广播本身作为一个协议是连续的，然而共识期望终止。这就是说，每一个可以精简为另一个。共识可以被精简为原子广播通过决定第一个原子广播的值。原子广播可以精简为共识，通过依次运行许多共识协议的实例，然而存在一些微妙的考量，特别是在处理拜占庭故障方面。一个完整的参数空间的关于原子广播精简为共识的描述仍然是一个开放的研究话题。

历史上，尽管大多数用例实际上需要原子广播，采用的最为广泛的算法是称作Paxos的共识算法，在90年代介绍并且证明该算法正确性的是Leslie Lamport。Paxos同时赋予和困惑了共识科学，一方面提供了第一个真实世界的，实用的，容错的共识算法，另一方面又难于理解和解释。算法的具体实现使用了其专门的技术来从Paxos建立原子广播，使得这个生态难于操纵，理解和利用。不幸的是，几乎没有任何工作使得提高该框架更容易理解，尽管有尝试来描绘解决方案中的各种困难。

在2013年，Ongaro和 Ousterhout发表了Raft，一个状态机复制算法，其主要的设计动机是可理解性。与其从一个共识算法开始，并且尝试建立原子广播，Raft的设计首先考虑的是交易日志，寻求正交组件，其组合在一起来提供最终的原子广播，尽管其不是被这样描述的。

Paxos是工业领域的主要共识算法，在工业领域像亚马逊，谷歌和其他扩建了高可用性全球互联网服务的公司。Paxos 共识位于应用程序栈的底部，提供了资源管理和分配的一致接口，操作在一个更慢的时标相比于其他面对用户的高可用性应用程序。

Raft登场以来得到了广泛的采用，特别是在开源社区，其具有多个主流语言的实现，并且在多数项目中作为其主干。

Raft与Paxos在设计方面主要的不同是先聚焦于交易日志，而不是单个值，特别是允许一个leader持续提交交易直到其卸任，这时领导选举开始生效。在某种程度上，这类似于在区块链中采用的方法，尽管其主要优势是能够容忍不同种类故障。

**拜占庭容错（Byzantine Fault Tolerance）**

区块链通过在共享数据库上责任的去中心化，减少了对手方风险，因此被称为“信任机器”。比特币由其具有的抵抗任何攻击和恶意行为的能力而著称。传统地，容忍恶意行为的共识协议被称为拜占庭容错共识协议（Byzantine Fault Tolerant ）。术语“拜占庭”被使用，源于拜占庭将军们面对的类似问题，这些将军们尝试相互协调来攻击罗马，使用唯一的信使，其中一个将军可能是叛徒。

在一个崩溃故障中，一个进程可能宕机。在一个拜占庭故障中，故障节点能做任何事情。崩溃的故障更容易处理，因为没有进程会对其他进程说谎。只存在崩溃故障的系统可以通过简单的多数决定规则（ majority rule）来操作，因此通常能够同时容忍近一半的系统故障。如果系统能够容忍失败的数量是f，这样系统必须至少有2f+1个进程。

拜占庭故障复杂一些。在一个具有2f+1进程的系统中，如果f是拜占庭，这些拜占庭节点可以协作来说任何事情对另外f+1的进程。例如，我们尝试取得单比特共识，并且f=1,所以我们有N=3个进程，A，B，C，其中C是拜占庭，如图2.2所示。C可以告诉A这个值是0，告诉B这个值是1。如果A同意它是0，B同意它是1，那么他们都将认为他们获得了大多数，提交，这样就违反了安全条件。因此，拜占庭系统能够容忍的故障上限小于非拜占庭系统。

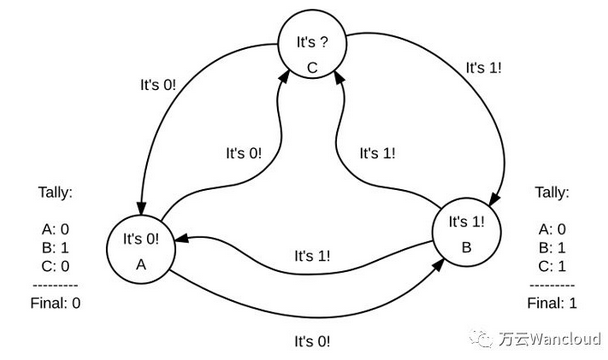


图2.2

图2.2 一个拜占庭进程C，告诉A一件事，告诉B另外一件，致使他们得出关于网络的不同的结论。这里。简单的大多数投票导致了安全违规，源于单个拜占庭进程。

事实上，可以证明拜占庭故障的上限为f<N/3。因此，为了单个拜占庭进程，我们至少需要N=4个节点。那么故障进程就不能像当N=3时那样分裂投票。

在1999年，Castro 和 Liskov 发表了实用拜占庭容错（Practical Byzantine Fault Tolerance），提供了第一个优化的适用于实际的拜占庭容错算法。其为工业系统中拜占庭容错的实用性设定了一个新的先例，每秒可以处理成千上万比交易。除此之外，拜占庭容错仍然是被人认为是昂贵的，大部分时间是不必要的，并且最流行的实现很难建立在其上面。因此，尽管学术对其兴趣日增，包括大量提高了的变种，但在实施和配置方面并没有太多进展。进一步，如果网络中超过1/3的节点是拜占庭，PBFT将不能提供任何保证。

**密码学，信任和经济学**

根本上说，容错这个问题起源于缺乏信任 - 不知道一些进程将如何表现（behave）。正式地，信任需要从理论上被定义成一种信息，其作为一种减少世界模型熵的手段-信任某个人就是乐观地减少这个人对于这个世界的不确定性，使得可以把注意力放在更高阶的组织层面上。

密码学原语从根本上与信任问题相关，主要被定义为允许熵大量减少的机制-成功认证一个密码学函数把一个可能结果的分布坍缩成一个点，或者在一些例子中一些少量的结果。

它是做所周知的有着更好制度信任的文明，例如法治具有更高的生产率和更充满生气的经济。结果产生了一个直观的感觉，能够更多的信任相互作用，减少可能结果的空间，其需要被主动建模，使其更容易协调。不幸的是，评价现代机构的信誉越来越困难，因为他们的复杂度在近些年增加了很多，增加了可能性，其声称的确定性是幻觉。

幸运的是，密码学形成了社会中心的信任体系的基础，奇迹大地提高了人类在全球范围内协作的能力，由于减少了欺骗和无责任行动的风险。比较有趣的是密码学原语在BFT算法中的重要性，为了认证和散播不确定性。

最有趣的是，经济机制也能当作减少熵的一种方式，迄今为止经济代理可以被激励-更有可能被用来执行一个特定的行为。比特币深入的洞察是密码原语可以与经济激励结合起来有效减少公共共识网络的熵来取得状态安全复制。

更正式的信任的信息理论根基，密码学，共识和经济学和他们之间的关系的调查将会在以后的工作中展开。

**区块链**

区块链的核心是一个关于拜占庭容错原子广播的聚焦完整性的方法。例如，比特币区块链结合了经济学和密码学随机化来提供一个强的概率保证，保证安全违规不会发生，给定了一个弱同步假设，即区块间的通讯比产生哈希碰撞更迅速。实际上，然而，众所周知，比特币的安全保证容易受到各种狡猾（subtle）的攻击。

区块链从两个关键的优化中得到它的名字，其利用这两个优化解决了原子化广播的问题。第一个是交易以块为单位进行分组来分摊高提交延迟（在10min量级）。第二个优化是通过加密哈希把区块链接起来成为一个不可篡改的链，这样就很容易验证历史记录。两个优化都相对于原始的BFT-ABC的有所提高，前者提高了性能，后者提高了容错。

经过这几年的发展，使用哈希链接交易块并以原子广播发送出去已经成为公共的区块链共识算法。据作者所知，Tendermint是第一个这样的提议，升级了知名的80年代的BFT算法，成为了自成体系的共识算法。Tendermint被IBM跟进，IBM升级PBFT到区块链，JP摩根升级了一个Raft的BFT版本。

**过程演算**

各个部分同时执行的分布式系统，因难以设计、构建和调试而饱受诟病。它们更难以正式验证，因为大多数技术的形式验证，以及实际上非常基础的计算机科学，都是通过推算得到的，因此很难正式验证。

过程演算是一种为并发计算提供了有条理的基础原理的模型系列。最通常的演算方法，Communicating Sequential Processes（CSP）构成了许多现代编程语言的理论基础。比如Go语言，在设计中包含了并发原语。

我们可以使用一个正式的逻辑来表达一个过程可能满足的属性。举例来说，模态Hennessy–Milner逻辑可以表示，在某些或所有形式的动作发生后，一个进程将满足其他一些逻辑表达式。通过将更复杂的运算方法添加到逻辑中，可以建立正式的系统，可以很容易地描述分布式系统的重要属性，比如安全性、可用性和本地化等。通过π-calculus编写的系统可以被正式验证，以满足使用模型检查软件的相关属性。

当我们使用π-calculus来详细说明Tendermint算法时，我们会使用相关的正式逻辑，以及相应的属性验证，以备将来的工作。

**Tendermint的需求**

比特币及其衍生物的成功，特别是以太坊和他们的关于安全，自治，分布式，对任意代码的容错执行的前景引起了事实上每一个主要的金融机构的兴趣。特别地，出现了对两种种区块链技术：一方面是公链，被亲切地称为Big Dad公链（Big Bad Public Blockchains），其协议被内建经济激励通过原生货币（native currency）的方式所支配。另一方面是所谓的私有链，更准确的被称为“联盟链”（ consortia blockchains），通过哈希树的使用，数字签名，p2p网络和加强的问责制，其对传统共识和拜占庭算法有一定的提高。

就像现代社会的基础设施持续的去中心化或者正如商业的跨组织本质，对透明，问责和高性能的拜占庭系统的需求越来越多，这个系统支持的应用程序从财政到域名注册到电子投票和与治理的高级机制协作和未来的演进。Tendermint这个解决方案对联盟或者跨组织逻辑进行了优化，但是足够灵活来容纳任何人，从私有企业到全球货币，并且性能足够高来与主要的，非拜占庭容错的，共识解决方案竞争例如 etcd, consul, and zookeeper，于此同时提供了更强的恢复性，安全保证，对应用开发者的灵活性。

## 区块链时代的拜占庭容错：Tendermint(二)

本章阐述Tendermint共识算法和用于原子广播（ atomic broadcast）的相关区块链。拜占庭容错共识问题将被详细讨论，并且Tendermint共识的一个正式说明将以π-calculus的形式给出。Tendermint区块链已经被非正式地证明为满足原子广播。将来我们将以进程演进的方式来描述完整的区块链协议，并证明相关特性。

### ****Tendermint综述****

Tendermint是区块链范式中的一个安全的状态机复制算法。其算法形态为BFT-ABC，并且附加责任制，便于验证拜占庭节点的不诚实行为。

Tendermint算法给每个区块赋予一个增量索引或者高度（height），在某一高度中只存在一个有效的区块，区块链从高度为0的创世纪块开始，由一个验证者集合投票产生下一个区块，其中每一个验证者由各自的公钥标识。每一个验证者需要维护一份完整的复制状态的拷贝。在投票产生某一高度的区块的过程中，在正式提交（commit）某一高度的区块之前，至少需要经过一轮（round）投票（vote）来达成共识。每一轮都会通过round robin的方法产生一个提议者（proposer），该提议者在当轮以广播的形式提出一个提议（proposal），提议经过验证者的集体投票，来决定是否最终提交该区块或者进入下一轮。在提议的区块真正被提交（commit）之前，验证者们需要进行两轮投票（pre-vote & pre-commit）, 通过一个简单的锁机制用来阻止少于总数1/3的拜占庭节点攻击。由于Tendermint网络的不同时性（asynchrony），当拜占庭节点超过总数的1/3，网络存在瘫痪的可能性。

注意到，tendermint的多轮投票机制的核心是共识算法。每一个区块包含一些元数据（metadata），称作区块头（header）。区块头里包含本区块的高度，提议时间，本区块所有交易的梅克尔根哈希值。

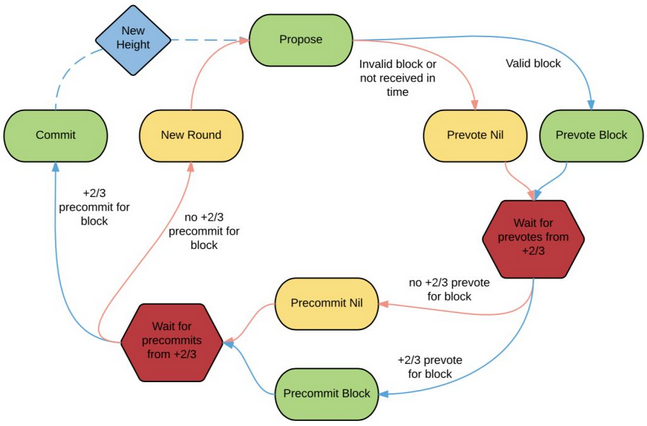
### ****共识****

共识算法可以大致分为以下几部分：

•  **提议（Proposals）**：在每一轮（round）中，新区块的提议者必须是有效的，并且告诉（gossiped）其他验证者。如果在一定时间内没有收到当轮提议（proposal），当前提议者将被后面的提议者接替。

•  **投票（Votes）**：两阶段的投票基于优化的拜占庭容错。它们分别被称作预投票（pre-vote）和预提交（pre-commit）。对于同一个区块同一轮如果存在超过2/3的预提交（pre-commit）则对应产生一个提交(commit)。   
•  **锁(Locks)**：在拜占庭节点数少于节点总数的1/3的情况下，Tendermint中的锁机制可以确保没有两个验证者在同一高度提交（commit）了两个不同的区块。锁机制确保了在当前高度验证者的下一轮预投票或者预提交依赖于这一轮的预投票或者预提交。

为了应对单个拜占庭故障节点，Tendermint网络至少需要包括4个验证者。每个验证者拥有一对非对称密钥，其中私钥用来进行数字签名，公钥用来标识自己的身份ID。验证者们从公共的初始状态开始，初始状态包含了一份验证者列表。所有的提议和投票都需要各自的私钥签名，便于其他验证者进行公钥验证。



验证人在发起提议（proposal）步骤之后，当且仅当收到其它验证人超过三分之二（+2/3）的投票后才会进一步推进流程。虚线箭头表示进入下一个区块高度共识流程的原子广播。

共识开始于第0轮，第一个提议者（proposer）是区块链头里验证者列表里的第一个验证者。每一轮最终要么完成了一个提交（commit），要么直接进入当前高度的下一轮，每一轮都会产生一个新的提议者。

与其他选举（leader election ）算法不同，Tendermint每一轮都会产生一个新的提议者(proposer)，验证者投票决定是否进入下一轮，这与接受提议的流程类似。

每轮的开始对同步有弱的依赖性。每一轮开始期间，存在一个用来计时的本地同步时钟，如果验证者在TimeoutPropose时间内没有收到提议，验证者将参与投票来决定是否跳过当前提交者。TimeoutPropose会随着轮数的增加而增加。

每轮收到提议以后，进入完全异步模式。之后验证者的每一个网络决定需要得到2/3验证者以上的同意。这样降低了对同步时钟的依赖或者网络的延迟。但是这也意味着如果得不到1/3以上验证者的响应，整个网络将瘫痪。

简言之，每轮，开始提议弱同步，之后投票完全异步。

为了增强Tendermint共识网络的安全性，引入了少量的锁定规则（locking rules）来迫使验证者自证其投票的合法性。尽管我们不需要实时广播他们的合法证明，但是我们确实期望验证者们保存相关数据。这样当网络被拜占庭故障节点瘫痪时，其可以存留为相关证据。这个问责机制确保在网络故障（例如PBFT）的时候Tendermint具有一个更健壮的担保（guarantees）。

验证者使用一组不同的消息（messages）来管理区块链，应用程序状态，p2p网络和共识。其中，核心的共识算法包含两类消息：

**ProposalMsg:** 对应某一高度及某一轮数的区块的提议（proposal），该提议已经由提议者签名

**VoteMsg:** 对某一提议的签名投票

### ****一、提议****

每轮开始于一个提议（proposal），提议者从内存池（Mempool）选取一批交易进而构成了一个区块，该区块随后被嵌套在ProposalMsg中，最后提议者广播（broadcast）ProposalMsg。如果这个提议者是拜占庭节点，他可能向不同的验证者广播不同的ProposalMsg。

提议者通过一个简单并且相对固定的的roubd robin轮流坐庄，所以每一轮只有一个有效且被所有验证者公认的提议者。如果验证者收到了之前更低轮次的提议或者提议来自于非法的提议者，该提议将被拒绝。

提议者的轮流坐庄对于拜占庭容错是必要的。比如，对于raft算法，如果选举出来的leader是拜占庭，并且leader与其他节点网络连接状态良好，该leader可以完全控制整个网络，网络节点的安全和正常运转将无从得到保障。Tendermint通过投票和锁的机制（voting and locking mechanisms ）确保了系统的安全性。如果一个提议者在限定时间内没有处理任何交易，排在其后的提议者将会接替他。更有趣的是验证者能通过治理模块投票来移出或者替换拜占庭验证者。

### ****二、投票****

一旦验证者从网络中收到了一份完整的提议（proposal ），他对该提议进行预投票（pre-vote）签名，并且广播到网络中。如果验证者在ProposalTimeout时间内没有接收到一个有效的提议，其对该提议的预投票为空（nil）。

在存在拜占庭节点的异步环境中，单阶投票，即每个验证者对每个提议只投一次，不能足以确保整个系统的安全。本质上，因为验证者可能做出一些不诚实的行为，并且消息的到达时间没有任何保障，一个不诚实的验证者可以与其他验证者进行协作来提交（commit）一个区块，然而其他没有看到这个提交区块的验证者进入了新的一轮，并提交（commit）了一个不同的区块。

一个单阶的投票允许验证者互相沟通他们知道的关于该提议的信息。但是为了容忍拜占庭故障，他们也需要互相告诉对方他们自己了解到的其他验证者声称了解到的关于该提交的信息。换句话说，二阶段提交确保了足够的验证者见证了第一阶段的结果。

对于某个区块的非空预投票是为网络提交（commit）区块已做好准备的投票。空预投票是为网络直接进入下一轮的投票。在理想的一轮中，超过2/3的验证者为该提议进行了预投票。在任意一轮中，区块具有的超过2/3的预投票被称作一个波尔卡（polka）。超过2/3的空预投票成为空波尔卡（nil-polka）。

当一个验证者收到了一个波尔卡（polka），他接受到了一个信号，即网络准备提交该区块，作为一个验证者签名并且广播预提交（pre-commit）的背书。有时，由于网络的不同时性，验证者可能没有收到对应的波尔卡或者波尔卡根本就不存在。在这种情况下，验证者没有对应的波尔卡为这个预提交背书，此时预提交为空。也就是说，在没有收到波尔卡背书的情况下，签名一个预提交被看作是一个恶意行为。

预提交（pre-commit）是关于提交(commit)一个块的投票。空预提交则投票进入到下一轮。如果验证者收到2/3以上验证者的预提交，则其在本地提交该块，计算结果状态，并移动到下一高度的第0轮。如果验证者接收到超过2/3的空预提交，则投票进入下一轮。

### ****三、锁****

多轮投票的安全问题是棘手的，必须避免同一高度不同轮数分别提交两个不同区块的情形。在Tendermint中，这个问题可以通过锁机制（locking mechanism）得到解决。锁机制的大致定位在波尔卡附近。本质上，预提交必须有一个波尔卡为其背书，验证者被锁定在其最近预提交（pre-commit）的区块上。

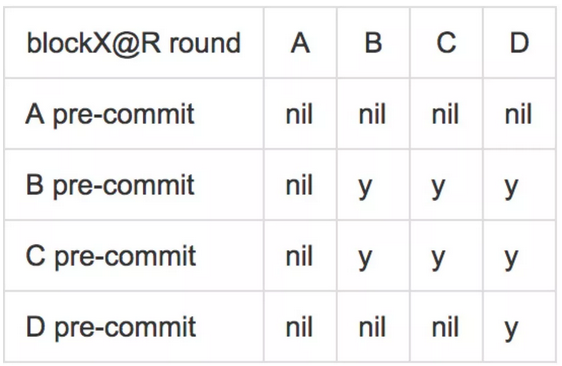
锁定规则：

**· 预投票锁**（Prevote-the-Lock）：验证者只能预投票(pre-vote)他们被锁定的区块。这样就阻止验证者在上一轮中预提交(pre-commit)一个区块，之后又预投票了下一轮的另一个区块。

**· 波尔卡解锁**（Unlock-on-Polka ）：验证者只有在看到更高一轮（相对于其当前被锁定区块的轮数）的波尔卡之后才能释放该锁。这样就允许验证者解锁，如果他们预提交了某个区块，但是这个区块网络的剩余节点不想提交，这样就保护了整个网络的运转，并且这样做并没有损害网络安全性。

简单来说，验证者可以被看作锁在任意高度-1轮的nil-block上，所以波尔卡解锁意味着验证者不能预提交一个新高度的区块直到他们看见一个波尔卡。

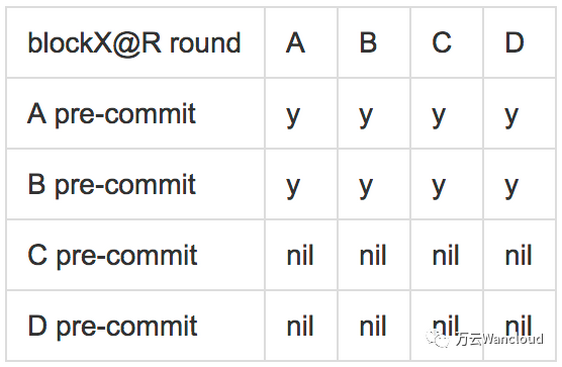
这些规则可以以例子的形式被更直观的理解。考虑4个验证者，A,B,C,D,假设有一个第R轮关于blockX的提议。现在假设blockX已经有一个波尔卡，但是A看不见它，预提交（pre-commit）为空，然而其他人对blockX进行了预提交。进一步假设只有D看见了所有的预提交，然而其他人并没有看见D的预提交（他们只看见他们的预提交和A的空预提交）。D现在将要提交（commit）这个区块，然而其他人进入到R+1轮。由于任何验证者都可能是新的提议者，如果他们提议并投票了一个新的区块blockY，他们可能提交这个区块。可是D已经提交了bockX，因此损害了系统的安全性。注意，这里并没有任何拜占庭行为，仅仅是不同时性。

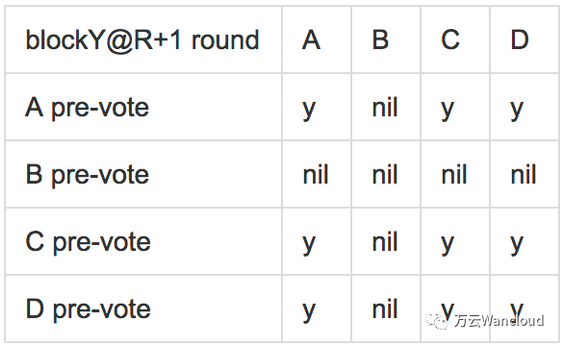


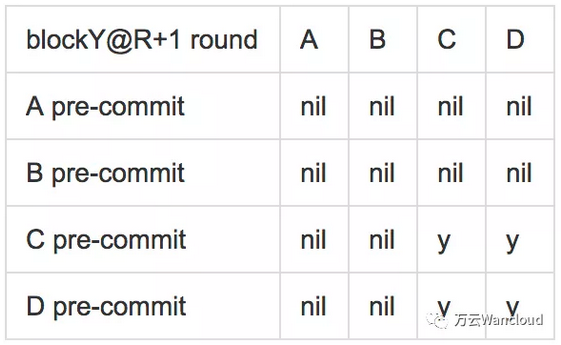
（为了便于读者理解，译者补充此表格，下同）

锁定解决了这个问题通过强迫验证者粘附在他们预提交（pre-commit）的区块上，因为其他的验证者可能居于这个预提交进行了提交（如上例中的D）。本质上，在任何一个节点一旦存在超过2/3预提交（pre-commit），整个网络被锁定在这个区块上，也就是说在下一轮中无法产生一个不同块的波尔卡。这是预投票锁的直接动机。

当然这里必须有相应的解锁方式。假设在某一轮中，A和B预提交（pre-commit）了blockX，与此同时C和D的预提交为空。因此所有的验证者进入到下一轮，预提议(pre-vote）blockY。假设A是拜占庭，为blockY也进行了预投票（不考虑其被锁在blockX上），导致了一个波尔卡。假设B并没有看见这个波尔卡，预提交为空，此时A下线，C，D预提交bolckY。他们进入到下一轮，但是B仍然被锁定在blockX上，C和D被锁定在blockY上。这时因为A下线了，他们将永远得不到一个波尔卡。因此即使在拜占庭节点少于1/3的情况下，这里网络的正常运转仍然受到了影响。







解锁的条件是1个波尔卡。一旦B看见了blockY的波尔卡（用来为C和D的关于blockY的预提交背书），他应当能够解锁并预提交（pre-commit）blockY。这是波尔卡解锁的动机，其允许验证者在看见更高轮数波尔卡的时候解锁并且提交对应的新区块。

### ****区块链****

Tendermint对交易按批或块进行处理。区块之间通过加密哈哈希算法链成一个完整的区块链。区块链包括经过排序的交易日志和验证者提交的相关证据。

### ****一、为什么是区块？****

共识算法一次提交若干个交易（transactions）。正如在第二章提到的那样。从分批原子广播（batched atomic broadcast）的角度来看待这个问题，对应两个主要的优化，其给了我们更多的吞吐量和容错能力：

**·  带宽优化**：因为每一次提交(commit)需要验证者之间的两轮通讯，以块为单位交易的批处理，平摊了提交的成本在该区块中的所有交易上。

**·  完整性优化**：区块的哈希链形成了一个不可篡改的数据结构，跟git仓库很像，具备历史任意点的子状态认证检查的能力。

区块也引起了另外一个效应，看上去更微妙，但是可能更重要。他们增加了单个交易的最小延迟到区块的最小延迟，对于Tendermint来说在数百毫秒到数秒量级。传统的序列化数据库系统提供了提交延迟在毫秒到数百毫秒量级。他们的低延迟是因为这些数据库不是拜占庭容错的，只需要一轮通讯而不是两轮和来自于1/2而不是2/3节点的响应。然而，与其他具有快速提交时间（commit times）的选举算法不同，Tendermint提供了一个更常规的脉冲（pulse ），在节点故障和网络不同时方面对整个网络的状态具有更好的响应度。

脉冲在通讯自治系统一致性方面的角色现在并不明朗，但是由此引发的延迟在金融市场中是具有前景的。

### ****二、区块的结构****

区块的目的是打包一批交易，并且链接到前面一个块。链接包含两种形式：前面一个区块的哈希和前面区块的预提交的集合，其也被称作LastCommit。因此一个区块由三部分构成：区块头，交易列表和Lastcommit。

### ****安全性****

这里我们简要的证明一下Tendermint满足原子广播。原子广播被定义为满足以下条件：

**· 有效性（validity）** - 如果一个正确的进程广播m，它最终成功传达了m

**· 一致性（agreement）** - 如果一个正确的进程成功传达了m，所有最终所有的进程成功传达m

**· 完整性（integrity）** - m只传递一次，并且是以广播的形式被发送者发送出去

**· 总的顺序（total order）** - 如果正确的进程p和q分别传递出m和m'，p传达m在m'之前，那么q传达m在m'之前

注意到， 如果把m看作一个区块，Tendermint并不满足有效性，因为并不能保证提议的区块最会会被提交，因为验证者可能进入到新的一轮，并提交一个不同的区块。

如果我们把m看作某一区块里的一批交易，那么我们能够满足有效性通过验证者重新提议同一批交易直至交易最终被提交。

为了满足完整性的第一部分，我们必须引入额外的规则来禁止一个合法的验证者提议或者预提交一个区块，其中这个区块包含的这批交易已经被提交过。幸运的是，交易可以被梅克尔根索引，在提议和预提交以前可以进行相关的查找来滤除已经提交的交易。

或者我们可以把m当成一个交易（transaction），通过引入内存池的持久属性，可以满足有效性，即，交易可以驻留在内存池中直到它被提交。然而为了满足完整性的第一部分，我们必须依赖应用程序状态（application state）来制定一些针对交易的规则，这样一个给定的交易只能进行一次。例如，可以通过基于账户的序列号，正如在以太坊中的那样。或者保存一份未使用资源的列表，每一个资源只能被使用一次，正如在比特币中使用的那样。因为有多种方法，Tendermint本身并不保证消息只传达一次，但是允许应用开发者来指定相关特性。完整性的第二部分显而易见，因为只有正确的提议者提议的区块中的交易才能被提交。

为了证明Tendermint满足“总的顺序”，我们引入了一个新的特性，状态机安全性（state machine safety），并且可以证明满足状态机安全性的协议必定满足“一致性”和“总的顺序”。所谓的状态机安全是指如果一个正确的验证者在高度H提交了一个区块，没有其他的验证者在同一高度提交一个不同的区块。考虑到所有的消息最终被接收，这个立刻暗示了一致性，因为如果一个正确的验证者在高度H提交了一个区块B，包含了交易m，所有其他的正确的验证者不能提交其他的区块，因此最终提交了区块B，传达了消息m。

现在，我们需要证明状态机安全满足“总的顺序”，并且Tendermint满足状态机安全。为了证明前者，考虑两个消息m和m'分别由验证者p和q发出。状态机安全确保p发出消息m在高度Hm当且仅当q发出消息m在高度Hm，并且p发出消息m'在高度Hm'当且仅当q发出消息m'在高度Hm'。不失一般性，因为高度是严格递增的，假设Hm<Hm'。那么我们有p发出消息m在m'之前当且仅当q发出消息m在m'之前，这恰恰就是“总的顺序"的定义。

最后，为了证明当拜占庭节点少于1/3的时候，Tendermint满足状态机安全，我们采用反证法。假设Tendermint并不满足状态机安全，允许在某一高度提交多个区块。那么我们可以证明至少需要1/3的拜占庭节点，与假设矛盾。

考虑一个有效的验证者在高度H和轮数R提交了一个区块B。提交一个区块意味着验证者在第R轮收到了关于区块B的超过2/3的预提交。假设另一个区块C在高度H提交。我们有两个选项：要么在第R轮提交要么在S轮提交（S>R）。

如果区块C在第R轮提交，那么超过2/3的验证者必须为该区块预提交，那么意味着至少1/3的验证者在第R轮同时对区块B和C进行了预提交，那么显然这些同时节点是拜占庭节点。假设区块C在S轮提交。因为超过2/3对B区块进行了预提交，他们在S轮也将被锁定在区块B上，因此他们必须对B进行预投票。为了对区块C进行预提交，他们必须接收到关于区块C的波尔卡，因此需要关于区块C的超过2/3的预投票。然而，超过2/3的验证者已经被锁定在区块B上。节点为了收到区块C的波尔卡至少需要网络中1/3的验证者违背锁机制，这部分节点显然是拜占庭节点。因此，为了违背状态机安全，至少需要1/3的拜占庭验证者。即若网络中的拜占庭节点少于总数的1/3，Tendermint满足状态机安全性。

综上，Tendermint满足原子广播。

在未来的工作中，我们会提供关于Tendermint的安全性的更正式的证明。

### ****责任制****

一个具有问责制的拜占庭容错算法能够在存在安全隐患时标识所有的拜占庭验证者。传统的拜占庭容错算法并没与这个特性，对应地也没有任何相应的保证。当然，问责制仅能适用在拜占庭节点在1/3到2/3的情况。如果超过2/3的节点是拜占庭，他们能够完全占据协议，此时无法保证一个合法的验证者可以收到任何拜占庭节点违法的证据。

进一步，问责制是在异步网络环境下最终性的尽力而为，在这样的网络环境中着安全问题，关键消息（critical messages）的延迟使得在探测到安全问题以后才可能发现拜占庭验证者。事实上，如果正确的进程（correct processes）可以接受拜占庭行为的相关证据（evidence），但是在他们能够通讯之前不可逆地失败了（fail irreversibly），可能使得问责制永久失效（ Permanently compromised），尽管实际上这种情形可以通过高级备份策略来克服。

通过枚举安全问题的各种隐患，拜占庭验证者是可以识别的，这样协议是具有问责制的。与其它竞选相关的协议相比，Tendermint的简洁给予了其更简单的分析方法。

在Tendermint存在两类安全隐患，每一种都是可问责的。第一种，拜占庭提议者在单轮中产生两个冲突的提议，并且拜占庭验证者同时对这两个提议进行投票（vote）。第二种，一些验证者在单轮已经提交（commit）之后，拜占庭验证者违反锁机制（locking rules），致使其他验证者在随后的轮数提交一个不同的区块。注意到，若拜占庭验证者少于2/3，只通过违反解锁机制的方法是无法引发安全性问题的，同时超过1/3的节点必须违背波尔卡锁机制，因为每一个提交（commot）需要有一个波尔卡为其背书。

在存在提议或者投票冲突的情况下，同时接受冲突的提议或者投票，可以根据这些提议或投票的签名来辨别这些拜占庭节点。

在违反锁定机制(locking rules)的情况下，伴随着相应的安全性问题，有效的验证者必须广播在当前高度看到的所有投票，这样证据可以被收集起来。少于2/3的正确验证者在所有导致两个区块被同时提交的投票中集体隐匿。此时在这些投票中，如果没有1/3或者更多的验证者签名冲突的投票，那么存在1/3或者更多的验证者违反了锁定机制。

如果预投票（ pre-vote ）或者预提交（ pre-commit）影响了一个提交，它一定会被一个合法的验证者看见。因此，通过搜集所有的投票，通过匹配每一个预投票和最近的预提交，可以探测到违反锁机制的行为（violations of Prevotethe-Lock ）。

类似的，通过匹配预提交（pre-commit ）和为其背书的波卡尔卡（polka），可以探测到违反解锁机制的行为（violations of Unlock-on-Polka ）。注意到这就意味着如果拜占庭验证者可以在看见波尔卡之前预提交（pre-commit），并且如果相应的波尔卡最终发生的话，拜占庭验证者将逃脱责任制。然而，如果每一个预提交有波尔卡背书的话，这些安全隐患就不存在。

目前的设计提供了问责制，伴随着后危机广播协议（post-crisis broadcast protocol），但是其能够用来提高实时的问责制。也就是说，一旦提交被改变，相应的预提交，为预提交背书的预投都会发生改变，这样一直回退到创世纪块。通过上面的方式，如果发生安全问题，没有背书的投票可以立即被探测到。

### ****故障和可用性****

作为一个拜占庭共识容错算法，Tendermint可以容忍拜占庭故障节点到（但不包括）节点总数的1/3。这就意味着节点可能会崩溃，发送不同和冲突的消息到不同的节点，拒绝中继消息或者表现异常，安全或者运转存在问题。

协议中有两个地方我们可以通过使用本地时钟的超时特性，为不同时性做一些优化：在接收到2/3或者更多预投票（pre-votes）之后（不针对单个区块或者nil）和在收到2/3或更多预提交（pre-commit）以后（不针对单个区块或者nil）。在每一中情形中，我们可以睡眠一段时间用来给延迟的投票一个被接受的机会，因此减少在新的一轮没有提交区块的可能性。时钟不需要在验证者之间同步，因为验证者在观测到2/3或更多的投票时会重置各自的时间。

如果1/3或者更多的验证者崩溃，网络瘫痪，因为任何共识进展需要2/3以上验证者的投票。网络仍然可以读取数据，但是没有新的区块的提交。只要验证者重新上线，他们能够从之前的投票状态开始。共识状态机应该配置一个预写式日志（write-ahead log），这样重新上线的的验证者可以快速回退到之前机器崩溃时的位置，确保没有违反规则。

如果1/3或者更多的验证者是拜占庭，他们能够以多种方式损害系统的安全性。例如，在同一轮提交两个块，并且投票提交这两个区块或者通过通过违反锁定机制在同一高度不同轮提交两个不同的区块。在每一种情形中，有清晰的证据显示哪些验证者是拜占庭节点。在第一个例子中，他们在同一轮签名两个不同的提议，违反规则。在第二个例子中，他们锁定在r-1轮在第r轮提交了一个不同的区块，违反了锁定机制。

当使用经济和治理组件来激励和管理共识，这些额外的责任制保证是具有决定性的。

### ****结论****

Tendermint本身是弱同步，拜占庭容错，状态机复制协议，拥有优化的拜占庭容错和额外的责任制来保证当超过拜占庭容错假设上限时的情形。协议采用round robin的提议者产生方法，用同样的机制跳过一个提议者。多轮投票之间的安全性通过锁机制得到了保障。

本章是关于协议的表述，存在许多有待进一步讨论的重要细节，例如块之间有效的通讯（efficient gossiping of blocks），缓存交易，验证者集合的改变和应用逻辑的接口。这些重要的话题将在随后的章节得到进一步的解释。

## 区块链时代的拜占庭容错：Tendermint（三）

**Tendermint子协议**

上一章中对Tendermint共识的介绍省略了有关用于广播区块（blocks），投票(votes)，交易(transactions)和其他节点信息的gossip协议的一些细节。之所以这样做是为了聚焦在共识协议本身，而不让工程实践的细节分散注意力。本章讲述一种特定的用来填充这些细节的方法，通过把这些组件实现为相对独立的反应器（reactors），在这些反应器中每个对等连接（peer connection）是复用的。

**P2P网络**

在启动时，每个Tendermint节点都会收到一份需要拨号的对等节点的初始列表（initial list）。对于每个对等节点，节点会保持一个持久的TCP连接(persistent TCP connection)，在该连接上以速率受限的方式复用了多个子协议（subprotocols）。消息被序列化为紧凑的二进制表示，并且通过认证的加密协议对连接进行加密。

本章的其余部分的每一小节分别描述了一个独立的反应器，该反应器在每一个对等链接进行了复用。有一个额外的对等节点交换反应器（peer exchange reactor），该反应器允许节点彼此请求其他对等节点地址，并跟踪它们先前连接到的对等节点，以便保持连接其他对等点的最小数量。

**共识Gossip**

共识反应器（consensus reactor）封装了共识状态机(consensus state machine)，用来确保当每个节点的状态发生变化时都向其所有对等节点广播其当前状态。以这种方式，每个节点跟踪所有对等节点的共识状态，只发送对等节点此刻需要的信息，或者对等节点没有的信息，以此来优化消息的gossip过程。对于每个对等节点，本地节点维护来两个例程（routine）来不断检查发送给对其等节点的新信息，即提议（proposals）和投票(votes)。信息应该以“第一稀有”（ratest first）的原则来最大限度地提高gossip的效率，并尽量减少某些信息变得不可用的概率。

**区块数据**

在上一章节中，假设提议信息是包含区块的。然而由于区块是从一个验证节点发出的，并且可以相当大，对于块提议节点来说把数据上传到所有其他节点的方式压力过大；如果把区块分成若干块后再广播出去会快很多。

一个常用来保证数据被安全广播的方法，就是利用Merkle树，此方法也被各种P2P协议所采用，具体方法就是允许每一段数据伴随一个简短的证明（原数据的对数级大小），用来证明该片段是整体的一部分。为了使用这种方法，根据区块大小和验证者的数量，区块被序列化并分割成适当大小的块（chunks），块（chunks）被散列成Merkle树。被签名的提议（proposals）不再包含整个区块，而是只包含Merkle的根哈希（root hash），允许网络通过协作来广播被分割后的块（chunks）。节点每次收到块(chunks)时都会通知它的对等节点，来避免把同一个块对一个节点传输多次，以此来节省带宽。

一旦接收到所有块，就对块进行反序列化和验证，以确保它正确地指向前一个块，并且其各种校验和（用Merkle树实现）是正确的。虽然先前假定验证者在接收到提案（包括块）之前不预先投票，但是通过允许验证者在收到提案之后，在接收到完整的块数据之前进行投票，可以提升性能。这意味着预先投票到最后证明是无效的块也是可以的。不管怎么样，对一个无效区块进行预提交（pre-commit）被认为是拜占庭行为。

节点通过接受块（chunks）的方式同步到最新的高度，以一次一个区块的方式进行同步（progress one block at a time）。

**投票**

在共识状态机中，一个块被提议以后，节点就会开始等待投票（或者本地计时器过期）以往前推进。如果一个节点刚好进入了下一个区块高度的状态下收到一个对上一个块的pre-commits投票，而这个节点如果同时是这轮区块的提议者，它也许会把pre-commits投票当作下一个块的中的LastCommit值存入区块。如果一个对等节点（peer）已经预投票，但尚未预先提交，或者已经预先提交，但尚未进入下一轮，则其被分别发送预投票（pre-votes）或预提交(pre-commits)。如果一个节点是在同步区块的情况下，它将会收到在它当前的高度下预提交（pre-commits）。

**内存池**

交易独立地在内存缓存中被管理，这个缓存沿用比特币的叫法就是内存池。交易被业务逻辑验证合法后加入内存池，并用有序组播算法发送给相邻节点。一个节点会为每个相邻节点维持一个例程（routine），用来确保发送交易的顺序是和自己处理的顺序是一致的。

块提议者会把内存池的有序交易列表中的交易打包进新的区块。一旦一个区块被确认提交，区块中的所有交易都会被从内存池中删除，而留下来的交易会被业务逻辑重新验证，因为账户中的其他交易被确认提交，导致它们的有效性也许会改变。

**区块同步**

共识反应器(consensus reactor)提供一种相对较慢的区块同步方法，这是为了实时共识而设计的，也就是说节点在处理下一个块之前，会等待接受到所有的信息来确认当下的一个块。为了适应一个节点远落后与当前高度的情况，有一个区块链反应器（blockchain reactor），这个反应器允许节点同时并行地下载多个区块，使节点可以百倍于共识反应器的速度来同步区块数据。

当一个节点链接一个新的对等节点时，这个对等节点需要把它当前区块高度发送给对方。节点并行的从所有比自己高度高的对等节点中循序的请求区块，并下载区块加入到自己的区块池(block pool)。另一个例程不断尝试从池中删除块并通过验证和执行将它们添加到区块链中，一次两个块，相对于区块链的最新状态。必须每次同时处理两个区块，因为一个区块的提交确认被包含在下一个区块的LastCommit字段中。

节点持续的查询它所有对等节点的当前高度，同时不断的并行请求区块，直到同步到其所有对等节点的最高高度，一旦同步到最高高度就会停止请求区块，并开启共识反应器。

**总结**

实现Tendermint区块链涉及很多子协议。这些子协议包括包括gossip区块数据和交易的共识数据（投票和提议），和新节点快速同步区块链状态的方法。

## 区块链时代的拜占庭容错：Tendermint(四)

**构建应用**

Tendermint被设计成为可复制的最终确定性状态机的通用的算法。它使用Tendermint套接字协议（TMSP）标准化共识引擎和状态机之间的沟通，使得应用开发者可以用任何语言去构建他们的状态机，并通过这种方式自动复制了基于Tendermit的BFT算法。

**背景**

互联网上的应用程序通常可以表征为包含两个基本要素：

* 引擎：处理核心安全性，网络，复制。这通常像一个网络服务器，如Apache或Nginx，可以构建Web应用程序，或者构建分布式应用程序的一致性算法。
* 状态机：处理交易的实际应用程序逻辑，从引擎接收并更新内部状态。

这种引擎和状态机结构的分离使应用程序开发人员可以用任何编程语言编写想要的应用程序的状态机，在引擎之上可能专门定义其性能，安全性，可用性，支持性和其他考虑因素。与Web服务器及其应用程序不同，后者通常采用的形式是进程通过公共网关接口(CGI)协议使用套接字进行通信。而共识算法一般来说要以更少的可用性或者更少的通用性接口来构建应用程序。像zookeeper，etcd，consul和其他分布式键值对存储应用一样，提供HTTP接口到简单键值对应用程序的特定实例，还有一些更有趣的功能，如原子比较和交换操作并推送通知。但他们没有给应用程序开发人员控制状态机代码本身。

比特币的成功以及源源不断对区块链技术的兴趣产生了要对状态机运行进行高级别的超越共识引擎的控制需求。通过构建更多高级应用直接进入共识，用户，开发商，监管机构等等，可以在任意状态机上实现更大的安全保障，远远超出键值对存储，如货币，交易所，供应链管理 - 治理等等。它之所以引起了这么多人的注意，是因为这个系统巨大的潜力是允许集体执行前执行代码。它实际上是对许多方面的重新发明法律制度，使用分布式共识算法和确定性可执行合同，而不是警察，律师，法官，陪审团和个人喜好。这对人类社会发展的影响是爆炸性的，就像引入民主法治一样。Tendermint旨在提供基本的接口和共识引擎可以构建此类应用程序。

**Tendermint 套接字协议**

Tendermint套接字协议(TMSP)定义了共识引擎和应用程序状态机之间通讯的核心接口。这个接口定义包括一系列的消息类型，使用Google’s Protocol Buffers (一个长度作为前缀通过套接字传输的协议),一系列的消息类型，他们的参数，返回值和用途都在图5.1中，整体的结构和消息流程也在图5.2中。

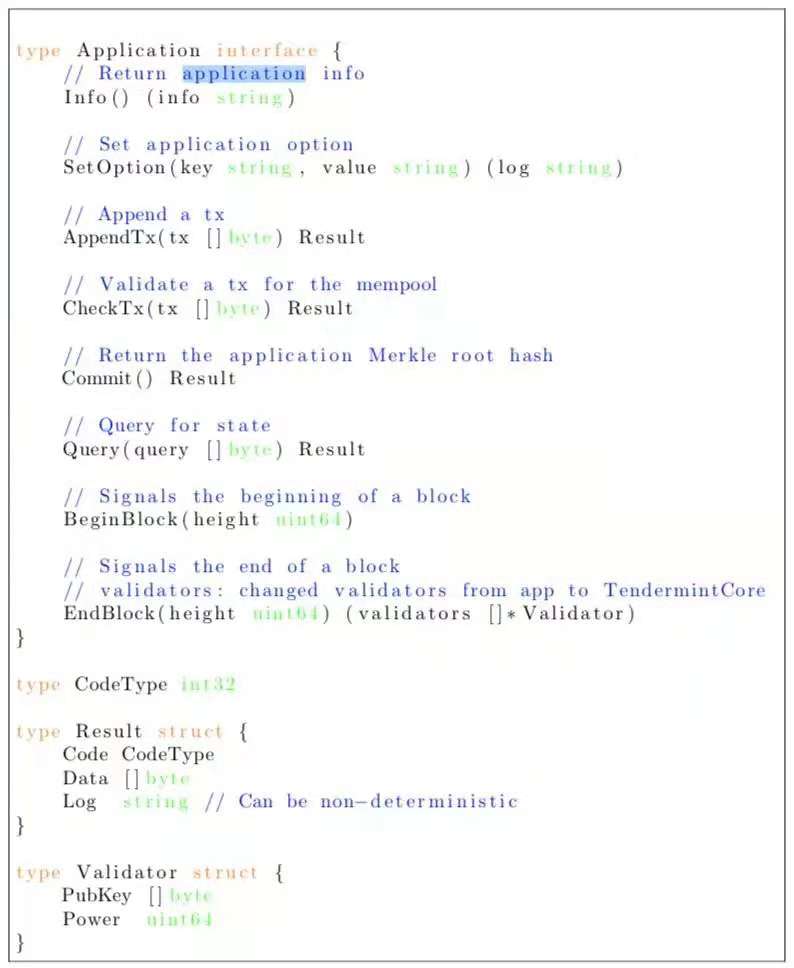


图5.1

Golang定义的TMSP Application interface,TMSP消息被定义为使用Google’s Protocal Buffers,序列化格式是通过TMSP套接字传输之前将长度作为前缀，返回值包括一个代码，和HTTP的状态码类似，代表了任意的错误信息，０表示没有错误，消息在客户端缓存直到一个flush消息被发出，在这个点所有的缓存消息都会被发出。服务端设计的是异步化的，消息返回必须是和请求相匹配的正确的顺序。

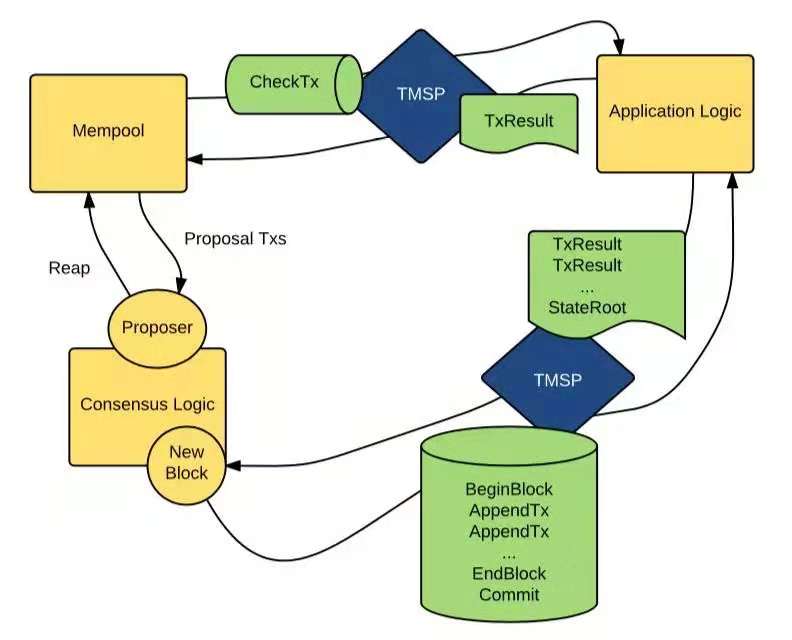


图5.2

共识逻辑与应用程序逻辑进行通信是通过TMSP（套接字协议）。维护两个sockets(套接字），一个用于mempool检查新交易的有效性，另一个用于执行共识的新提交的块。

TMSP协议被实现为一个有序的异步的服务，消息类型请求和返回是成对的，当一个特殊的消息类型出现的时候，清空并通过连接推送缓存的所有消息然后等待消息。

在TMSP协议有两种消息是核心，AppendTx 和Commit，一旦块由共识产生，引擎就会在块上调用AppendTx处理每个块中的交易，将其传递给应用程序状态机处理。如果交易有效，则会导致应用程序中的状态转换。

一旦所有AppendTx调用返回，共识引擎调用Commit，使得应用程序提交到最新状态，并将其永久存储到磁盘。

**分离协议和执行**

使用TMSP可以使我们明确区分共识，或者关于交易顺序的协议，以及它们在状态机中的实际执行情况。特别是，我们首先就顺序达成共识，并且然后按顺序执行交易。这种分离实际上有所提升系统的容错,但是仍然需要总数3f + 1个节点去容忍f个拜占庭式节点的失败，只需要2f + １个节点执行。也就是说，虽然我们仍需要三分之二的多数来排序，我们只需要一半的多数来执行。

另一方面，事实上交易的执行放在排序之后会导致可能是无效的交易，这可能会浪费系统资源。这种情况使用额外的TMSP消息CheckTx解决的，由mempool调用，允许它检查交易是否会对最新的状态有效。但请注意，事实上块中的一次性提交会引起CheckTx处理消息的复杂性。特别是，应用程序希望维护第二个状态机只执行主状态机相关的交易有效性的规则。第二个状态机由CheckTx消息更新状态并在每次提交后重置为最新的已提交状态。在本质上，第二个状态机描述了交易池的过滤规则。

在某种程度上，CheckTx可以用作｀乐观的执行｀返回并将结果发送给交易发送者，但结果可能是错误的，比如一个带有冲突交易的块在利息交易之前提交。这种｀乐观的执行｀是扩展BFT系统的方法的关注焦点，可以很好地适用于交易之间冲突很少的应用程序。与此同时，它增加了客户端的额外复杂性，因为可能需要处理无效的结果。

**微服务架构**

采用分离关注点作为应用程序设计的策略是从普遍上来看是明智的做法。特别是，许多当今的大规模应用部署采用微服务架构。其中每个功能组件实现为独立的网络服务，通常封装在Linux容器中（例如使用Docker）以提高部署效率，可伸缩性和可升级性。

在Tendermint共识之上运行的应用程序通常会被分解成微服务,例如，许多应用程序将使用一个键值对存储用于存储状态。为了利用数据存储的优势和某些特有功能，将键值存储作为独立运行的服务很常见，如高性能数据类型或默克而树。  
应用程序的另一个重要的微服务是治理模块，它管理某个TMSP消息子集，启用该应用程序控制验证器集更改。这样的模块可以在BFT中变成强大系统治理范式。

某些应用程序可能为用户使用本机货币或帐户结构，因此，提供支持基本元素的模块是有用的，例如，处理数字签名和管理账户动态。用微服务构成复杂TMSP应用程序例子还有很多。实际上，人们甚至可以构建一个可以启动的，包含了使用交易中发送的数据的子应用程序的应用程序。例如，在交易中包含了一个存储docker镜像的哈希，以便从一些文件存储后端pull下来运行并作为未来的子应用程序运行共识中的交易以导致其运行。这是以太坊使用的方法，允许开发人员将一些代码部署到网络中可以通过上面的方式触发在以太坊虚拟机中运行交易（智能合约)，以及IBM最近的OpenBlockChain（OBC）项目，它允许开发人员在交易中发送完整的docker上下文，定义运行任意代码以响应发往他们地址的交易的容器。

**确定性**

关于使用TMSP构建应用程序的最为严格的忠告是他们必须是确定性的，也就是说，对于复制状态机而言不保证安全。每个节点在执行时针对同一状态的同一交易必须获得相同的结果。这不是Tendermint的独特要求。比特币，raft，以太坊，任何其他分布式共识算法，以及锁步等多人游戏必须严格具有确定性，以免达成共识失败。编程语言中有许多非确定性的来源，最明显的是通过随机数和时间，但也可以通过使用浮点精度，并通过哈希表迭代（一些诸如Go之类的语言强制对哈希表进行随机迭代程序员明确何时需要有序数据结构）。对确定性机制的严格限制，以及确定性在每种主流编程语言中的缺失，促使以太坊开发自己的，图灵完备的，完全确定的虚拟机，它构成了应用程序开发人员在以太坊区块链之上构建应用程序的平台。虽然确定性，它有许多特别之处，例如32字节堆栈字，存储键和存储值，不支持字节位移操作，一切都是大数算术。确定性编程在实时世界中得到充分研究，例如锁步，多方游戏。这类游戏构成了另一个复制状态机的例子。并且在许多方面与共识算法非常相似。鼓励使用TMSP构建的应用程序开发人员进行学习使用他们的方法，并在实现应用程序时要小心。使用函数式编程语言和证明方法可以启用正确程序的构建。在另一方面，编译器也在构建以将可能的非确定性程序转换为规范化确定性的。

**终止性**

如果说确定性是为了保证安全很严格，那么交易执行的终止性就是为了严格的保证存活。然而，它不是一般的可以确定给定程序是否因单个输入而停止，更不用说所有这些问题，这个问题被称为停机问题。

以太坊的虚拟机通过计量解决问题，具体来说，在执行中的每个操作收费。这样，交易就保证在发起者资金不足时终止。这样的计量在更一般的情况下，可以通过编译程序的编译器实现计量他们自己的版本。

没有显着的开销就很难解决这个问题。在本质上，验证器无法判断执行是处于无限循环中还是只是缓慢，但几乎完成。可以使用Tendermint共识协议决定交易超时，超过三分之二的验证人必须同意交易超时，因此被视为无效（即对状态没有影响）。但是，我们没有在这里进一步实践这个想法，留待未来的工作。同时，预计应用程序将在进行全面测试之前部署在任何共识系统中，监控和治理机制在达成共识的情况下，将用来恢复系统失败。

**举个例子**

在本节中，将介绍和讨论越来越复杂的TMSP应用程序，特别是CheckTx和管理mempool。

**Merkleeyes**

TMSP应用程序的一个简单示例是基于Merkle树的键值存储。Tendermint提供Merkleeyes，这是一个包装的TMSP应用程序，一个自平衡的Merkle二元搜索树。交易的第一个字节确定交易是get，set还是delete操作。对于get，delete操作，剩余的字节是键(key)。对于set操作，其余字节是包含键和值的序列化列表。Merkleeyes可能使用CheckTx的简单实现对交易进行解码，以确保其格式正确。也可以实现更高级的CheckTx，get和delete操作对未知key无效。调用Commit后，将添加最新更新进入Merkle树，重新计算所有哈希值，树的最新状态也在磁盘上提交存储。

请注意，Merkleeyes被设计为其他TMSP应用使用的基于Merkle树的键值存储的模块。而不是单独的TMSP应用程序，虽然TMSP接口的简单性使它适用于两者。

**货币应用**

一个更重要的完整实例是一个简单的货币，用的账户结构由以太坊创立，每个用户都有一个公钥和一个基于该公钥的带有余额的账户，该帐户还包含一个序列号，等于帐户发送的交易数量。交易可以从账户发送资金，如果它们包括正确的序列号，并由正确的私钥签名。没有序列号，系统容易受到重放攻击，可以重播从帐户中扣除帐户的已签名交易，从而导致借记多次发生。此外，防止重放攻击在多链环境中，交易签名应包括网络或区块链标识符。

一个支持货币职能的应用程序自然具有比键值存储更多的逻辑。特别是，某些交易明显无效，例如签名无效，序列号不正确或发送超过发起人帐户余额的金额。这些情况都可以通过CheckTx检查。

此外，必须在CheckTx中维护补充应用程序状态是为了在更新时更新序列号和帐户余额当同时在mempool中有多个涉及相同帐户的交易时。调用commit时，将重置补充应用程序状态到最新的提交状态。任何仍在mempool中的交易都可以通过CheckTx重播最新状态。

**以太坊**

以太坊使用已经描述的机制来过滤掉mempool中的交易，但它也在虚拟机中运行一些交易，它会更新状态并返回结果。虚拟机执行没有在CheckTx中完成，因为它们包含在块中，它更昂贵且严重依赖交易的最终顺序。

**总结**

TMSP提供了一种简单而灵活的方法来构建任意应用程序，可以用任何编程语言，继承复制自Tendermint一致性算法的BFT状态机。它扮演的角色大致类似于共识引擎和一个应用程序。例如，CGI扮演为Apache和Wordpress。但是，应用程序开发人员必须特别注意确保他们的应用程序是确定性的，并确保交易执行终止。

## 区块链时代的拜占庭容错：Tendermint(五)

目前为止的章节，论文阐述了Tendermint共识协议和应用环境相关的基础要素。现实世界中系统的关键要素，例如验证人集合（validator set）的变更管理、故障恢复机制等，还尚未讨论。

这个章节提出了一种解决这些问题的方法，正视共识系统的治理任务。当验证人集合包含更分散的代理人集合时，维护网络的有效治理将变得非常重要。

**Governmint**

治理（governance）的基本职能是筛选提议行为，通常是以一种投票的形式来完成的。治理的实现作为软件的最基础模块，它允许用户发起提议，对其进行投票，并对投票进行计数。提议可能是程序化的，在这种情况下，它们可能在成功的投票后进行自动执行；亦或并非程序化的，这种情况下它们的执行依赖于手工运行。

为了在Tendermint中启用特定的操作，比如验证人集合变更、升级软件，而实现了一个叫Governmint的治理模块。Governmint是一个最小的、可行的治理应用程序，它支持多组实体，每一个都是可以内部投票的提议，其中一些可能产生程序化执行的行为，就像验证人集合变更，或Governmint的自动升级等。

系统利用数字签名对投票验证人进行身份验证，并可能使用各种可能的投票方案。特别有趣的是二次投票方案，投票的成本是投票权重的二次方。这点被证明了更加能满足投票验证人的选择权。

**验证人集合变更**

验证人集合的变更是现实世界共识算法的一个重要组成部分，许多以前的方法论都没有这方面的说明，也或许像某种魔法（black art）一样失传了。Raft算法为验证人集合变更阐述了一个健全的协议，这需要使用一个新的消息类型，使更改通过协商一致。Tendermint采用了类似的方法，不过它是通过使用EndBlock消息的TMSP接口标准化的，该消息在所有AppendTx消息之后运行，但在提交之前。如果一个交易或一组交易包含在一个块中，其目的是更新验证人集合，那么应用程序可以通过指定它们的公钥和响应EndBlock消息的新的投票权来返回要更新的验证人列表。验证人可以通过将其投票权重（voting power）设置为零来删除。这为应用程序提供了一种通用的方法来更新验证人集合，而无需指定交易类型。

如果高度为H的区块返回了一组已更新的验证人集合，那么在H+1高度的区块中将反映出这个变更。但是请注意，在H+1这个高度区块的LastCommit方法必须使用高度为H区块的验证人集合，因为它可能包含了已剔除验证人的签名（signatures）。

投票权重的变更应用到H+1高度的区块，如此下一个出块提议人将受到这个更新的影响。尤其是本应是下一个提议者的验证人已被剔除的情况。这里Round-robin算法可以优雅地处理这个问题，简单有序地转移到下一个出块提议人。因为同一个区块在至少三分之二的验证人节点上被复制，同时Round-robin是确定性算法，所以他们都将进行相同的更新，并等待下一个提议人。

**惩罚拜占庭验证人**

比特币设计中的一个显著的特点是它的激励机制（incentive structure），就目前而言，协议的目标是通过奖励验证者来激励他们正确行事。虽然这在比特币的共识协议背景下是有意义的，但更好的奖惩制度也许需要提供一种健壮性反激励机制（strong dis-incentives），如此对于网络验证人来说利益共享、风险共担，而不是一些温和的机会成本（opportunity cost）。

在Tendermint中的奖惩可以使用一种方法来实现，这就是由Vitalik Buterin以一种名为Proof-of-Stake的协议来提出的。从本质上来说，验证人必须准备一份安全保证金（他们必须持有一定的股份）才能参与到共识中。如果他们被发现有双重签名的提议或投票行为，其他验证人就可以以交易的形式发布这些违规行为的证据，通过剔除违规者，应用程序状态可以使用交易的形式来变更验证人集合，并惩处他的保证金。这就产生了将显性经济成本与拜占庭行为相关联的影响，进而使得诸如贿赂三分之一甚至更多验证人的拜占庭节点（Byzantine）考量自己的违规成本。

请注意，一种共识协议可能会指定更多的可以被惩罚的行为，而不仅仅是双重签名。我们尤其喜欢惩罚任何带有强烈信号的不当行为——典型如 任何上报的状态变更都不是基于其它节点已上报的状态。例如在Tendermint实现版本中，所有的预提交必须带有证明他们的波卡锁，验证人可能因广播不合理的预提交而受到惩罚。但是请注意，我们不能仅仅因为任何的意外行为而严厉惩罚——例如，当一个验证人在不是他的当前回合而发起提议，可能就是一种异步抢占（pre-empt asynchrony）或节点奔溃（crashed nodes）的情况为依据来优化。

**软件升级**

Governmint也可以在一个去中心化的网络（decentralized network）上以一种自然的方式商议软件升级问题。众所周知，在一个公共网络上的软件升级是一个具有挑战的事情，需要慎重规划去保持因没有第一时间升级的用户的向下兼容性（backwards compatibility），也不能由于软件引入缺陷、移除产品特性、增加复杂度、或者，也许最糟糕的是未经许可而自动升级而让忠实的用户感到不快。

在比特币上升级一个去中心化的共识系统的挑战是显而易见的。以太坊已经完成了一项成功的、非向下兼容的升级，是因为它有强大的领导力和意见统一的社区；而比特币由于带有恶意、分化的社区以及缺乏强大的领导力，尽管大量的软件工程问题暴露，却不能完成一些必须的升级。

由于变更的适用范围不同，区块链的升级通常可分为软分叉（soft forks）和硬分叉（hard forks）。软分叉意味着向下兼容，并使用协议中的自由度，没有升级的用户可能会忽略这一点，但是这可以为已升级的用户提供新的特性。另一方面，硬分叉是不向下兼容的升级，就比特币而言，硬分叉可能会导致安全方面的破坏，而就Tendermint而言，则会导致整个系统停止运行。

为了应对这一问题，比特币软件的开发者们推出了一系列软分叉，验证人可以通过在新的区块中发出信号来投票。一旦验证人的某个阈值发出更新的信号，它就会自动在整个网络中生效，至少对于支持更新的软件版本的用户来说是这样的。比特币系统的实用性在这些软分叉的基础上得到了极大的提高，而且可以预见这一切将会继续。有趣的是，社区未能对软件成功的硬分叉，一方面引起了人们对该系统长期稳定性的担忧，另一方面也引起了对系统恢复舞弊治理的激励和启发——这是一种无法治理的能力。

考虑到当今世界明显的政府贪腐现象（government corruption），有很多理由采用后一种方法（译者注：去中心化系统治理自治的方案）。如此，由密码学和分布式共识提供了一套新的工具，很大程度上能够实现一定透明度和问责制，这在现代政府的纸质版世界里是无法想象的，甚至在传统网络（traditional web）中的数字世界（digital world）里也不存在，因为传统网络严重缺乏足够健壮的身份验证系统（suffers tremendously from a lack of sufficiently robust authentication systems）。

在使用Governmint的系统中，开发者将是区块链上的可识别个体，可以为系统升级提交提议。这种机制与Github上的Pull Request非常相似，只是它被集成到一个实时运行的系统中，并通过共识协议达成一致。应该使用可配置的更新参数来实现客户端，这样他们就可以指定是系统自动更新还是先通知客户端。

当然，任何未经彻底审查的软件升级对系统来讲都可能形成风险，一般来说应该采取保守的方法来升级。

**故障恢复**

在某些诸如交易日志中的歧义，或者系统停止运行等紧急事件的情况下，传统的共识系统几乎不提供任何保证，通常需要通过人工干预来解决。

Tendermint确保那些违反安全的验证人可以被标识出来，如此，那些可以连接到至少一个诚实节点的任意客户端，能够明确辨别谁是不诚实的验证人，并由此选择跟随诚实的节点进入一个新的链，其中包含一个不包括拜占庭节点的验证人集合。

例如，假设有三分之一甚至更多的验证人违反了锁定规则（locking rules），导致在高度H提交了两个区块，诚实的验证人可以通过广播（gossipping）所有的投票来确定是谁在重复签名（double-signed）。在这点上，它们不能使用协商一致协议，因为已经违背了基本错误假设。注意，此时能够在高度H搜集投票意味着一种很强的假设场景，在故障期间网络的连通性和可用性，如果p2p网络已经无法提供这个能力，则可能需要验证人使用诸如社交媒体和高可用服务的替代方法来传递凭证（evidence）。只要超过三分之二的诚实节点收集到所有的凭证，就可以启动一条新的区块链。

另一种方法是，修改Tendermint协议来使得预提交（pre-commits）带有波尔卡锁，这能够确保分叉的责任人（those responsible for the fork）会立即受到惩罚，而且不需要额外的公布时间。这项修改有待今后的工作来实现。

Governmint更复杂的应用可能更适合处理各种故障细节，比如永久性崩溃（permanent crash failures）和私钥泄露（compromise of private keys）等。不管怎样必须仔细考虑这些问题，因为它们可能破坏底层协议的安全保障。我们把对于这些问题的研究留给未来的工作，但请注意在理解区块链从故障恢复的能力方面，它所处的社会经济环境的重要性。

无论故障恢复如何进行，能否成功还是取决于客户端集成。如果客户端不接受新链，那么它提供的服务实际上是离线的。因此，客户端必须知道指定区块链用于恢复的具体规则。在上述安全违规的情况下，它们还必须搜集凭证来确定要移除哪个验证人，并通过其余的验证人重新计算状态。在没有在线的情况下，保持治理能力。

**最后结论**

治理能力是分布式共识系统的一个关键要素（critical element），尽管主流的治理系统仍未得到充分理解。Tendermint提供了一种称为Governmint的TMSP模块，旨在促进对分布式系统的基于软件治理（software-based governance for distributed systems）的更多实验。

## 区块链时代的拜占庭容错：Tendermint（六）

**客户端的考量**

这章回顾了客户端的考量（consideration），该客户端直接与一个基于Tendermint的应用进行交互。

**发现**

网络节点发现机制(network discovery)是简单地通过TCP连接向种子节点拨号的方式进行。P2P网络使用了认证加密技术，然而验证者公钥需要通过带外数据验证。确实，在这些系统中，创世状态（genesis state）必须通过带外同步，理想情况下，这也是唯一需要进行通讯的数据，因为其已经包含了验证者在认证加密中使用的公钥，这与共识中用来签名投票的公私钥不同。

对于一个会随时间变化的验证者集合，可以通过DNS来注册所有的验证者，可以在他们真正成为验证者之前注册一个新的验证者，可以将它们作为验证者移出后将其删除。或者，验证者位置可以通过其他的拜占庭容错分布式数据存储来注册，包括可能的其他的Tendermint集群。

**广播交易**

作为一个通用的应用层平台，Tendermint提供了一个简单的接口用来为客户端广播交易。通用的范例是客户端通过代理连接到Tendermint共识网络中，代理既可以是本地的也可以是远端的。代理可以作为一个网络中的非验证者节点，用来同步共识和处理交易，但不签名投票。代理使得客户端交易可以通过gossip协议，快速地广播到网络中。

一个节点只需要通过连接网络中的另外一个节点来广播交易，但是，在默认情况下，它将连接许多节点来最小化交易丢包的可能。交易会被送进内存池（mempool），通过gossip协议借助内存池反应器（mempool reactor）缓存在所有节点的内存池中，所以最终他们之一会被包含在一个区块内。

注意到交易并不会真正地参与到状态的执行直到交易被打包进一个区块中，所以客户端除了被告知该交易已经被包含进内存池中，并且广播到了其他节点之外，并不能马上得到一个预期的结果。客户端应当注册一个代理，这个代理在交易最终被提交的时候会发出相应的通知。

客户端并不需要连接到当前的提议者（proposer），因为最终任何一个验证者都会轮流成为下一个提议者。然而在高负载情况下，客户端优先广播到下一个提议者能够减少交易延迟。否则，交易需要快速地被gosssiped到每一个验证节点。

**内存池**

内存池负责在交易被包含到区块里之前，缓存交易到内存中。其行为比较微妙，对整个系统架构形成了许多挑战。首先，如果内存池缓存所有的交易将很容易受到dos攻击。大多数区块链解决这个问题通过使用原生货币，只允许附带交易费用的交易驻留在内存池里。

在一个更一般化的系统中，像Tendermint，并没有用来作为交易费用的原生代币，系统必须建立更严格的过滤规则，并且依赖更智能的客户端来重新提交之前没有成功提交的交易。这种情形甚至更微妙，因为用来过滤内存池中交易的规则集合一定是应用程序自己的功能。因此，TMSP的CheckTx消息类型可以被内存池根据根据应用程序的当前状态，用来决定是否接受该交易。

处理应用程序的当前状态并不是一件简单的工作，像许多应用程序例子一样，其被留给应用程序的开发者来解决。无论如何，客户端需要监控内存池的状态（即没有确认的交易）来决定他们是否需要重新广播他们的交易，这些交易可能是高度并行的，某个交易的有效性依赖于之前处理的交易。

**语义学（Semantics）**

Tendermint的核心共识算法仅提供了至少一次的语义（at-least-once semantics），也就是说系统会遭受到重放攻击，同样的交易可能被提交多次。然而，许多用户和应用程序希望对数据库的更强的保障（guarantees）。Tendermint系统的灵活性使得应用程序开发者可以实现这些功能。通过利用CheckTx消息类型并且相应地管理应用程序的状态，应用程序开发者能够提供一个适合他们并且满足他们需要的数据库语义。例如，正如在本系列第四篇“构建应用”所讨论的那样，通过使用附加序列号的基于账户的系统能够减轻重放攻击，改变语义从至少一次到仅此一次。

**读**

客户端发起读请求到之前用来广播交易的代理节点。代理总是可以进行读操作，即使网络停顿下来。然而，当整个网络出现分区，该代理读取的数据可能不是最新的。

为了避免上述情况，读请求可以被作为一个交易发送出去，假设应用程序允许这样的查询。通过使用交易，能够保证读取的一定是最新提交的数据，即当读交易被提交到下一个区块中。当然这样做的成本要比简单的查询本地代理的状态要高。当然可以通过一些启发式的方法来测定读取的数据是否是最新的，比如这个代理是否与网络中的其他节点连接良好，这个代理是否在出块或者其投票是否正常，但是没有方法能够替代实际的交易读取的方法。

**轻量客户端证明**

区块链相对于传统数据库的主要创新在于引入了梅克尔哈希树，其中便用到了轻量级客户端证明（light client proofs）。所谓的轻量客户端证明是梅克尔树上的一条路径，使得客户端可以验证某一个key-value键值对，在已知根哈希的情况下。梅克尔树根哈希(root hash)的状态被包含在区块头中，这样仅通过最新的区块头便可以验证当前状态的任意一部分。当然，为了确保这个区块头是有效的，他们必须要么通过验证整个链，或者与最新的验证者集合保持同步，并且依赖于经济激励来保证状态转换是正确的。

**结论**

尽管在客户端中需要考虑到基于区块的提交特性和内存池的行为，Tendermint网络中的客户端功能还是类似于其他分布式数据库。除此之外，客户端必须被设计成特定的应用程序。尽管这添加了一些复杂度，它却赋予了客户端极大的灵活性。

## 区块链时代的拜占庭容错：Tendermint (七)

### ****软件实现****

Tendermint采用Go语言实现，代码在https://github.com/tendermint/tendermint。Go是一个类C的语言，带有强大的标准库，可以很轻易的启动大量轻量级并发，并且提供了简单并安全的开发环境。

该代码使用了许多软件包，这些软件包足够模块化，可以作为自己的库进行隔离。这些软件包大部分由Jae Kwon编写，包含错误修复，测试以及作者提供的临时功能。以下小节将介绍这些软件包中最重要的软件包。

### ****二进制序列化****

Tendermint使用针对简单性和确定性而优化的二进制序列化算法。它支持所有整数类型（包括用一个字节长度前缀编码的varints），字符串，字节数组和时间（unix时间，精度为毫秒）。它还支持任何类型的数组和结构体（编码为有序值列表，忽略键）。它受到Go类型系统的启发，特别是它对接口类型的使用，它可以作为许多具体类型之一实现。可以注册接口，并且每个具体实现在其编码中给出前导类型字节。请在https://github.com/tendermint/go-wire中查看更多细节。

### ****加密****

诸如Tendermint之类的共识算法使用三种主要加密原语：数字签名，散列函数和经过身份验证的加密。虽然存在这些原语的许多实现，但选择用于企业软件的加密库并非易事，特别是考虑到世界上最常用的安全库OpenSSL的严重不安全性。

造成加密系统不安全的原因是政府机构（如美国国家安全局）可能会故意破坏其安全属性，后者与NIST合作，设计并标准化了当今使用的许多最流行的加密算法。鉴于这些机构明显不合法，例如，爱德华·斯诺登所显示的，以及试图破坏公共密码标准的历史，密码学界的许多人更喜欢使用在开放式设计的，学术环境下的算法。类似地，Tenminmint仅使用这样的算法。

Tendermint采用RIPEMD160作为其hash算法，此算法会产生一个20字节的输出。它被用于交易的Merkle树，验证人的签名，计算区块hash。Go语言在其扩展库中提供了算法的实现。RIPEMD160还被比特币用作从公钥导出地址的两个散列函数之一。

作为其数字签名方案，Tendermint在ED25519椭圆曲线上使用Schnorr签名。ED25519由Dan Bernstein公开设计，旨在实现高性能且易于实施而不会引入漏洞。Bernstein还引入了NaCl，这是一个高级库，用于使用ED25519曲线来做认证加密。Tendermint采用Go自身扩展库的实现。

### ****Merkle Hash树****

Merkle树的功能与其他基于树的数据结构非常相似，其附加功能是可以生成树中键的成员资格证明，该证书的大小与树的大小成对数。这是通过成对地递归连接和散列密钥来完成的，直到只剩下一个散列，即树的根散列。对于树中的任何叶子，从它到根部的一串哈希作为其成员资格的证明。这使得Merkle树对于p2p文件共享应用程序特别有用，其中大文件的各个部分可以被验证为属于该文件，而不需要拥有所有数据。Tendermint利用此机制来处理在网络Gossip区块的部分，其中根哈希包含在提议区块。

Tendermint还提供了一个自平衡的Merkle二叉树，以AVL树为模型，作为一个名为Merkleeyes的TMSP服务。IAVL树可用于存储动态大小的状态，允许以对数时间查找，插入和删除。

### ****RPC****

Tendermint提供HTTP API接口，用于查询区块链，网络信息，共识状态和广播交易。同样的API可以通过三种方法进行访问：利用URI编码参数的GET请求，利用标准JSONRPC的POST请求，利用标准JSONRPC的Websocket。Websockets是高事务吞吐量的首选方法，是接收事件所必需的。

### ****P2P网络****

Tendermint使用的P2P子协议在本系列第三篇“Tendermint子协议”中有更全面的描述。

### ****反应器****

Tendermint节点是由多个并行的反应器组成，每个反应器管理状态机通过网络向对等节点的信息发送和收取，就如本系列第三篇描述的一样，反应器根据锁定共享数据结构来进行同步，但同步点保持在最小，因此每个反应器大部分情况下和别的反应器是并行的。

### ****内存池****

内存池反应器管理内存池，它将交易在打包入块和提交前进行缓存。内存池使用应用程序状态机的子集来检查交易的有效性。交易保存在并发链表结构中，允许安全写入和许多并发读取。新的有效交易被加入到列表的末尾。每个对等节点的例程遍历列表，按顺序将每个交易发送给对等节点一次。列表同样被扫描用于为新提议收集交易，当新提议块被提交确认后列表会按照如下规则进行更新：删除被确认提交的交易，未确认的交易重新通过一次CheckTX方法的确认，并删除那些无效的交易。

### ****共识****

共识反应器管理共识状态机，它处理提议，投票，锁定和块的实际提交。状态机使用一些持久的例程来管理，这些例程排序接收的消息并使它们能够被确定地重放以调试其状态。这些例程包括readloop，用于读取接受消息的队列，timeoutLoop用于注册和触发超时事件。

共识状态机中的转换是在收到完整的提议和区块时，或者在给定轮次中收到超过三分之二的预选票或预先提交时进行的。在广播中的提议，区块数据，或者投票的交易结果被加入internalQueue队列，并和从对等节点接受的信息一起顺序的被readLoop处理。这使内部消息和对等节点消息在共享状态机的输入上处于平等的地位，但允许更快地处理内部消息，当它们不与来自对等节点的消息位于同一队列中。

### ****区块链****

区块链反应器以一种比共识反应器更快的技术来同步区块链。也就是说，验证者请求增加区块高度直到所有对等节点都没有更高高度为止。块在块池中收集并通过工作例程同步到区块链，该例程定期从池中获取块并针对当前链验证它们。

一旦区块链反应器完成同步，它就会切换到共识反应器。

### ****总结****

Tendermint实现利用了语言的并发原语，垃圾收集和类型安全，提供了一个清晰，模块化，易于阅读的代码库以及许多可重用的组件。如下一章所示，该实现获得了高性能并且对许多不同类型的故障具有稳健性。

## 区块链时代的拜占庭容错：Tendermint (八)

**性能和容错**

Tendermint被设计为拜占庭容错的状态机复制算法。只要不超过总量的三分之一的验证人是拜占庭节点，就能保证网络的安全性，并且只要网络消息最终得到传递，即使对于gossiping提议的网络同步性假设很弱，它也能保持运行。在发生此类故障时，Tendermint共识的落实不会影响安全性，它会对性能产生最小的影响，并且可以快速恢复。 可以通过几种关键的方式来评估Tendermint算法的性能。最明显的衡量指标是块提交时间，它是衡量最终延迟的指标，也是衡量交易吞吐量的网络容量指标。我们在网络上收集每个分布在地球上的验证人节点的测量值，其中验证人的数量范围是2的倍数，从2到64。

**概览**

可以使用https://github.com/tendermint/network\_testing 上的代码库重现本章中的实验。 所有实验都在t2.medium或c3.8xlarge的Amazon EC2实例上运行的docker容器中进行。 t2.medium有2个CPU和4GB RAM，c3.8xlarge有32个CPU和60GB RAM。 实例分布在七个数据中心，跨越五大洲。 负责生成交易的第二个docker容器在每个实例上运行。 交易大小为250字节（包括一些32或64字节的哈希和签名的合理大小），并且构造为可调试的方式，也便于快速生成指令，并且也包含了一些随机性。 因此，前导字节是表示该实例的交易编号和验证人索引的Big-Endian编码整数，从操作系统中随机抽取尾随的16字节，并且中间字节仅为零。

网络监控工具用于维护每个验证人节点的Tendermint RPC服务器的活跃websocket连接，并在第一次收到新的已提交块时使用其本地时间作为该块的正式提交时间。 首先在没有监控的情况下运行实验，方法是复制验证人节点中的所有数据进行分析，并使用提交块的2/3验证人节点的本地时间作为提交时间。 使用监控要快得多，适合在线监视，只要只有块头信息（而不是整个块）通过websocket传递，就不会影响到结果。

使用docker-machine工具可以轻松管理远程计算机上的Docker容器，并且网络测试存储库提供了一些工具，这些工具利用Go语言的并发功能在多个远程计算机上同时对docker容器执行操作。

每个验证人节点直接相互连接，以避免网络拓扑的混淆影响。

对于涉及崩溃故障或拜占庭行为的实验，故障节点的数量取决于Nfault = ⌊(N − 1)/3⌋，其中N是验证人节点的总数。

**吞吐量和延迟**

本节描述了测量Tendermint在非对抗条件下（编者注：作者意指不考虑拜占庭节点等容错的情况）的原始性能的实验，其中所有节点都在线并同步，并且没有为异步做出调整。 也就是说，使用了极高的TimeoutPropose（10秒）参数，并且所有其他超时参数都设置为1毫秒。 此外，所有mempool活动都被禁用（没有交易的gossiping亦或在提交完成后的复检），进程内的nil应用程序为了绕过TMSP。

实验基于验证人集合运行，集合大小从2增加到64，块大小从128到32768加倍。每个验证人节点都预先加载了交易。每个实验运行16个区块。

从图1中可以看出，Tendermint每秒可以轻松处理数千个交易，大约有一秒的出块延迟，尽管每秒大约有一万个交易处理容量限制。 16384个交易的块大小约为4 MB，并且网络带宽分析显示每个连接容易达到20MB / s以上，但是对日志的分析表明，在达到高的区块高度时，验证人节点可能花费超过两秒的时间来等待块部件。 此外，如图2所示，单个数据中心的实验表明，可以实现更高的吞吐量，而在更好的机器上进行的实验表现出更一致的性能，从而减轻了容量限制，如图3所示。 我们将此容量限制的进一步调研留待未来的工作。

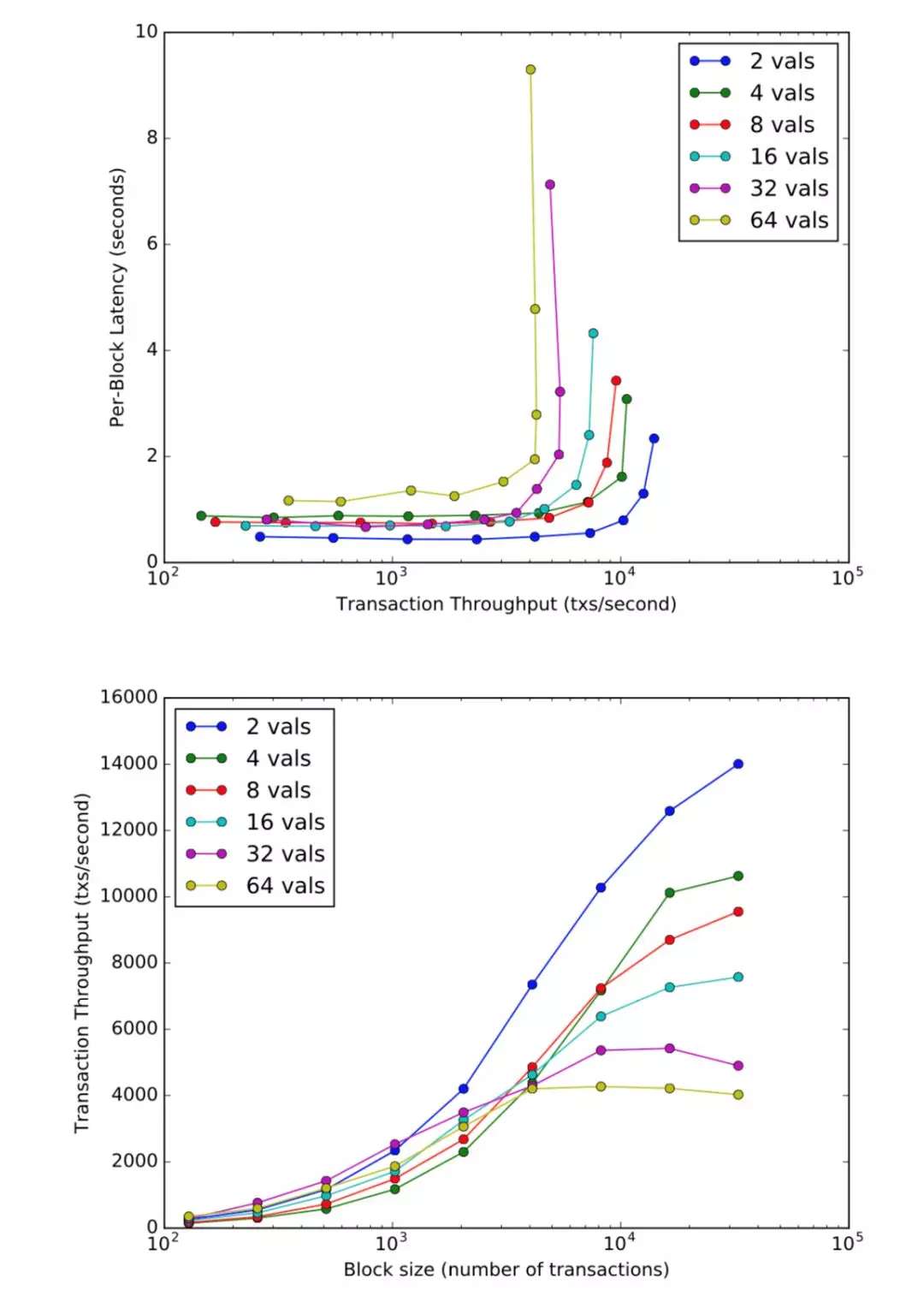


图1：延迟-吞吐量权衡。 较大的块会导致交易吞吐量逐渐减少，最终容量约为10,000 txs / s

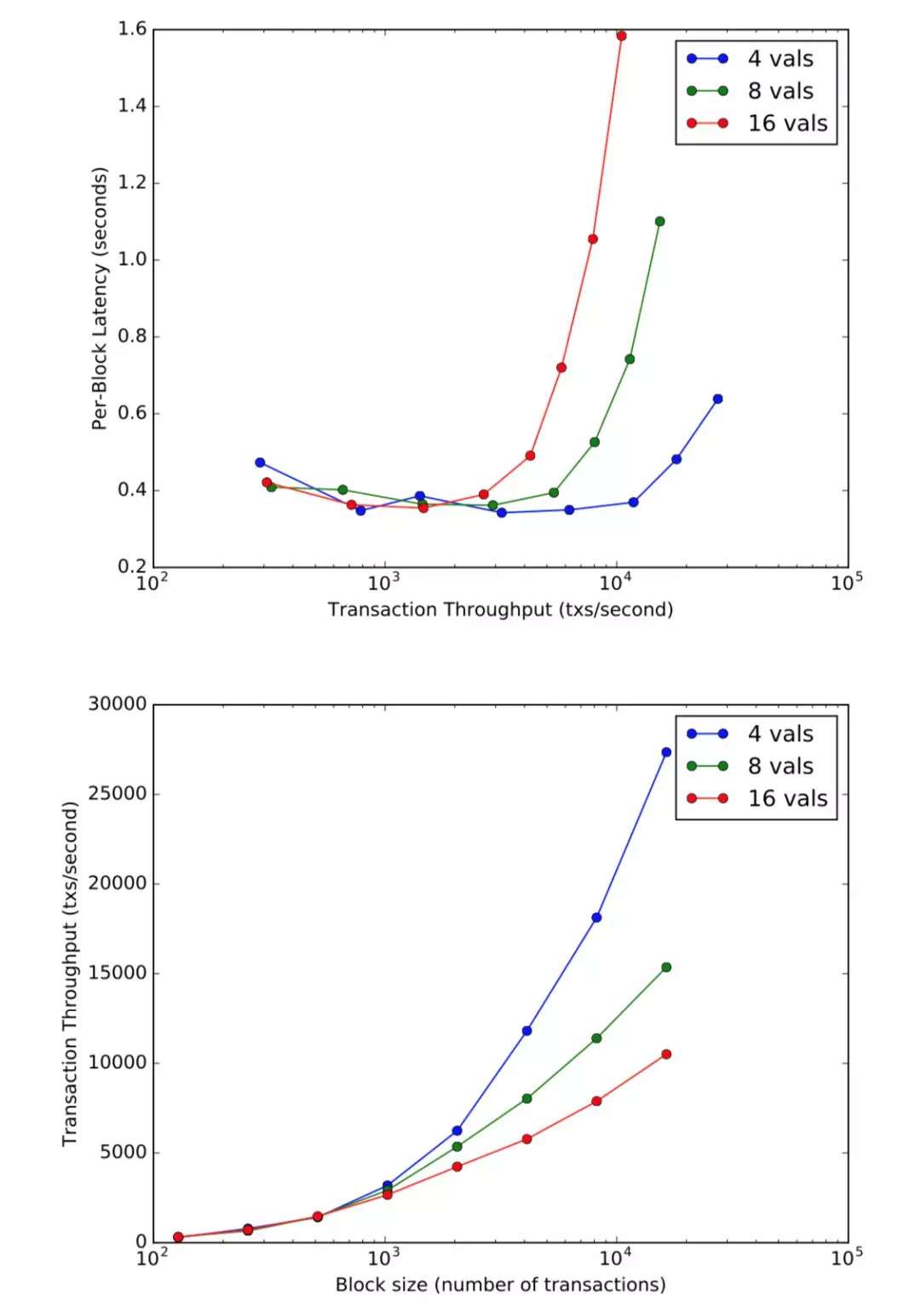


图2：单个节点的情况。 当消息不需要通过网络时，Tendermint每秒可以进行数万次交易。

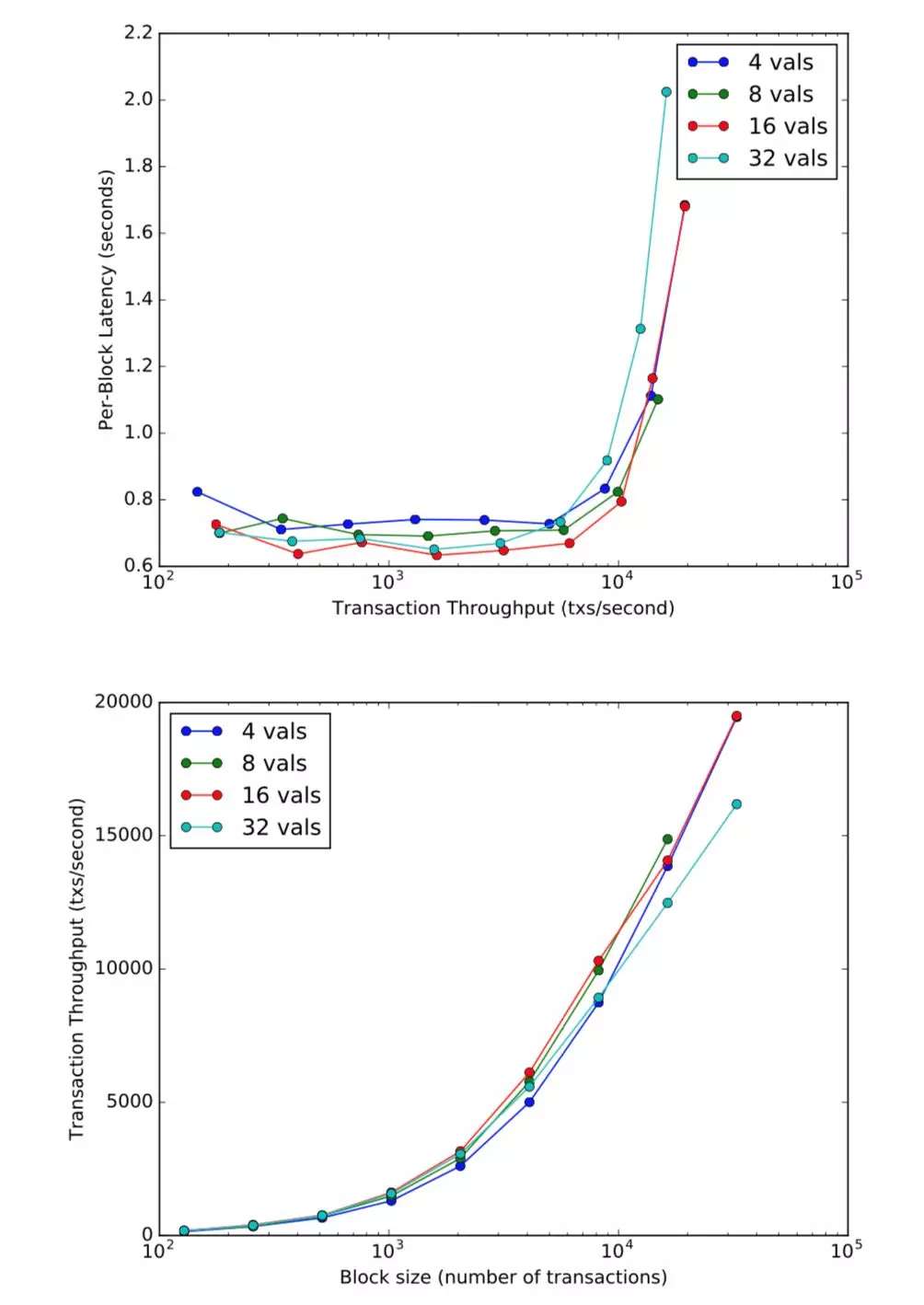


图3：大型机。 使用32个vCPU和60GB RAM，交易吞吐量随块大小线性增加，从而减轻了较小机器上的容量限制。

在后续的实验中将注入各种形式的故障并呈现延迟统计。 对于不同的TimeoutPropose值，以及块大小为2048个交易，每个实验运行的验证人集合大小从4增加到32。

**死机故障**

为了评估遭受节点崩溃故障的网络的性能，每隔三秒就会随机选择，停止并在三秒钟后重新启动Nfault个验证者。

表1中的结果表明，此崩溃故障情形下的性能下降了约50％，而较大的TimeoutPropose值有助于减轻延迟。 虽然平均延迟增加到大约两秒，但中位数接近一秒，并且延迟可能高达十甚至二十秒，虽然极端情况可高达七十秒。 而将TimeoutPropose修改为略微的不确定性可以缓解这种极端延迟的可能性。

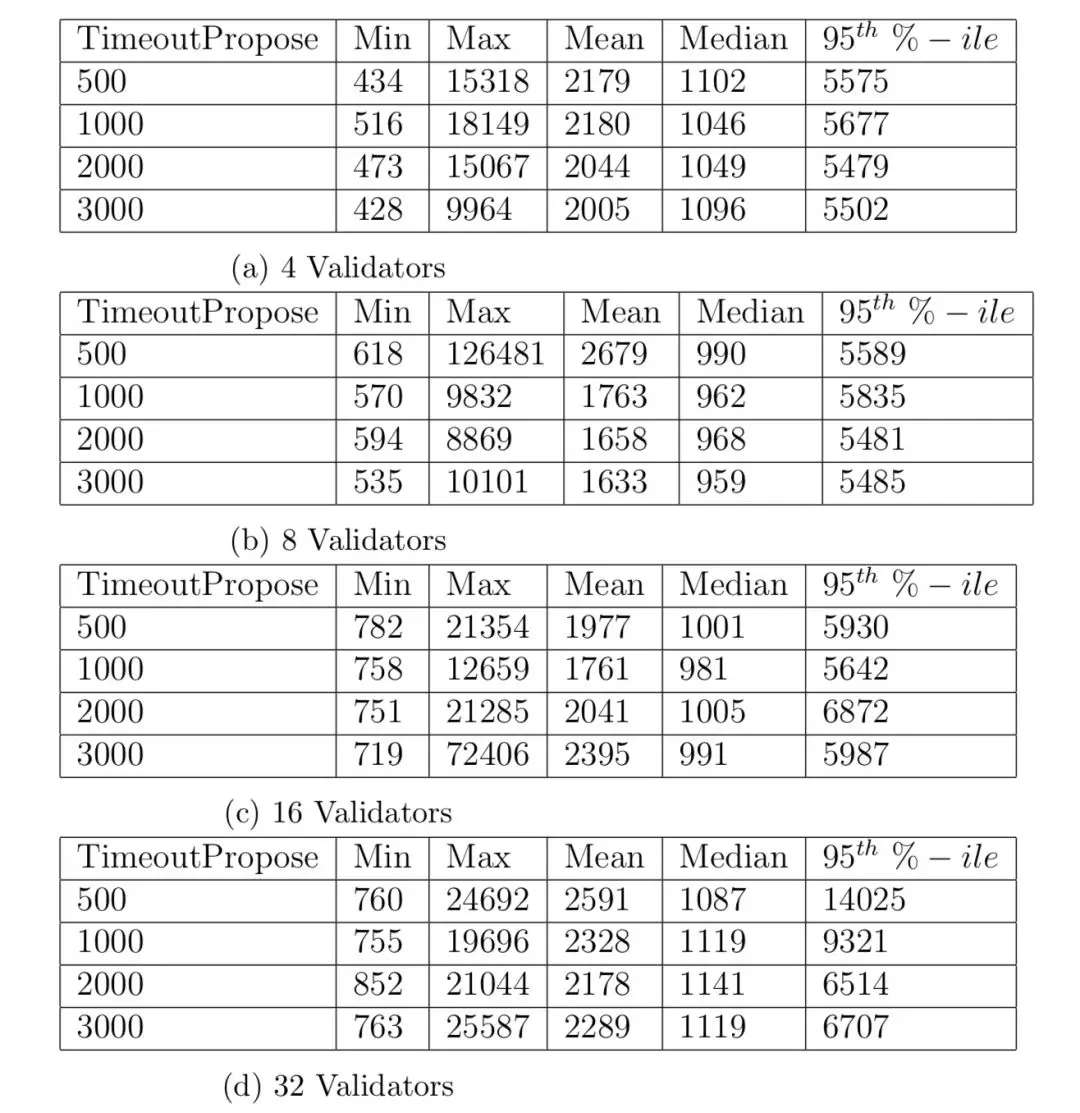


表1：崩溃 - 故障延迟统计信息。 每三秒钟，一次随机选择的Nfault个验证者崩溃，并在三秒钟后重新启动。 此崩溃重启过程持续了200个块。 对于TimeoutPropose参数的不同值，每个表都报告出块延迟的最小值，最大值，平均值，中位值和第95百分位数(95th percentile)。

**随机网络延迟**

另一种形式的故障，可能归因于拜占庭行为或网络异步，是为每次读取和写入网络连接注入随机延迟。在这个实验中，在每次网络连接的每次读写之前，验证者的Nfault都会休眠X毫秒，其中X统一绘制在（0,3000）上。 从表2中可以看出，延迟与崩溃失败情况类似，但增加TimeoutPropose会产生相反的效果。 由于并非所有验证人节点都有错误，因此TimeoutPropose的小值允许快速跳过错误的验证人节点。如果所有验证人节点都受到网络延迟的影响，则预期较大的Timeout-Propose值将减少延迟，因为不会有无错误的验证人节点跳过，并且将提供更多时间来接收延迟的消息。

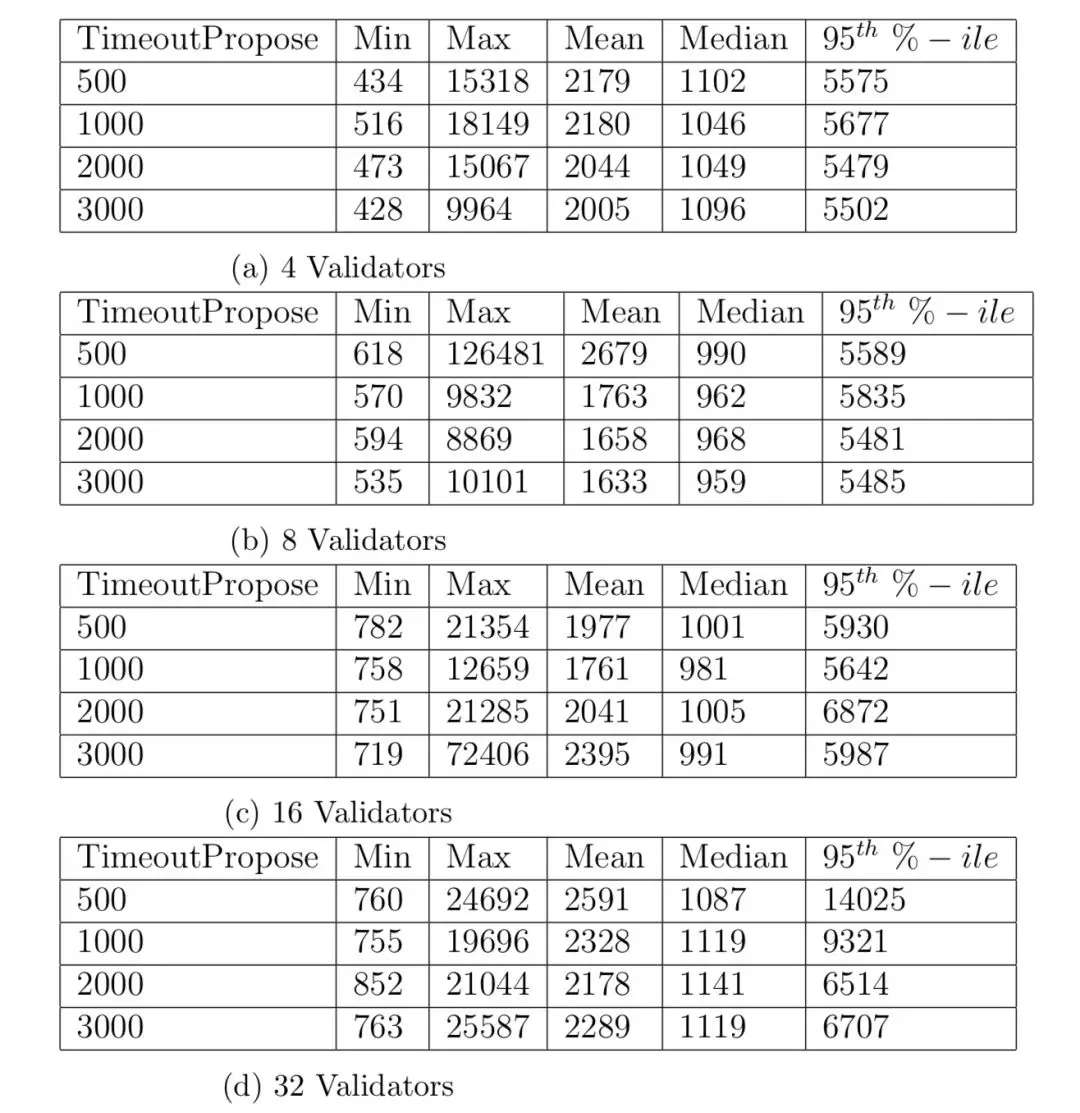


表2：随机延迟延迟统计。 Nfault个验证者设置为在每次读取和写入之前注入随机延迟，其中延迟时间在（0,3000）毫秒上均匀选择。

**拜占庭式的失败**

通过对状态机的以下修改，可以注入更明确的拜占庭故障：

* 相互矛盾的提案：在提出建议时，拜占庭验证员会分别签署两个相互冲突的提案并进行广播，同时进行投票前和预先提交，以分离其连接对等体的一半。
* 没有nil票：拜占庭验证者从不签署一次空的投票（nil-vote）。
* 签署每个提案：拜占庭验证者提交投票前和投票一看到它就会预先提交它看到的每个提案。

总之，这些行为明显违反了双重签名和锁定规则。 但请注意，这种行为主要是由于冲突提案的广播，以及最终提交其中一个提案。 更复杂的拜占庭策略则有待后续的工作。

虽说注入了拜占庭故障会导致许多系统完全立即失效，但Tendermint保持了可观的延迟，如表3所示。 由于这些故障与异步处理几乎没有关系，因此TimeoutPropose没有真正可辨别的效果。性能也随着更大的验证者集合而下降，这可能是处理拜占庭投票的一个想当然结果。

**相关工作**

本章节中的吞吐量实验基准测试了PBFT实现的性能和称为HoneyBadgerBFT的新随机BFT协议。 在他们的结果中，PBFT在四个节点上每秒实现超过15,000次交易，但随着节点数量的增加呈指数衰减，而HoneyBadgerBFT每秒可实现10,000到15,000次交易的比较均匀的性能。 然而，HoneyBadgerBFT中的出块延迟要高得多，对于大小为8,16和32的验证人集合，接近10秒，对于较大的验证人集合则更接近10秒。

用于研究共识实现的众所周知的工具是Jepsen （译者注：JEPSEN - Distributed Systems Safety Analysis. http://jepsen.io.），它用于通过模拟多种形式的网络分区来测试数据库的一致性保证。 使用Jepsen测试Tendermint仍然是未来工作中一个令人感到兴奋的领域。

**结论**

作者和Jae Kwon编写的Tendermint的实现很容易在全球分布的机器上实现每秒数千个交易，最多64个节点，延迟大多在一到两秒范围内。 这也使它与其他解决方案竞争激烈，尤其是区块链的当前状态，例如比特币，支持每秒约7笔交易。 此外，我们的软件实现体现出即使面对崩溃故障、消息延迟和故意的拜占庭故障等都是健壮的，在每个方案中也能够保持每秒超过一千个交易。

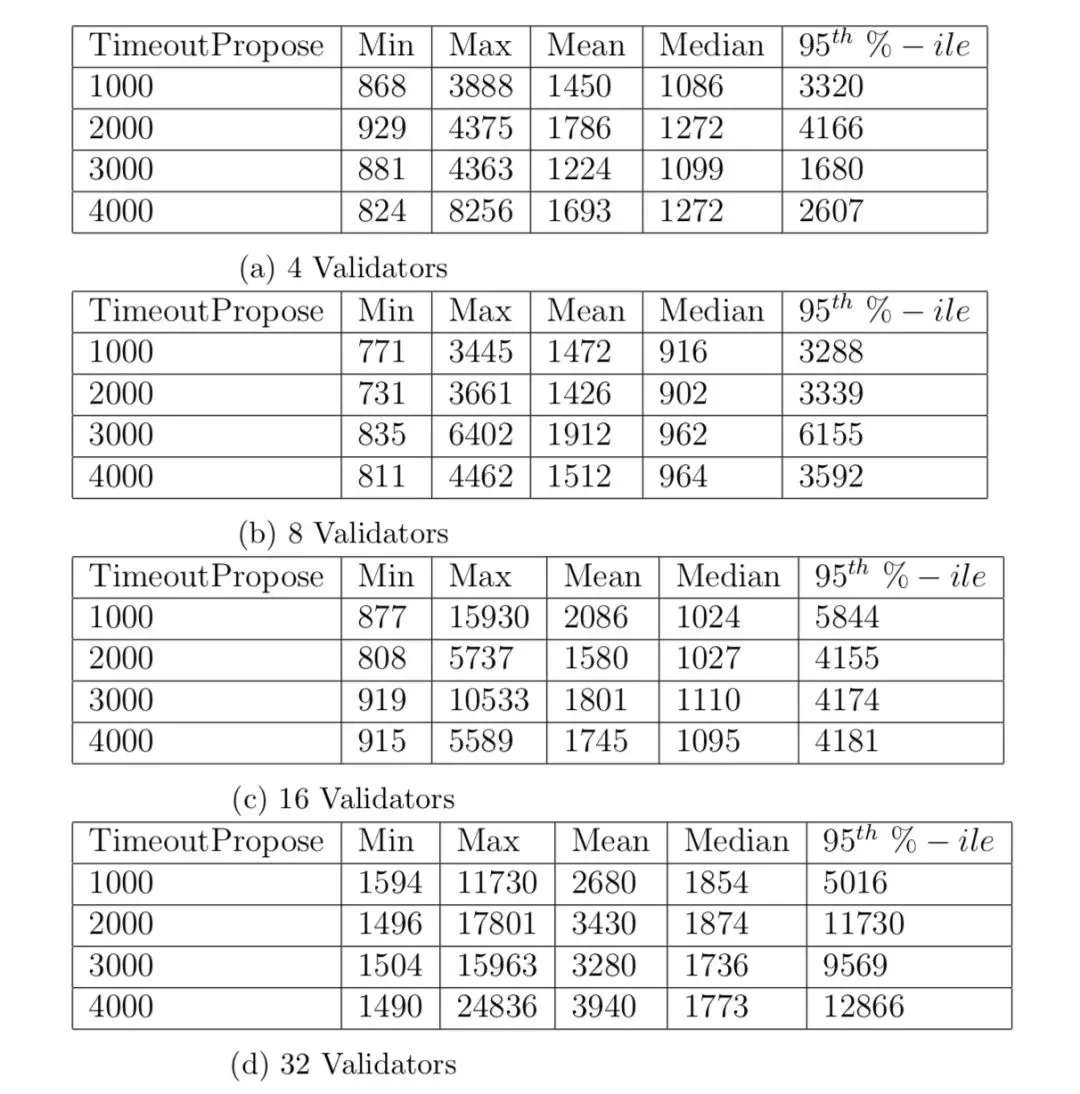


表3：拜占庭故障延迟统计。 拜占庭验证人提出冲突的阻止，并在他们看到任何提案后立即对其进行投票。 对于TimeoutPropose参数的不同值，每个表都报告出块延迟的最小值，最大值，平均值，中位值和第95百分位数。

至此，论文《Tendermint: Byzantine Fault Tolerance in the Age of Blockchains》中文译名《区块链时代的拜占庭容错：Tendermint》系列的核心内容一共八篇译文，全部推送完毕。