Abstract

Tendermint 是一种在对抗条件下分布式网络中的新事件排序协议，更常见地被认为是一致性算法或原子广播。由于广泛地成功解决了不需中央集权的公共设置问题在比如比特币、以太坊的虚拟货币领域，这个问题最近吸引了大量关注。Tendermint 将为了提供一个有问责保障制度的安全一致性协议以及在一致性算法之上建立任意应用的接口的经典学术工作现代化。Tendermint 是高性能的，每秒可以在全球数十个节点上以大约一秒的延迟完成数千个交易，并且在面对对抗攻击时，性能受到适度降低。

Chapter 1

介绍

今天，计算机工程的残酷事实是计算机是有缺点的，它们开始宕机，速度变慢，表现越来越差。更糟糕的是，我们倾向于关注把电脑连入网络（比如互联网），然而网络比计算机具有更高的不可预见性。这些挑战首要任务是关注“可容错的分布式计算（fault tolerant distributed computing）”，来找到有原则的协议设计，使得当被提供有用的服务时，在错误网络中连接的错误计算机保持同步。本质上，就是在不可靠当中生产出一个可靠的系统。

然而，在这个迅速增长的数字化和全球化世界，系统不能仅仅在面对不可靠部分时可靠，还得在面对恶毒环境或“拜占庭（Byzantine）”情况可靠。在过去十年间，大部分关键构造的组件已经被传送到有网络支持的系统中，就如同世界经济的大量组件。作为回应，则产生了网络战争、经济诈骗和一个彻底的金融和政治原则的扭曲。

1.1 比特币（Bitcoin）

在2009年，一个自称Satoshi Nakamoto的匿名软件开发者引介了一种解决这些问题的方法，就是一个同时进行在计算机科学、金融和政治领域的实验，一种叫做比特币的虚拟电子货币。比特币是第一个用来解决在面对恶性对抗情况下环境的容错分布式计算问题的协议。用区块链技术实现了一种数字货币，通过经济激励解决哈希碰撞的人，交易（transactions）达成共识。实质上，交易是以区块的形式被那些发现数据交易中的部分哈希碰撞的人排序。通过这种方式，正确顺序就是使得碰撞有最高累计难度的那个。这个解决办法被叫做Proof-of-Work (PoW)。

比特币的精妙之处在于它发明了一种货币，一种加密货币，而且发给那些解决哈希碰撞的人，交换他们做的解决部分哈希碰撞如此昂贵的事情。在精神上，可能假设解决这些问题的能力将会作为计算能力来分发，这样使得每个人都可以用一个CPU就能参与其中。不幸的是，比特币网络已经成长为这个星球上最大的超级计算机实体，超过所有其他的总和，仅评估一个单独的功能，通过几个数据中心分发运行主要由一小部分中国公司生产的专有集成电路（ASICs），就要大概一天消耗两百万美元的电力。长远地看，它的技术设计也有局限性：需要花费高达一个小时的时间来确定交易，这样就很难在顶层建立应用，而且无法以一种维持安全保证的方式来衡量。这还没有提及由于比特币社区的不成熟管理机制导

致的内部政治斗争。

放下这些问题，令人惊讶的是，比特币和它的技术持续地搅动密码学和分布式数据库以及合作经济，同样也持续吸引着几十亿的资本投资，包括新公司和新的加密货币，它们都以其独特的方式从比特币中分化出来。

1.2 Tendermint

在 2014 年，Jae Kwon 开始开发 Tendermint，一个通过使存在了数十年但是缺少社会背景，直到今天还需要广泛部署的问题的解决办法现代化的方法，寻找解决排序和执行一系列在对抗环境下的交易的一致性问题的办法。

在 2015 年年初，在一项由 Eris Industries 领导的项目中，把实际区块链解决方案应用在工业生产中，作者加入 Jae Kwon 开发 Tendermint 软件和协议。

这次合作的产物就是 Tendermint 平台，包含一份一致性协议，一个高性能的 Go 语言实现，一个灵活的可以在一致性基础上建立任意应用的接口，以及一套部署和管理工具。我们相信，对比之前的方法，Tendermint 完成了一份极好的设计和实现。以前的方法包括那些经典学术文献和比特币及其通过组合每个正确元素达到安全性、性能和简单性平衡的衍生物。

Tendermint 平台是开源的，可以在 https://github. com/tendermint/tendermint 获得，以及相关的存储库在 thttps:// github.com/tendermint。其核心是许可的 GPLv3，大部分的库都是 Apache 2.0。

1.3 贡献

这篇文章的主要贡献参见第三章和第九章。一些重要的意义有：

- 一份正式的 Tendermint 的 pi 演算说明书，一份非正式的关于其安全和义务的正确性证明（第三章）。
- 一个使得其核心一致性状态机更鲁棒，更具确定性和更加易于理解的重构。
- 软件在正常，错误及恶劣环境下大规模部署的性能和特征评估（第九章）。
- 无数的附加测试带来的无数的 bug 修复和性能提升。

第四到第八章讲述了一个完整系统中的很多其他的组件。其中一些，比如用来传播数据的子协议（第四章）和各种底层软件库（第八章），是 Jae Kwon 在作者加入之前设计并实现的。其余部分是在常规资讯作者和被作者启发设计与实现的。对于更多直接详尽的解释，请参见 Github。

尽管没在文章里阐述，作者还在这段时间做了一些以太坊项目的贡献，一个比特币的替代品，概括从货币到任意应用的技术的应用。另外，作者被邀请参加数不清的场合，做关于以太坊和 Tendermint 私下的或公开的演讲，包括以指导人和演讲者的身份。

最后一点，尽管第十章被放到了最后，但是如果在第三章之前读它，它会提供极其重要的内容并加固理解这篇文章。然而，为了不耽误读者认识 Tendermint，它被放在了最后。

Chapter 2

背景

分布式一致性系统已经成为现代互联网基础设施的关键组件，在某些水平上，为每一个互联网应用提供动力。本章介绍了理解和讨论这些系统的必要背景材料。另外，还介绍了 pi 演算，一个描述并发进程的正式语言，它将在第三章中被用来阐释 Tendermint 算法。

2.1 复制状态机（replicated state machine）

最常见的学习和实现分布式一致性的范例就是复制状态机。在其中，一个决定性状态机通过一系列进程被复制，为了使其以一个单状态机运行，不去管那些失败的进程。状态机由一组称为交易的输入驱动，每个交易根据其有效性可能会或可能不会导致状态转换并返回结果。更正式地说，一个交易就是一个数据库原子操作，意味着它要么完成，要么就没发生，不会出现中间状态。状态转移逻辑由状态机的交易函数控制，映射一个交易和当前状态到一个新的状态和返回结果。状态转移函数有时也会被指向为应用逻辑。

交易排序是一致性协议的职责，这样，产生的交易日志刚好被每个进程复制一遍。使用一个决定性的状态转移函数意味着每个进程将计算由相同交易日志提供的相同状态。

图 2.1 给出了复制状态机架构的总览。

Tendermint 从渴望创造一个通用的，高性能的，安全的，以及鲁棒的复制状态机中获得激发。

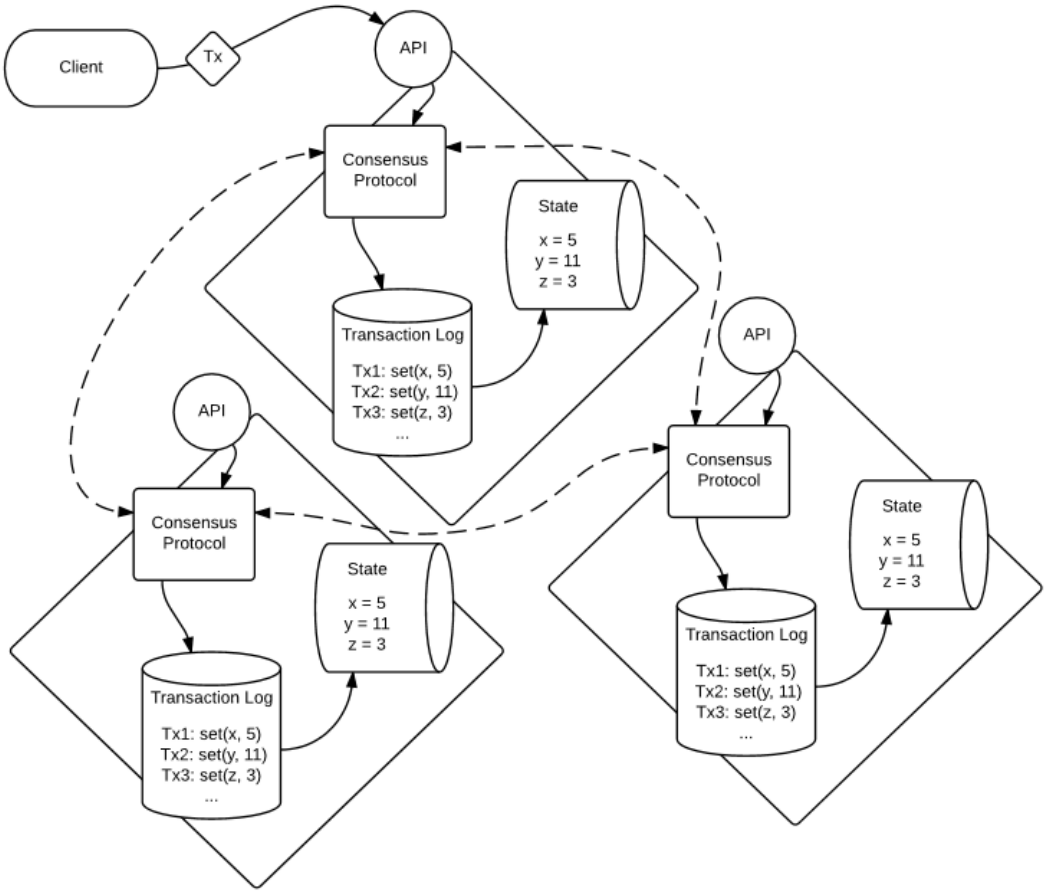


Figure 2.1: A replicated state machine replicates a transaction log and resulting state across multiple machines. Transactions are received from the client, run through the consensus protocol, ordered in the transaction log, and executed against the state. In the figure, each diamond represents a single machine, with dotted lines representing communication between machines to carry out the consensus protocol for ordering transactions.

2.2 异步性（asynchrony）

容错复制状态机的目的是协调一个计算机网络，当提供有用的服务时保持同步，尽管存在那些错误。

保持同步相当于成功地复制交易日志；提供有用的服务相当于保持状态机对于新交易可用。系统这两方面在传统上分别被认知为安全性（safety）和活性（liveness）。通俗地讲，安全性意味着没有坏事情发生；活性意味着好事情终将发生。违背安全性导致两个或更多有效的相互矛盾的交易日志。违背活性导致无响应的网络。

通过接受所有交易可以很容易满足活性。通过不接受任何交易可以很容易满足安全性。因此，状态机复制算法可以被看见地操作在一个由这些极端条件定义的范围内。典型地，进程需要在提交新交易之前接受从其他进程得到的信息的阈值。在同步环境中，我们做出网络消息延迟的最大值或者处理器速度的最大值的假设，那就足够容易去轮流提出新交易，赞成得到大多数投票的提出者，跳过没有在假设的限定内提议的提议者的回合。

在异步环境中，在没有网络延迟和处理器速度保障的情况下，交易变得特变难于管理。实际上，所谓的 FLP 不可能结果（FLP impossibility result）论证了在一个异步的分布式系统中，即是只是容忍一个进程的错误，也不存在一个一致性算法可以保证正确性。因为进程可以失败，证据相当于显示了存在协议的有效执行，在恰好的时机进程失败来阻止一致性。因此，我们不能保证一致性。

典型地，一个协议的同步性由使用超时反射来管理特定的转变。在异步环境，消息可能会反复延迟，依赖同步（超时）安全会导致在交易日志中的分歧。依赖同步来确保活性会引起一致性停止和服务无响应。前一种情况通常被认为更严重，因为调节冲突日志是令人头疼甚至无法完成的任务。

实际上，同步的方法只应用在消息延迟极其好控制的情况下，比如在同一架飞机上的控制员之间，或者利用同步原子钟的数据中心之间。因此，虽然存在许多有效的同步解决方案，但是计算机网络的一般不可靠性对于在没有显着的额外成本的实践中来说，风险太大。

根本上，有两个方法可以克服 FLP 不可能性结果。第一个是使用更有力的同步假设，即使弱一点的假设已经很充分了。比如说，只有在最终的时候，崩溃的进程会被怀疑会崩溃，正确的则不会。典型地，这种方法利用那些扮演特定协调角色的，还有那些如果在超时后被怀疑会出错被跳过的领导者们。实际上，这种领导者选举机制很难正确。

第二种克服 FLP 的方法是使用非决定论，包含随机因素，从人的直观上来说，可以克服 FLP，原因就是大概率上，是可以达成一致的。聪明地，依赖随机会显著降低速度，尽管某些高级的密码编写的技术最近几年在速度上取得了极大的提升。

2.3 广播和一致性

为了使一个进程可以在其他进程上复制自己的状态，它必须有权限进入允许它传播或者传送信息的基本通信基元（primitives）。一个最有用的这种基元之一就是可靠广播（reliable broadcast）。可靠广播（RBC）是一个基元，对于消息 m，满足：

- 有效性（validity）-如果一个正确的进程广播 m，它最终会传送 m
- 一致性（agreement）-如果一个正确的进程传送 m，所有正确的进程最终都会传送 m
- 完整性（integrity）-m 只会被传送一次，而且只在被自己的发送者广播的时候

本质上，RBC 使得一条消息最终被所有的正确进程传送一次。

另一种更有用的基元原子广播（atomic broadcast – ABC），满足所有 RBC 的属性和一条额外的属性：

- 总序性（total order）-如果正确进程 p 和 q 传送 m 和 m’，则如果 q 在传送 m’前传送 m，p 就在传送 m’前传送 m

当值在每个主机上以相同的顺序传送，原子广播也是一个可靠广播。注意这一点恰好是复制一条交易日志的问题所在。通俗地说，这个问题会被归诸于一致性（consensus），一致性基元的标准定义满足一下要求：

- 终止性（termination）-每个正确进程最终都会决定
- 完整性（integrity）-每个正确进程最多决定一次

• 一致性（agreement）-如果一个正确进程决定 v1，另外一个决定 v2，则 v1 = v2

• 有效性（validity）-如果一个正确进程决定 v，则至少是有一个进程提议了 v

直观地看，consensus 和 ABC 惊人的相似，最不同的地方就在于 ABC 是连续的协议，而 consensus 需要有终止。也就是说，被人们所知的，它们都可以简化成彼此。consensus 可以很容易地通过决定第一个值为原子广播来简化为 ABC。ABC 可以依次地运行一致性协议的多个实例来简化为 consensus，尽管某些微妙的事项一定要注意，尤其那些用来处理拜占庭错误的。围绕 ABC 简化为 consensus 的参数空间的一个完整描述依旧是一个开放的研究话题。

从历史来看，尽管大部分用例实际上需求 ABC，但是最广为被采纳的是一个用 Leslie Lamport 在 90 年代引入的，也被证实是正确的一致性算法，Paxos。Paxos 同时即允许又拒绝一致性科学（consensus science）的准则。一方面，它提供第一次真实世界（first real-world）、实际的、容错的一致性算法，另一方面，却又十分的难以理解和解释。该算法的每种实现都使用了自己的专门的技术包从 Paxos 建立 ABC，使得整个生态系统难以操纵、理解、利用。遗憾的是，尽管在针对不同困难去寻找解决方案付出了很多努力，但是想使其更容易被理解，几乎没有取得任何进展。

2013 年，Ongaro 和 Ousterhout 发表了 Raft，一个激发设计目的非常易懂的状态机复制算法。Raft 的设计不是从一致性算法出发并试图建立它所需的（ABC），它最先也是最重要考虑的是交易日志，并寻找可以组合在一起最终能提供 ABC 的正交组件，虽然它不是被描述成这样的。

Paxos 已经成为行业内主要的一致性算法，超过像 Amazon，Google 等公司，其他也已经建立起了具有高可用性的全球互联网服务。Paxos 一致性算法位于应用栈的最底层，提供一个统一的资源管理和分配的接口，操作时间尺度比用户面对的具有高可用性的应用更慢。

然而，自问世以来，Raft 已经被广泛采纳，尤其是在开源社区，几乎被每种主流语言所实现，并且在主要项目作为骨干使用，包括 CoreOs 分布式 Linux 发行版，以及开源时间序列数据库，InfluxDB。

Raft 从 Paxos 来的主要不同的设计决策是要首先关注于交易日志，而不是一个单独的值，特别是要让一个领导者坚持提交交易直到他传下去，也就是领导选举开始生效的时候。在某种程度上，这和区块链采取的方式很相似，虽热区块链的最主要优势在于它容忍不同错误的能力。

2.4 拜占庭容错 Byzantine Fault Tolerance

由于区块链通过在一个共享的数据库上分发责任来降低交易对手风险的方式，它们已经被认为是“可信任的机器（trust machines）”。特别是比特币，已经被注意到它承受其他参与者攻击和恶性行为的能力。习惯上，一致性协议对恶性行为的容忍被作为拜占庭容错（BFT）一致性协议。使用拜占庭一词是因为与拜占庭将军们面对的问题很相似，当将军之中有一个可能是叛徒的时候，他们尝试只用信使来协作攻打罗马。

在一个宕机故障中，一个进程很简单地终止掉。在拜占庭故障中，它可能反复出现。宕机故障比较容易处理，因为没有进程会对其它进程撒谎。只容忍宕机

故障的系统可以通过简单的多数原则来处理，因此通常容忍高至半个系统的同步失败。如果系统能容忍错误的数量是 f，这样的系统必须至少有 2f + 1 个进程。

拜占庭故障更复杂一些，在一个有 2f + 1 个进程的系统里，如果有 f 个进程是拜占庭，它们可以协作对其他 f + 1 个进程说任意的事情。举个例子，假设我们要试着在一个单独的比特的值上面达成一致，f = 1，所以我们一共有 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 值为 0，告诉 B 是 1。如果 A 同意是，B 同意是 1 的话，它们就都会认为自己得到多数答案，并且提交它们，至此，就违背了安全条件。因此， 拜占庭系统的容错边界要严格地比非拜占庭系统低。

实际上，可以看得出来拜占庭故障 f 的上限应该是非 f < N/3。因此，容忍一个单独的拜占庭进程，我们需要至少 N = 4 个进程。这样，故障进程就不会把它的票以不同的答案分给所有其他进程，如图 N = 3 那样。

在 1999 年，Castro 和 Liskov 发表了 Practical Byzantine Fault Tolerance，也称作 PBFT，它提出了第一个实际使用中的拜占庭容错优化算法。它为工业上处理以每秒上万条为度量的交易量的系统的拜占庭容错在实际应用上开创了先例。放下这些成就不说，拜占庭容错仍然被认为是昂贵的并且很大程度上是不必要的，而且最受欢迎的实现方法是很难建立在顶层上的。因此，尽管在学术兴趣上有所复兴，包括大量改进过的变种，在实现和部署上，并没有很大的进展。此外，PBFT 不会提供任何保证是否有三分之一或更多的网络协调者违反了安全性原则。

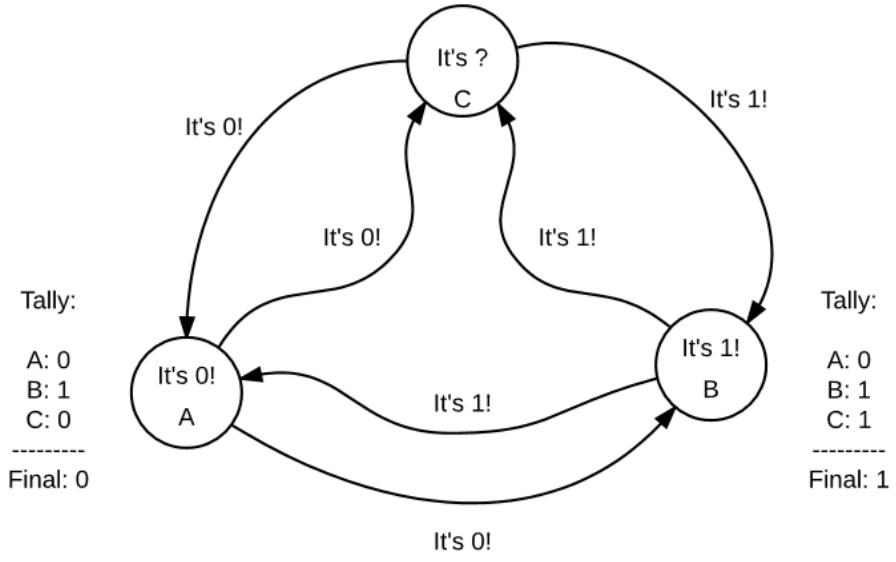


Figure 2.2: A Byzantine process, C, tells A one thing and B another, causing them to come to different conclusions about the network. Here, simple majority vote results in a violation of safety due to only a single Byzantine process.

2.5 密码学，信任，经济学

根本上，容错是一个源自信任（trust）缺失的问题—一个掌握其他进程如何表现的能力。正式地，信任可以由信息理论定义为一种降低某个人的世界模型的熵的方法—信任某人就是乐观地降低这个人对这个世界的不确定性，使其把更多的注意力放在更高的组织形式的秩序上。

加密原语（Cryptographic primitives）同样也是在根本上与信任问题相关联，而且也可以类似地被定义为一种允许大幅降低熵的机制—成功验证一个加密的函数会把一个分布式可能产生的结果瓦解成单个，在某些情况也可能会变成一小部分数量的结果。

众所周知，一个有着更高形式制度信任的文明，比如法律制度，有更高的生产力和更有生气的经济。这个结果给出直观的感觉就是可以更容易的协调，比如更多地信任一个互动可以降低需要被积极建模的可能输出结果的空间。遗憾的是，评估现代机构的可信度变得越来越难，因为在最近几十年来它们的复杂度飞速增长，也增加了它们所提供的确定性是一种假象的可能性。

幸运的是，密码学可以形成新的社会信任机构的基础，这可能会因为欺诈和/或不负责任的活动的风险降低，而在全球范围内大大提高人力协调能力。特别感兴趣的是加密原语在 BFT 算法中的重要性，既用于认证也用于播种非确定性。

最有趣的是，经济机制也可以作为减少熵的手段，只要经济主体能被激励 - 这就是说更有可能执行某种特定的行为。事实上，比特币的深刻见解是，加密原语可以与经济激励一起使用，以便能够有效地减少公共一致性网络的熵来实现状态的安全复制。

一个对信息理论的信任，密码学，一致性，经济，尤其是它们的内部关系的更正式探索，仍然是未来需要做的工作。

2.6 区块链

一个区块链的核心是一个以拜占庭容错原子广播为基础的专注完整性的方法。比如比特币区块链，使用经济学和加密随机化的组合来提供强烈的概率保证不会违反安全性，给出一个弱同步假设，区块据说比通过部分哈希碰撞彩票找到的数据块快得多。然而，实际上，众所周知，比特币的安全保障容易受到一些微妙的攻击。

区块链是从其在解决 ABC 中使用的两个关键优化获得的名称。第一个是它把交易分成区块（blocks）以便在许多交易中分摊延迟（大约十分钟）。第二个是将区块通过加密哈希链接到一个不可变的链，这样可以很容易验证历史记录。这两个优化都是对原始 BFT-ABC 的天然改进，前者改进了性能，后者改善了某些难以建模的拜占庭故障的容忍度。

在过去几年中，“区块链化”一致性算法已经变得很普遍，也就是使用哈希链接交易批次的区块链模型来适应 ABC。 据作者所知，Tendermint 是第一个这样的提案，从 80 年代后期升级了一个众所周知的 BFT 算法，尽管它已经演变成了一个自己的一致性算法。 随后，IBM 将 PBFT 升级成为一个区块链，然后摩根大通升级了一个 Raft 的 BFT 版本。

2.7 进程演算

那些部分系统要同时和其他系统一起执行的分布式系统一直因为它们难于设计，构建，调试而被人诟病。因为大部分需要被正式认证的技术和计算机科学的基础已经由顺序计算特定开发，所以它们很难被正式认证。

进程演算是被引入提供并发式计算正式基础的一系列模型。最流行的演算，通信顺序进程 Communicating Sequential Processes (CSP) 为许多现代编程语言提供了理论基础，比如 Go，在设计语言时就包含了并发原语。

在 80 年代，Robin Milner 推出了通信系统演算 Calculus of Communicating Systems (CCS)，被设计为顺序 lambda 演算的并发类似物，成为大部分函数式编程语言的基础。虽然 lambda 演算有功能应用程序作为其基本计算单位，但 CCS 使用共享通道上的两个并发进程之间的通信作为其基本操作原语。CCS 的更一般形式是 pi 演算，使得进程之间的通信图中的移动性能够使得通信信道本身可以沿其它信道传递，从而模糊数据，变量和信道之间的区别。这个结果是一个比前任更强大的连贯简约的计算模型。

Pi 演算已被证明是研究并发系统的高效工具，其应用从业务流程管理到细胞生物学。非常简单的注解简化了并发协议的描述。此外，众所周知的计算和逻辑之间的等价性使得可以将逻辑系统定义为与各种进程演算的补充，提供正式的方式来讨论和验证在适当的演算中指定的系统的属性。

我们对 pi 演算的描述只能指定 Tendermint 算法。

一个简单的 pi 演算以 Backus-Naur 形式的语法如下表示：

$$
\begin{array}{rrr}
P := & 0 & void \\
\mid & P \mid P & par \\
\mid & \alpha.P & guard \\
\mid & \alpha.P + \alpha.P & guarded\text{-}choice \\
\mid & (\nu x)P & fresh \\
 & & \\
\mid & F^s(y) & func \\
 & & \\
\alpha := & \tau & null \\
\mid & x!(y) & send \\
\mid & x?(y) & receive \\
\mid & susp_i & suspect
\end{array}
$$

每个语法规则都以其功能意义的参考为标记。一个进程可能是空的进程，0。它可能是两个进程的并行组合，P | P 表示同时运行的两个进程。一个被保护的进程，α.P，只允许进程 P 在动作 α 发生后执行。这个动作可以是空动作，τ，或者是 y 沿 x 的发送，x!(y)，或接收，x?(y)。保护的选择将非确定性注入到演算的运算中，使得进程 α.P + β.Q 将不确定地执行 α 或 β，然后分别运行 P 或 Q。

可以通过(vx)P 创建一个新的通道 x，这样 x 只能在 P 中访问。函数形式 $F^s(y)$允许

我们将变量 s 和 y 传递到一个可能导致自身执行递归的 F 进程中。通常，我们让 s 是像状态一样的变量，而 y 是演算中的通道。最后，由于我们感兴趣的是异步网络的一致性，我们采用被认为是不可靠的故障检测器的超时的抽象，并将其建模为非确定性动作。当进程 i 被怀疑失败（即在一些超时后），$susp_i$ 动作就被触发了。

注意一点，我们可以使用 ΣP 表示超过两个进程的守卫选择，ΠP 表示两个以上进程的并行组合。我们也承认发送和接收的多种形式，比如，x?(v, w) | x!(y,z) 等同于 x?(d).d?(v).d?(w) | (vc)x!(c).c!(y).c!(z)。

操作语义定义了进程可能执行的实际不可逆的计算步骤。有效的是，唯一相关的操作是通信，称为 comm 规则：

$$(x?(y).P \mid x!(z)) \rightarrow P\{z/y\} \quad (2.1)$$

符号 P{z/y}表示 P 中所有出现的 y 都被替换为 z。换句话说，z 被发送到 x 上，以 y 的形式接收，并被馈送到 P。

给定一个 pi 演算过程，我们可以通过应用 comm 规则来执行它。举个例子，

$$(x?(y).y!(x) \mid x!(z)) \rightarrow z!(x) \quad (2.2)$$

现在，我们可以使用一个形式的逻辑来表达进程可能满足的属性。例如，模态的 Hennessy - Milner 逻辑可以表示一个过程将在一些或所有形式的行动发生之后满足一些其他逻辑表达。通过将更复杂的运算符添加到逻辑中，可以构建形式化的系统，其可以容易地描述分布式系统的重要属性，例如安全性、活性和本地化。用 pi 演算写的系统之后可以被正式验证来用模型检查软件满足相关属性。

虽然我们使用 pi 演算来指定 Tendermint 算法，但是我们仍将使用相关的形式逻辑，以及相应的属性验证，以供将来使用。

2.8 Tendermint 的需求

比特币及其衍生物尤其是以太坊的成功，还有它们对任意代码的安全、自主、分布式、容错执行的承诺，使得这个星球上几乎所有的主要金融机构都对区块链现象产生了兴趣。特别是对两种形式的技术有了一种了解：一方面是公共区块链，被亲切地称为大坏的公共区块链或 BBPBs，后者的协议以原生货币为基础的经济激励为主。另一方面是所谓的私有区块链，或者可能应该更准确地被称为“财团区块链”，通过使用哈希树，数字签名，P2P 网络和增强功能有效地改进了传统的一致性算法和 BFT 算法。

随着我们社会的基础设施越来越分散，随着业务的性质变得更加组织化，越来越需要一个透明，负责任，高性能的 BFT 系统，可以支持从融资到域名注册到电子投票的应用， 并配备先进的治理机制和未来的演变机制。Tendermint 是针对财团或组织间逻辑进行优化的解决方案，但足够灵活，可以容纳任何从私营企业到全球货币的解决方案，以及足以与当今主要的非 BFT 一致性算法解决方案例如 etcd，consul 以及 zookeeper 进行竞争的高性能，同时为应用程序开发人员提供更大的弹性，安全保证和灵活性。

Chapter 3

Tendermint 一致性算法

这章讲解了 Tendermint 一致性算法和一个相关的原子广播区块链。详细描述了 BFT 一致性问题，在 pi 演算中给出了 Tendermint 一致性算法的正式规范。Tendermint 区块链被非正式地证明可以满足原子广播。我们将其留在将来的工作中，以便在进程演算中捕获完整的区块链协议并验证其属性。

3.1 Tendermint 总览

Tendermint 是区块链模型中的安全状态机复制算法。它提供了一种 BFT-ABC 的形式，使得更加负责任—如果安全性被违反，总是可以验证出是谁的恶意行为。

Tendermint 从一组验证者开始，通过其公钥识别，每个验证者负责维护复制状态的完整副本，并提出新的区块（交易批次），并对其进行表决。每个区块被分配一个增量索引或高度，使得有效的区块链在每个高度只有一个有效的区块。在每个高度，验证者轮流提出新的区块，这样在任何轮的时候最多只有一个有效

的提议者。由于网络的不同步，可能会花费多轮来提交给定高度的区块，如果三分之一或更多的验证者离线或分区，则网络可能完全停止。验证者在区块提交之前进行两个阶段的投票，并遵循一个简单的锁机制，防止不到三分之一的验证者的任何共同恶意损害安全性的行为。

要注意的是，轮流投票机制的核心是一致性算法，它是捆绑一起组成区块来产生原子广播。每个块都包含一些元数据，称为它的头，其中包含前一个高度的区块的散列，产生一个哈希链。头里还包括区块高度，区块被提出的当地时间，以及包含在区块中的交易的 Merkle 跟散列。

3.2 一致性算法

一致性算法可以大致分为以下几种几何正交组件：

• 提议（proposals）：每个轮次必须由正确的提议者提出一个新的区块，并向另一个验证者进行传播。 如果没有在足够的时间内收到提案，则应跳过该提案者。

•投票（votes）：必须进行两个阶段的投票，以确保拜占庭的最佳容错能力。他们被称为预投票（pre-vote）和预提交（pre-commit）。 在同一轮中，来自同一区块的三分之二以上验证者的一组预提交是一次提交。

• 锁（locks）：Tendermint 确保没有两个验证者在相同的高度处提交不同的区块，假设不到三分之一的验证者是恶意的。这是通过一种锁机制实现的，该机制决定验证者如何根据在同一高度上先前的预投票和预提交来预投票和预提交。要注意的是，这种锁机制必须仔细设计使其不损害活性。

为了给单独的拜占庭故障提供容忍，一个 Tendermint 网络必须包含最少四个验证者。每个验证者必须具有用于产生数字签名的非对称加密密钥对。验证者从共同的初始状态开始，其中包含验证器的有序列表 L。每个验证者都通过其公钥进行识别，所有提案和投票必须由相应的私钥签名。这可以确保任何观察者总是能够对提案和投票进行验证。假设多达三分之一的验证者是恶意的，以任意方式合作颠覆系统安全或活性的，是非常有帮助的。

一致性算法始于第零轮，第一个提议者就是 L 中第一个验证者。一轮的结果是做出提交，或决定转入下一轮。新一轮是下一个提议者。使用多轮可以给验证者多次机会在网络异步或验证失败的情况中达成一致。与需要一种领导选举形式的算法相反，Tendermint 在每一轮都有一个新的领导者（提议者）。验证者与投票接受该提议相同的方式跳到下一轮，提供给协议一个统一的，在明确的领导选举程序算法中缺席的机制。

每一轮的开始对同步性的依赖性很弱，因为它利用本地时钟来决定何时跳过提议者。 也就是说，如果验证者没有在本地估量的进入新一轮的“超时提议（ProposalTimeout）”中收到提议，则可以投票以跳过提议者。在这种机制中固有的是一个弱同步的假设，即该提议最终将在“超时提议（ProposalTimeout）”中传达，该延迟本身可能会随着每一轮的进行而增加。这个假设将在第十章中有更全面的讨论。

在提案之后，回合以完全异步的方式进行 - 验证者只有在至少三分之二的验证者听取之后才能继续进行。这减轻了对同步时钟或有界网络延迟的任何种类的依赖，但是这意味着如果有三分之一或更多的验证者无响应，网络就将停止。异步投票紧接着这个弱同步提议的流程，如图 3.1 所示。

为了安全地跳过回合，引入了一些锁的规则，迫使验证者证明他们的投票是合理的。 虽然我们并不一定要求他们实时广播其理由，但我们希望他们保留数据，以便在安全受到足够的拜占庭故障的影响的情况下将其作为证据。这种问责机制使得 Tendermint 能够在面临这种失败的情况下提供比例如，如果第三个或更多的验证者是拜占庭的话就不会提供保证的 PBFT，更有力的保证。

验证者使用不同的消息集进行通信，以管理区块链，应用状态，对等网络和一致性。 然而，一致性算法的核心只包含两条消息：

•提议消息（ProposalMsg）：由提议者签字的给定高度和回合的区块的提议。

• 投票消息（VoteMsg）：一个签过字的提议的投票。

实际上，我们使用额外的消息来优化区块数据和投票的传播，这会在第四章进行讨论。

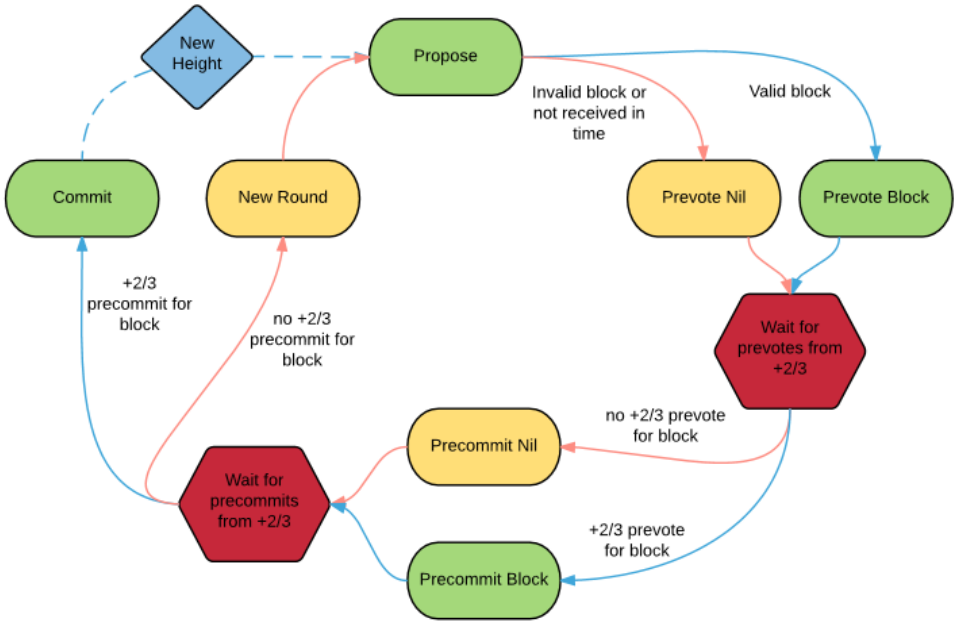


Figure 3.1: After the proposal step, validators only make progress after hearing from two-thirds or more (+2/3) of other validators. The dotted arrow extends the consensus into atomic broadcast by moving to the next height.

3.2.1 提议（Proposals）

每一轮都以提案开始。给定轮次的提议者从本地缓存中（Mempool，参见第四章）拿到一批最近收到的交易，组成一个区块，并广播包含该块的签名的 ProposalMsg。 如果提议者是拜占庭，则可能会向不同的验证者广播不同的提议。

提议者通过一个简单的，确定性的轮次排序，所以只有一个提议者对于给定的回合是有效的，每个验证者都知道谁是正确的提议者。如果收到低轮次的提议或错误的提议者的提议，它将被拒绝。

对于拜占庭容忍来说，循环提出建议是必要的。比如在 Raft 中，如果当选的领导者是拜占庭，并且与其他节点保持强大的网络连接，则它可以完全破坏系统，破坏所有的安全性和活性的保证。Tendermint 通过投票和锁机制保证安全性，并通过循环提议保证活性，所以如果一个提议者不处理任何交易，其他的也可以继

续。也许更有趣的是，验证者可以通过治理模块（参见第 6 章）投票去除或替换拜占庭验证者。

3.2.2 投票（votes）

一旦验证者收到一份完整的提议，它就会为该提议进行预投票并将其广播到网络。 如果验证者在 ProposalTimeout 中没有收到正确的提案，则会投 nil 票。在与拜占庭验证者的异步环境中，单个投票阶段每个验证者只投一票，是不足以确保安全性的。实质上，由于验证者可以说谎，而且对消息传送时间没有保障，一个流氓验证者可以协同一些别的验证者提交一个值，而其他验证者并没有看到这次提交就进入下一回合，在这个回合它们会提交不同的值。

单一的投票阶段允许验证者告诉对方他们对提案的了解。但是，为了容忍拜占庭故障（其实质上是谎言，欺诈，欺骗等），他们还必须告诉对方他们知道哪些其他验证者已经宣称对提议的了解。也就是说，第二阶段确保足够的验证者见证了第一阶段的结果。（*这个就是为什么需要三步提交的原因。per-vote 阶段：请求投票，并收集起他人的投票。commit 阶段，每个人收集到其他三分之二的人的投票，就认为自己已经见证了这次投票，发送 pre-commit 消息，然后，收集到足够的 per-commit 的消息，就认为 commit 成功了。也就是一次是发起投票，一次是足够的人见证了这次投票。）

因此，区块的预投票也是网络准备提交区块的投票。为 nil 的预投票是网络准备进入下一轮的投票。在一个在线提议者的理想回合中，超过三分之二的验证者将会为提议预投票。在给定回合的单独区块的一组超过三分之二的预投票被称作 polka。单独区块的一组超过三分之二的为 nil 的预投票被称作了 nil-polka。

当一个验证者接收到一个 polka（也就是一个单独区块的超过三分之二的预投票）时，它已经收到一个网络已经准备好提交区块的信号，它作为验证者对该区块进行签名和广播预提交投票的理由。有时，由于网络异步，验证者可能不会接收到 polka，或者可能就没有。在这种情况下，验证者在为该区块签字预提交是不合理的，因此必须为 nil 签署并发布预提交的投票。也就是说，在没有 polka 的理由的情况下签署预提交是一种恶意行为。

一次预提交实际上是提交一个区块的投票。一次对 nil 的预提交实际上是进入到下一轮的投票。如果一个验证者接收到一个区块的超过三分之二的预提交，它将提交该块，计算结果状态，并在下一个高度移动到回合 0。如果验证者收到超过三分之二的 nil 预提交，则转到下一轮。

3.2.3 锁（locks）

确保轮次的安全性可能是棘手的，因为必须避免将为同一高度的两个不同轮次的两个不同的区块提供合理性这种情况。在 Tendermint 中，这个问题是通过围绕 polka（即相同的区块的三分之二以上的预投票）的锁机制来解决的。实质上，一次预提交必须由 polka 证明，而且一个验证者会被认为在上一个它提交的区块上锁住。锁有两条规则：

• Prevote-the-Lock：验证者必须对他们被锁定的块进行预投票，如果它们是提议者，则提出它。这防止验证者在一轮中预先提交一个区块，然后在下一轮中为一个不同的区块提供 polka，从而危及安全性。

• Unlock-on-Polka：一个验证者只可以在一轮看到的 polka 比它锁定的那轮的 polka 大之后才可以释放锁。如果验证者预提交那些网络其余部分不想提交的东

西，这允许它们释放锁，从而保护活性，但是不损害安全性。如果有一个在验证者锁定那轮之后的一轮的 polka，只允许解锁。

为了简单起见，验证者被认为在每个高度的回合-1 处都被锁定在 nil，所以 Unlock-on-Polka 意味着验证者直到看到 polka 才能在新的高度预提交。

通过实例可以更直观地理解这些规则。考虑四个验证者 A，B，C，D，并且假设在第 R 轮有一个对区块 X 的提议。假设有一个区块 X 的 polka，但是 A 没有看到它，并且预提交为 nil，而其他的则预提交区块 X。现在假设唯一一个看到所有预提交的是 D，而其它则说没有看到 D 的预提交（它们只看到它们的两个预提交，和 A 的 nil 预提交）。D 现在将提交区块，而其它的则进入 R + 1 回合。因为任何验证者都可能是新的提议者，如果他们可以提出并投票支持任何新的区块，比如区块 Y，那么他们可能提交它并连累安全性，因为 D 已经提交了区块 X。请注意，这里甚至没有拜占庭行为，只是因为异步性！

锁机制通过强制验证者坚持使用它们预提交的区块来解决问题，因为其他验证者可能已经基于它们所预提交的提交了（如本例中 D 所示）。实质上，在一个回合中一旦超过三分之二预提交一个区块，网络就被锁定在该区块上，也就是说，在更高的轮次上一定不可能为一个不同的区块生成一个有效的 polka。这是 Prevote-lock 的直接动机。

然而，Prevote-the-Lock 是不够的。必须要有一种方式来解锁，以免我们牺牲活性。考虑这样的一个回合，其中 A 和 B 预提交区块 X，而 C 和 D 预提交 nil - 分立投票。他们都转到下一轮，并且提出了 C 和 D 所预投票的区块 Y。假设 A 是拜占庭，并且为 Y 块预投票（尽管被锁在区块 X 上），从而得到一个 polka。假设当 A 离线，C 和 D 预提交区块 Y 时，B 没有看到 polka 并且预提交 nil。他们移动到下一轮，但 B 仍然锁定在区块 X 上，而 C 和 D 现在被锁定在区块 Y 上，并且由于 A 离线，它们永远不会得到 polka。因此，我们以不到三分之一的拜占庭验证者（这里只有一个）连累了活性。

解锁的明显理由是 polka。一旦 B 看到区块 Y 的 polka（C 和 D 之前为区块 Y 验证的它们的预提交），它就应该可以解锁，因此提交区块 Y。这是 Unlock-on-Polka 的动机，如果他们在一轮中看到一个比他们锁定的那轮更大的 polka，它则允许验证者解锁（并预提交一个新的区块）。

3.2.4 正式规范

既然我们已经详细解释了这个协议，我们现在提出 pi 演算的正式规范。

使得 $\text{Consensus} \coloneqq \prod_{i=1}^{N} Y_i$ 表示一组 N 个验证者的一致性协议，每个都互相执行互不相容的一组进程中的一个，Y_i。内部状态 S = {r, p, v}由一个严格增加的回合，r，一个提议，p，组成，包含这轮提议的区块；和一组包含所有轮的所有的投票组成。我们在第 r 轮分别用v_r^1和v_r^2表示一组预投票和预提交，我们让 vote :: v 表示集合{vote}和 v（即，添加到 v 的 vote）的并集。我们定义 proposer(r) = r mod N 作为第 r 轮提议者的索引。 我们在协议的特定点表示一个对等体为$Y_i^{r,p,v}$。在 PR_i，PV_i，PC_i 范围内的进程 Y_i，分别表示提议（proposal），预投票（prevote），预提交（precommit）的缩写。我们引入 PV 和 PC 的附加子功能来捕获 PV1，PV2 等的递归。

对等体使用广播信道连接每个消息类型，即，$propose_i$，$prevote_i$ 和 $precommit_i$，以及用于决定或提交值 d_i 的渠道。通过大量用这些符号，可以通过 xxx_i 上的每个进程接收广播频道 xxx_i 上的单个发送。

我们只使用两种信息类型：提议和投票。每个都包含一个回合号，区块（哈希）和签名，表示为 msg.round，msg.block，msg.sig。注意，我们可以将签名吸收到广播频道本身，但是我们需要使用它作为拜占庭行为的证据。

该规范分为两部分，即图 3.2 和 3.3。

$$Consensus := \prod_{i=1}^{N} PR_i^{0,\emptyset,\emptyset},$$

$$\begin{aligned}
PR_i^{r,p,v} :=& \text{if } i = proposer(r) \text{ then} \\
& \quad propose_i!(prop) \mid PV_i^{r,prop,v}, \text{ where } prop = chooseProposal(p) \\
& \text{else if } p \neq \emptyset \text{ then} \\
& \quad PV_i^{r,p,v} \\
& \text{else} \\
& \quad propose_{proposer(r)}?(prop).PV_i^{r,prop,v} + susp_{proposer(r)}.PV_i^{r,\emptyset,v}
\end{aligned}$$

$$PV_i^{r,p,v} := prevote_i!(p) \mid (\nu\, c)(\prod_{j=1}^{n} prevote_j?(w).c!(prevote_j, w) \mid PV1_i^{r,p,v}(c))$$

$$\begin{aligned}
PV1_i^{r,p,v}(c) := & \text{ if } max_b(|\{w \in v_r^1 : w.block = b\}|) > \frac{2}{3}N \text{ then} \\
& \quad PC_i^{r,b,v} \\
& \text{else if } |v_r^1| > \frac{2}{3}N \text{ then} \\
& \quad PC_i^{r,\emptyset,v} \\
& \text{else} \\
& \quad c?(pv, vote).\ \text{if } vote.round < r \text{ then} \\
& \quad\quad pv?(w).c!(pv, w) \mid PV1_i^{r,p,v}(c) \\
& \quad \text{else if } vote.round = r \text{ then} \\
& \quad\quad PV1_i^{r,p,vote::v}(c) \\
& \quad \text{else} \\
& \quad\quad PR_i^{vote.round,p,vote::v}
\end{aligned}$$

Figure 3.2: Formal specification of Tendermint consensus in the π-calculus, part I. $chooseProposal(p)$ must return p if it is not $\emptyset$, and otherwise should gather transactions from the mempool as described in Chapter 4. After receiving a proposal or timing out, validators move onto prevote, where they broadcast their prevote and wait to receive prevotes from the others. If a vote is received for a later round, we skip ahead to that round.

$$PC_i^{r,p,v} := precommit_i!(p) \mid (\nu\, c)(\prod_{j=1}^{n} precommit_j?(w).c!(precomit_j, w) \mid PC1_i^{r,p,v}(c))$$

$$\begin{aligned}
PC1_i^{r,p,v}(c) := \text{if } max_b(|\{w \in v_r^2 : w.block = b\}|) > \frac{2}{3}N \text{ then} \\
&\quad d_i!(b) \\
&\text{else if } |v_r^2| > \frac{2}{3}N \text{ then} \\
&\quad PR_i^{r+1,\emptyset,v} \\
&\text{else} \\
&\quad c?(pc, vote).\ \text{if } vote.round < r \text{ then} \\
&\qquad pc?(w).c!(pc, w) \mid PC1_i^{r,p,v}(c) \\
&\quad \text{else if } vote.round = r \text{ then} \\
&\qquad PC1_i^{r,p,vote::v}(c) \\
&\quad \text{else} \\
&\qquad PR_i^{vote.round,p,vote::v}
\end{aligned}$$

Figure 3.3: Formal specification of Tendermint consensus in the π-calculus, part II. Validators broadcast their pre-commit and wait to receive pre-commits from the others. If a vote is received for a later round, we skip ahead to that round. When more than two-thirds pre-commit for block b, we fire b on channel d_i, signalling the commit, and terminating the protocol.

3.3 区块链（Blockchain）

Tendermint 一次性操作交易的批次或区块。通过将每个区块通过其加密哈希明确地链接到它之前的一个区块，从而将一个区块维持到下一个区块，从而形成区块链。区块链包含有序交易日志和验证者提交区块的证据。

3.3.1 为什么用区块？

一致性算法通常通过设计一次提交交易，并在其发生之后实现批处理。正如第二章中提到过的，从批量原子广播的角度解决问题，主要得到两个优化方式，这给我们提供了更多的吞吐量和容错能力：

• 带宽优化（bandwidth optimization）：由于每次提交都需要所有验证者之间进行两轮通信，所以区块中的批次交易会分摊区块中的所有交易的提交成本。

• 整体性优化（integrity optimization）：区块的哈希链形成一个不可变的数据结构，很像 Git 仓库，可以在历史记录中的任何一个位置对子状态进行确定性检查。

区块也产生另一个效果，这是更微妙但可能更重要。它们将交易的最小延迟增加到整个区块的最小等待时间，对于 Tendermint 来说约为数百毫秒到几秒。而传统的可序列化数据库系统提供了大概几毫秒到几十毫秒的提交延迟。他们能够做到这一点，是因为它们不是拜占庭容错的，只需要一轮通信（而不是两次），并且需要超过一半的副本（而不是三分之二）的响应。然而，与其他算法中的领导选举中断的快速提交时间不同，Tendermint 提供了一种更为规则的脉冲，可以在节点故障和异步方面更好地响应网络的整体运行状况。

尽管有目的地引发的延迟在金融市场上显示出希望，这类脉冲在互联网通信自治系统的一致性中所起的作用还有待确定。

3.3.2 区块结构

区块的目的是包含一批交易并链接上到之前的区块。链接以这两种形式呈现：前一个区块的哈希，和导致前一个区块提交的一组预提交，或者也可以称之为 LastCommit。因此，块由三个部分组成：区块头，交易列表和 LastCommit。

3.4 安全性（safety）

在这里，我们草拟了一个简短的证明，即 Tendermint 满足原子广播，满足：

- 有效性（validity）-如果一个正确的进程广播 m，它最终会传送 m
- 一致性（agreement）-如果一个正确的进程传送 m，所有正确的进程最终都会传送 m
- 整体性（integrity）-m 只被传送一次，而且只当它被自己的发送者广播的时候
- 总序性（total order）-如果正确的进程 p 和 q 传送 m 和 m′，则如果 q 在传送 m′之前传送 m，p 就在传送 m′之前传送 m

请注意，如果我们将 m 作为一个块，Tendermint 不能满足有效性，因为不能保证提议的区块最终被提交，因为验证者可能移动到新一轮并提交不同的区块。如果我们将 m 作为区块中的一批交易，那么我们可以通过让验证者不断重新提出相同的批次直到它被提交，来满足有效性。然而，为了满足前一半的完整性，我们必须引入一个额外的规则，禁止一个正确的验证者提出一个区，或者提交一个包含一批已经提交过的交易的区块。幸运的是，批次可以由它们的根目录来索引，并且在提案和预提交之前进行查找。

或者，如果我们将消息 m 作为一个交易，那么我们可以通过在 mempool 上断言一个持久性（persistence）属性来满足有效性，即一个交易在 mempool 中一直存在，直到它被提交。然而，为了满足前一半的完整性，我们必须依靠应用状态来对交易执行一系列规则，使得给定的交易只有效一次。举个例子，这可以通过在帐户上使用序列号来完成，就像在以太坊中所做的一样，或通过保留一个未使用的资源的列表，每个资源只能使用一次，就像比特币中使用的一样。由于有多种方法，Tendermint 本身不能确保一个消息只被传递一个消息一次，而是允许应用开发人员指定。请注意，完整性的后半部分是很好满足的，因为只有正确提议者提出的区块中的交易才能被提交。

为了表明 Tendermint 满足其余的属性，我们引入了一个新的属性，状态机安全性，并表明满足状态机安全性的协议也满足一致性和总序性。状态机安全说明，如果一个正确的验证者在某个高度 H 处提交一个区块，则不会有其他的正确的验证者将在 H 处提交不同的区块。假设所有消息最终都会被接收到，这就马上可以证明一致性，因为如果一个正确的验证者在包含交易 m 的高度 H 处提交区块 B，所有其他正确的验证者将不能提交任何其他区块，因此最终必须提交 B，从而传送 m。

现在，需要做的就是证明状态机安全性满足总序性，Tendermint 满足状态机安全性。先看前者，考虑由验证者 p 和 q 传递的两个消息 m 和 m′。状态机安全性确保当且仅当 q 在高度 H_m 传送 m 时，p 在高度在高度 H_m 传送 m。并且当且仅当 q 在高度 $H_{m'}$ 传送 m′时，p 在高度在高度 $H_{m'}$ 传送 m′。不失一般性，由于高

度严格上升，所以 $H_m < H_{m'}$。这样，我们就得到了当且仅当 q 在 m'之前传送 m 时，p 在 m'之前传送 m，这也正好是总序性的声明。

最后，为了证明当少于三分之一的验证者是拜占庭的时候，Tendermint 满足状态机安全性，我们以反证的方式进行。假设 Tendermint 不满足状态机安全性，允许在同一高度提交多个区块。那么我们表明，这种情况至少有三分之一的验证者必须是拜占庭才能发生，与我们的假设相矛盾。

考虑一个正确的验证者已经在高度 H 和回合 R 提交区块 B。提交一个区块意味着验证者在回合 R 从超过三分之二的验证者的情况下见证了区块 B 的预提交。假设另一个区块 C 在高度 H 处被提交。我们有两个选择：它在回合 R 被提交，或在回合 S > R 被提交。

如果是在第 R 回合提交的，那么超过三分之二的验证者必须在第 R 回合中预提交，这就意味着至少有三分之一的验证者在第 R 回合中即预提交了区块 B 也预提交了区块 C，显然这是拜占庭，假设区块 C 在回合 S > R 中被提交。由于超过三分之二的验证者预提交了 B，他们被锁定在第 S 轮锁定了 B，因此必须为 B 预投票。为了预提交区块 C，他们必须为 C 证明一个 polka，这需要超过 三分之二的验证者为 C 预投票。然而，由于超过三分之二验证者被锁定在区块 B 并需要对 B 进行预投票，C 的 polka 将要求至少三分之一的验证者违反 Prevote-the-Lock，这很显然是拜占庭。因此，为了违反状态机安全性，至少要有三分之一的验证者必须是拜占庭。因此，当不到三分之一的验证者是拜占庭时，Tendermint 满足状态机安全性。

鉴于上述，Tendermint 满足原子广播。

在未来的工作中，我们的目标是提供一个更正式的 Tendermint 安全性证明。

3.5 问责制（accountability）

一旦安全性被违反，一个负责的 BFT 算法可以识别出所有的拜占庭验证者。传统的 BFT 算法没有这个属性，在事件安全性上没有提供任何保证。当然，问责制只适用于三分之一到三分之二的验证者是拜占庭的时候。如果拜占庭超过三分之二，他们可以完全控制协议，我们不能保证正确的验证者会收到任何不当行为的证据。

此外，问责制可以在异步网络中最大限度地发挥作用—在违反安全性之后，关键信息的延迟交付可能直到检测到安全违规之后的某个时间，才能确定哪些验证者是拜占庭。事实上，如果正确的进程可以接收拜占庭行为的证据但是在它们可以传播下去之前不可逆转地失败，那么可能会导致问责制受到永久性损害的情况，尽管在实践中，这种情况应该是可以以高级的备份解决方案克服的。

通过列举可能发生违反安全性的可能方式，并在每种情况下展现，拜占庭式验证器都是可识别的，协议可以被证明是有责任的。Tendermint 的简单性使其比管理领导选举的协议更简单。

在 Tendermint 中只有两种违反安全规定的方式，两者都是负责任的。第一种，拜占庭提议者在一轮中提出两个冲突的提案，然后拜占庭的验证者对两者都投票。第二种，拜占庭验证者在一些验证者已经提交之后违反锁定规则，导致其他验证者在稍后的一轮中提交了一个不同的区块。请注意，不可能通过三分之二或更少的拜占庭验证者仅使用违反 Unlock-on-Polka 规则来违背安全性—超过三分之一的验证者必须违反 Prevote-the-Lock，因为应该有一个 Polka 证明剩下的诚实节点的提交。

在冲突提议和冲突投票的情况下，通过接收两个消息来检测这个冲突，并通过他们的签名来识别元凶并不重要。

在违反锁定规则的情况下，违反安全性，正确的验证者必须广播在该高度所看到的所有投票，以便将证据拼合在一起。数量不足三分之二的正确的验证者，是产生两个区块提交的的全体投票。在这些投票中，如果没有三分之一或更多的验证者签名冲突的投票，则有三分之一以上验证者违反 Prevote-the-Lock。

如果预投票或预提交影响到提交，则必须由正确的验证者看到。因此，通过收集所有投票，可以通过匹配每个预投票和被同一个验证者最近的预提交来检测违反 Prevote-the-Lock 的行为，除非不存在。

相似地，对 Unlock-on-Polka 的违法可以通过匹配每个预提交和被证明的 Polka 来检测出来。这意味着拜占庭验证者可以在看到 Polka 之前就可以预提交，而且如果一个恰当的 Polka 最终到来的话，它可以逃避问责。然而，如果不管怎样 Polka 都会产生的话，这种情况实际上并不会对安全性的违反做出什么贡献。

目前的设计在 post-crisis 广播协议中提供了问责制，但可以进一步改进以便实时问责。也就是说，一次提交可以被改变成不仅包含预提交，而且还包含所有证明预提交的投票，一路回到该高度的开始。这样一来，如果安全性被违反，可以立即检查出不合理的投票。

3.6 故障和可用性

作为一个 BFT 一致性算法，Tendermint 可以忍受最多至（但不包括）三分之一的验证者的拜占庭故障。这意味着节点会崩溃，向不同的对等点发送不同且相互矛盾的消息，拒绝中继消息，或者在不损害安全性和活性的情况下随意做出行为（伴有活性的通常 FLP 警告）。

在协议中有两个地方可以通过利用基于本地时钟的超时来优化异步：在收到三分之二或更多的预投票后，但不是单个区块或 nil，和在收到三分之二或更多的预提交后，而不是单个区块或 nil。在每种情况中，我们可以睡眠一段时间，给予较慢或延迟的投票被接收的机会，从而减少没提交一个区块就进入新一轮的可能性。时钟不需要在验证者上同步，因为每当验证者观察三分之二或更多其他验证者的投票时，时钟就将被重置。

如果三分之一或更多验证者崩溃，网络就会停止，因为没有验证者可以在没有听到其他超过三分之二验证集的情况下取得进展。该网络仍然可用于读取，但不能进行新的提交。验证者一旦恢复在线，就可以从它们离开的地方继续下去。一致性状态机应该使用预先记录日志，以便恢复的验证者可以快速返回到其崩溃时的步骤，确保它不会意外违反某项规则。

如果三分之一或更多的验证者是拜占庭，则他们可以通过多种方式来危及安全性，例如通过在同一轮中提出两个区块，并且通过将这两个都提交进行投票，或者通过违反锁规则来预提交两个在相同的高度的区块。在每种情况下，都有明确可靠的证据表明某些验证者的行为不当。在第一个例子中，他们在同一轮签署了两项提议，这明显违反了规则。在第二个例子中，他们可能已经在第 R 轮中预投票了与在 R-1 轮中锁定的不同的区块，这违反了 Prevote-the-Lock 规则。

当使用经济和治理组件来激励和管理一致性（第 6 章）时，这些额外的问责保障变得至关重要。

3.7 结论

Tendermint 是一种弱同步的拜占庭容错状态机复制协议，在违反 BFT 假设的情况下，具有最佳的拜占庭容错能力和额外的问责制保障。协议对提议者使用轮流的方法，并使用相同的机制来跳过提议一个已经提交的区块的提议者。通过一个简单的锁定机制，可以保持安全。

本章中介绍的协议忽略了许多重要的细节，例如块的高效传播，缓冲交易，验证集的更改以及与应用逻辑接口。这些重要课题将在后续章节中讨论。

Chapter 4

Tendermint 子协议

前一章中的 Tendermint 一致性的介绍，忽略了一些关于用于传播区块，投票，交易和其他对等信息的流言协议的细节。这样做是为了将注意力集中在一致性协议本身上，排除实用软件工程的干扰。本章介绍了通过将组件实现为在每个对等连接上进行复用的相对独立的反应器来实现这些细节的一种特定方法。

4.1 点对点网络（P2P-Networking）

在启动时，每个 Tendermint 节点接收要拨打的对等点的初始列表。对于每个对等点，一个节点维护一个持久的 TCP 连接，其中多个子协议以一种速率限制的方式进行多路复用。消息被序列化成紧凑的二进制表示，以在线上发送，并且通过经认证的加密协议对连接进行加密。

本章的其余部分描述了在每个对等连接上复用的单独反应器。可以运行一个额外的对等点交换反应器，允许节点请求彼此的其他对等地址，并跟踪它们之前连接到的对等点，以便保持与其他最小数量的对等点的连接。

4.2 一致性传播（consensus gossip）

一致反应器包装一致状态机，并确保每个节点每次更改时间的时候向所有对等点广播其当前状态。以这种方式，每个节点跟踪所有对等点的一致性状态，允许它优化消息的传播，以便仅在非常时刻发送他们需要的对等点信息，以及它们不具有的信息。对于每个对等点，节点维护两个连续检查新信息以发送对等体的例程，即提议和投票。信息应该以“rarest first”方式传播，以最大限度地提高传播效率，并尽量降低一些信息不可用的机会。

4.2.1 区块数据（block data）

在第三章中，假设提案消息包括区块。然而，由于区块从单一来源出现并且可能相当大，这给区块提议者带来了不必要的压力将数据上传到所有其他节点；如果将它们分成几部分传播，则可以更快地传播区块。

通过各种 p2p 协议普及的安全传播数据的常见方法是使用 Merkle 树，允许每个数据片段伴随有一个简短的证据（数据对数大小），该片段是整体的一部分。为了使用这种方法，块被序列化并且被分割成适当尺寸的块，用于期望的块大小和验证器的数量，并且块被散列到 Merkle 树中。为了使用这种方法，对于期望的区块大小和验证者的数量，区块被序列化并且被分割成适当尺寸的块，并且块被散列到一个 Merkle 树中。已签署的提议，不是包括整个区块，而是只包括 Merkle 根哈希，允许网络在传播块上合作。一个节点在每次接收到一个块时通知它的对等点，以便通过多次发送相同的块到一个节点来最小化浪费带宽。

一旦接收到所有的块，区块被反序列化和验证，以确保它正确地引用到前面的区块，并且其各种被 Merkle 树实现的校验也是正确的。虽然以前假定验证者在收到提议（包括区块）之前不会预投票，但可以通过允许验证者在收到提议之后，在收到完整的区块之前进行预投票，可以获得一些性能收益。这就意味着预投票那些最终是无效的票也是可以的。但是，无效区块的预提交必须始终被视为拜占庭。

被赶上的对等点（即处于较早高度的）会被发送区块到它们所在的高度。一次只处理一个区块。

4.2.2 投票（votes）

在一致性状态机的每个步骤中，在提案之后，节点需要等待投票（或本地超时）来往下进行。如果一个对等点刚进入一个新的高度，它将被从上一个区块预提交，所以如果它是一个提议者，它可能会在下一个区块的 LastCommit 中包含它们。如果一个对等点已经预投票但尚未预提交，或已经预提交，但尚未到下一轮，则分别发送预投票或预提交。如果一个对等点赶上来，那么它会将已提交区块的预提交发送到当前的高度。

4.3 内存池（mempool）

第 3 章几乎没有提到交易，因为 Tendermint 在每次交易的区块上运行，并且不关心单独的交易，只要它们的校验和在区块中是正确的。

交易在内存中的缓存中独立管理，而在比特币之后，它被称为 mempool。交易在接收时被应用程序逻辑验证，如果有效，则添加到 mempool，并使用有序的多播算法来传播。节点为每个对等点维护一个例程，以确保 mempool 中的交易以相同的顺序被发送到对等点，在该顺序中，它们由节点处理。

提议者从 mempool 中的有序列表中获取交易，以获取新的区块建议。一旦某个区块被提交，块中的所有交易将从内存池中删除，剩余的交易由应用程序逻辑重新验证，因为它们的有效性可能由于提交的其他交易而改变，节点可能已经不在其内存池中。

4.4 同步区块链

一致性反应器提供了与最新的区块链状态同步的相对较慢的方式，因为它被设计为实时一致性，意味着对等点等待接收所有信息以提交单个区块，然后再担心下一个区块。为了适应可能不止几个区块的对等体，一个附加的反应器，区块链反应器允许对等点并行下载多个区块，使对等点能够比使用一致性反应器同步快几百倍。

当节点连接到新的对等点时，对等点发送其当前高度。节点将从其当前高度顺序地向所有自发报告更高高度的对等点请求区块，同时下载区块，并将其添加到区块池中。另一个例程连续地尝试从池中移除区块，并通过验证和执行它们，一次两个区块，违反区块链的最新状态，将其添加到区块链。区块必须一次被两个区块验证，因为一个区块的提交作为 LastCommit 数据包含在下一个区块中。

节点连续查询其对等点的当前高度，并继续同时请求区块，直到其已经达到其对等点中的最高高度，此时它停止请求对等点高度并启动一致性反应器。

4.5 结论

对于 Tendermint 区块链的实际实现，需要许多子协议。这些包括对一致性数据（投票和提议），区块数据和交易的传播，以及一些新对等点可以快速赶上最新的区块链状态的方法。

Chapter 5

建立应用

Tendermint 被设计为一个用于复制一个确定性状态机的通用目的算法。它使用 Tendermint Socket Protocol (TMSP)来标准化一致性引擎和状态机之间的通信，使应用程序开发人员能够以任何编程语言构建其状态机，并通过 Tendermint 的 BFT 算法自动复制它们。（*可以参考 etcd 的设计，实际上，完全没有必要这样深度耦合日志和逻辑，日志归日志，逻辑归逻辑，这个部分的设计并不是非常的好。从这个角度来说，比特币的设计还是非常不错的。）

5.1 背景

互联网上的应用一般可以体现为包含两个基本要素：

•引擎（engine）：处理核心安全，网络，复制。当为一个 web 应用供电（powering）时，它通常是一个网络服务器，比如 Apache 或 Nginx，或者当它为一个分布式应用供电（powering）时，它就是一个一致性算法。

• 状态机（state-machine）：处理从引擎接收的交易并更新内部状态的实际应用程序逻辑。

这种关注点的分离使得应用的开发者可以以任何编程语言任意应用程序的形式编写状态机，位于可能专门用于其性能，安全性，可用性，支持和其他考虑因素的引擎之上。

与 Web 服务器及其应用程序不同，Web 服务器及其应用程序通常采用通过公共网关接口（CGI）协议在 socket 上进行通信的形式，但一致性算法传统上具有少得多的可用或通用接口来构建应用程序。一些像 zookeeper，etcd，consul 和其他分布式键值存储为一个简单的键值对应用的一个特定实例提供 HTTP 接口，和一些更有趣的特性比如原子比较交换（compare-and-swap）操作和通知推送。但是它们不提供给应用开发者状态机代码本身的控制。

对于在一致性引擎上运行的状态机的高水平控制的需求主要是由于比特币的成功以及随之而来的对区块链技术的兴趣驱引的。通过将更高级的应用程序直接构建到一致性算法，用户，开发人员，监管机构等之中，可以在任意状态机上实现更大的安全保障，远远超过诸如货币，交易所，供应链管理，治理等键值对存储之上。是允许集体实施代码的执行的系统的潜力吸引了如此多的注意力。它实际上是法律制度的许多方面的重新发明，使用分布式一致性算法和确定性可执行的合同，而不是警察，律师，法官，陪审团等。人类社会发展的影响力是爆炸式的，就像民主法治的引入一样。

Tendermint 旨在提供可能构建此类应用程序的基本界面和一致性引擎。

5.2 Tendermint Socket Protocol

Tendermint Socket Protocol (TMSP)定义了一致性引擎与应用状态机通信的核心接口。接口定义由许多使用 Google 协议缓冲区指定的消息类型组成，这些消

息类型通过 socket 进行长度预定义和传输。图 5.1 给出了消息类型，参数，返回值和目的的列表，并且图 5.2 中显示了体系结构和消息流的概述。

TMSP 被实现为有序的异步服务器，其中消息类型为请求和响应的对，特殊消息类型 Flush 通过连接推送任何缓冲的消息并等待所有响应。

TMSP 的核心是两个消息：AppendTx 和 Commit。

```
type Application interface {
    // Return application info
    Info() (info string)

    // Set application option
    SetOption(key string, value string) (log string)

    // Append a tx
    AppendTx(tx []byte) Result

    // Validate a tx for the mempool
    CheckTx(tx []byte) Result

    // Return the application Merkle root hash
    Commit() Result

    // Query for state
    Query(query []byte) Result

    // Signals the beginning of a block
    BeginBlock(height uint64)

    // Signals the end of a block
    // validators: changed validators from app to TendermintCore
    EndBlock(height uint64) (validators []*Validator)
}

type CodeType int32

type Result struct {
    Code CodeType
    Data []byte
    Log  string // Can be non-deterministic
}

type Validator struct {
    PubKey []byte
    Power  uint64
}
```

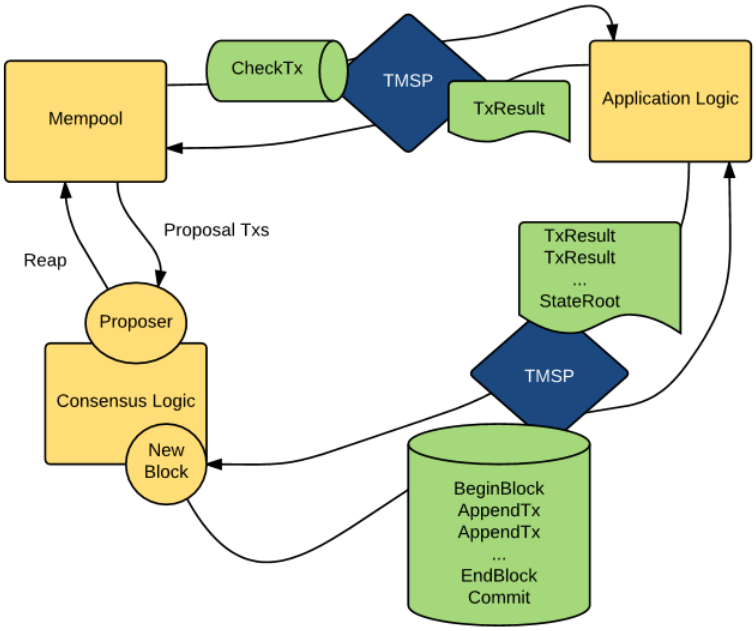


Figure 5.2: The consensus logic communicates with the application logic via TMSP, a socket protocol. Two sockets are maintained, one for the mempool to check the validity of new transactions, and one for the consensus to execute newly committed blocks.

一旦一致性决定了一个区块，引擎将在区块中的每个交易上调用 AppendTx，将其传递给要处理的应用程序状态机。 如果交易有效，将导致应用程序中的状态转换。

一旦所有的 AppendTx 调用都返回，一致性引擎调用 Commit，导致应用程序提交最新状态，并将其持续到磁盘上。

5.3 分离协议和执行

使用 TMSP 可以让我们明确分离出一致性，或一致的交易顺序，和在状态机上的实际执行情况。具体来说，我们首先在顺序上达到一致性，然后执行有序的交易。这种分离实际上提高了系统的容错能力：尽管 3f + 1 个副本仍然需要同意以容忍 f 个拜占庭故障，但执行只需要 2f + 1 个副本。也就是说，我们仍然需要三分之二的多数来排序（ordering），但是我们只需要二分之一的多数来执行（execution）。

但是另一方面，交易在排序后再执行可能导致无效的交易，这样就会浪费系统资源。这可以通过使用由内存池调用的附加 TMSP 消息 CheckTx 来解决，允许它检查交易是否对最新状态有效。但是要注意，区块的一次提交会引起 CheckTx 消息处理的复杂性。特别地，应用需要维护另一个仅执行与交易有效性相关的主状态机规则的状态机。第二个状态机由 CheckTx 消息更新，并在每次提交后重置为最新的提交状态。 实质上，第二个状态机描述了交易池的过滤器规则。

在一定程度上，CheckTx 可以用 optimistic execution，将结果返回给交易发送方，并注意到，如果在提交感兴趣的交易之前在区块中提交了一个冲突的交易，则结果可能会出错。这种 optimistic execution 是可扩展 BFT 系统的一个方法的重点，对于特殊的应用程序，在很少有交易之间的冲突的情况下，这种方法可以很

有效率。同时，由于需要处理可能的无效结果，它增加了客户端的额外复杂性。该方法在第 10 章进一步讨论。

5.4 微服务架构（Microservice Architecture）

将应用程序设计中的关心点分离为一个策略是非常明智的做法。特别地，今天许多大规模应用的部署都采用微服务架构，其中每个功能组件被实现为独立的网络服务，并且通常封装在 Linux 容器中（例如使用 Docker）以实现有效的部署，可扩展性和可升级性。

运行在 Tendermint 一致性算法之上的应用程序通常可以分解成微服务。举个例子，许多应用程序利用键值对存储来存储状态。作为独立服务运行键值对存储非常常见，以便利用数据存储的特性，如高性能数据类型或 Merkle 树。

应用程序的另一个重要的微服务是一个治理模块，它管理 TMSP 消息的某个子集，使应用程序能够控制验证集的更改。这样一个模块可以成为 BFT 系统治理的强大范例。

一些应用程序可能会为用户使用本地货币或帐户结构。 因此，提供支持例如处理数字签名和管理帐户动态的基本元素的模块可能是有用的。

组合复杂 TMSP 应用程序的可能的微服务列表继续进行。实际上，甚至可以构建一个可以使用交易中发送的数据启动子应用程序的应用程序。例如，在交易中包括 docker 图像的哈希，使得图像可以从一些文件存储后端被拉出，并作为在一致性算法的未来交易中可能导致其执行的子应用程序运行。这就是以太坊（ethereum）的方法，它允许开发人员将一些代码部署到网络中，可以通过未来的交易以及 IBM 最近的 OpenBlockChain（OBC）项目触发在 Ethereum 虚拟机中运行，从而允许开发人员发送完整的 Docker 交易中的内容，定义运行任意代码的容器，以响应发往它们的交易。

5.5 决定论（Determinism）

关于使用 TMSP 构建应用程序的最关键的注意事项是它们必须是确定性的。也就是说，对于复制的状态机来说，在不损害安全性的情况下，每个节点在同一状态下执行相同的事务时必须得到相同的结果。

这不是 Tendermint 自己特有的要求。比特币、Raft、以太坊、任何其他分布式一致性算法，以及如同 lock-step multi-player 这样的游戏等应用，都必须有严格地确定性，以免出现一致性失败的情况。

编程语言中有许多非确定性的来源，最明显是通过随机数和时间，但也可以通过浮点精度的使用，以及通过哈希表的迭代（一些语言，比如 Go，在哈希表上强制执行随机迭代，以强制程序员明确地了解何时需要有序数据结构）。决定论的严格限制，以及它每一个主要编程语言的明显缺陷，都促使以太坊开发自己的图灵完整的完全确定性的虚拟机，这些虚拟机构成了应用程序开发人员构建位于以太坊区块链之上的应用程序的平台。但是决定性有很多怪癖，比如 32 字节堆栈字，存储键和存储值，并且不支持字节移位操作—一切都是大数算术。

确定性编程在实时（real-time），锁步（lock-step），多方游戏（multi-party gaming）世界中得到了很好的研究。这样的游戏构成复制状态机的另一个例子，并且在许多方面与一致性算法非常相似。使用 TMSP 构建应用的开发者被鼓励去研究他们的方法，并且小心的实现一个应用。一方面，使用功能编程语言和验证方法可以

实现正确程序的构建。另一方面，将非确定性程序转化为规范确定性程序的编译器来正在被开发。

5.6 终止（Termination）

如果决定论对于保护安全性至关重要，则交易执行终止对于保持活性至关重要。然而，一般来说，确定一个给定的程序是否停止，即使只是单个输入，更不用说所有输入，一个称为“停止问题（Halting Problem）”的问题，通常是不可能的。

以太坊虚拟机通过计量（metering）来解决问题，即对执行中的每个操作进行计费。这样一来，当发送者用尽资金时，交易就保证会被终止。通过将程序编译为自己的计量版本（metered version）的编译器，这种 metering 可以在更一般的情况下是可行的。

没有显著的开销是难以解决这个问题的。实质上，验证者不能够判断一个执行到底是在一个无限循环中，还是只是慢但是几乎是完成的。使用 Tendermint 一致性协议来决定交易超时是可行的，这样超过三分之二的验证者必须同意一个交易超时，因此被认为是无效的（即对状态没有任何影响）。然而，我们不在这里继续纠缠这个想法，而是把它留在今后的工作中。与此同时，预计应用程序将在部署到任何一致性系统之前进行彻底的测试，并且在一致性算法失败的情况下，将使用监控和治理机制来重启系统。

5.7 例子

在本节中，将介绍和讨论越来越复杂的 TMSP 应用程序的示例，特别关注 CheckTx 和管理内存池。

5.7.1 Merkleeyes

TMSP 应用程序的一个简单示例是基于 Merkle 树的键值对存储。Tendermint 提供 Merkleeyes，一个 TMSP 应用程序，它封装了一个自平衡的，Merkle 二叉搜索树。交易的第一个字节决定交易是 get、set 还是 remove 操作。对于 get 和 remove 操作，剩下的字节是键（key）。对于 set 操作，剩下的字节是包含键（key）和值（value）的序列化列表。Merkleeyes 可以使用一个简单的 CheckTx 实现，只对交易进行解码，以确保它的格式正确。我们还可以做一个更高级的 CheckTx，在那里 get 和 remove 未知键的操作是无效的。一旦调用 Commit，最新的更新被添加到 Merkle 树中，所有的散列都被计算出来，并且树的最新状态被提交到磁盘。

请注意，Merkleeyes 旨在成为其他 TMSP 应用程序使用的模块，用于基于 Merkle 树的键值对存储，而不是独立的 TMSP 应用程序，尽管 TMSP 接口的简单性使其适用于两者。

5.7.2 Basecoin

一个更完整的例子是一个简单的货币，使用由以太坊开发的帐户结构，其中每个用户都有一个公钥和一个公共密钥的余额帐户。该帐户还包含一个序列号，该序列号等于该帐户发送的交易的数字。如果交易包含正确的序列号，并由正确的私钥签名，则可以从帐户中发送资金。没有序列号，系统将容易发生重播攻击（replay attack），即可以重播记录帐户的签名交易借记，导致借记发生多次。

此外，为了防止多链环境中的重放攻击，交易签名应该包括一个网络或区块链标识符。

支持货币的应用程序自然比简单的键值对存储有更多的逻辑。特别地，某些交易明显无效，例如签名无效，序列号不正确或发送量大于发送者账户余额的交易。这些条件可以在 CheckTx 中检查。

此外，为了更新序列号和帐户余额，补充应用程序状态必须被维护，以便一次在内存池中涉及相同帐户的多个交易时更新序列号和帐户余额。当 commit 被调用时，补充应用状态被重置为最新的提交状态。任何仍然在内存池中的交易都可以通过 CheckTx 以最新的状态重播。

5.7.3 以太坊（Ethereum）

Ethereum 使用已经描述的机制将交易从 mempool 中删除，但它也会在虚拟机中运行一些交易，从而更新状态并返回结果。CheckTx 中没有完成虚拟机的执行，因为它代价更昂贵，并且在很大程度上取决于交易包含在区块中的最终顺序。

5.8 结论

TMSP 提供了一种简单而灵活的手段，以任何编程语言构建任意应用程序，从 Tendermint 一致性算法继承 BFT 状态机复制。对于一个一致性引擎和一个应用程序来说，它扮演着更多同样的角色，例如，CGI 扮演 Apache 和 Wordpress。然而，应用程序开发人员必须特别注意确保其应用程序是确定性的，并且交易执行是结束的。

Chapter 6

管理（Governance）

到目前为止，本文已经回顾了 Tendermint 一致性协议和应用环境的基本要素。在现实世界中操作系统的关键要素，如管理验证集的变更和从危机中恢复，还没有讨论。

本章提出了一种解决这些问题的方法，以使治理在协商一致系统中发挥作用。当验证集包含更多分散的代理集时，维护网络的有效治理系统将对网络的成功变得越来越重要。

6.1 Governmint

治理的基本功能是通过一种投票方式来删除行动建议。治理作为软件最基本的实现是使用户能够提出提议，投票并统计投票的模块。提议可能是程序性的，在这种情况下，他们可以在成功投票后自动执行，或者它可能是非程序性的，在这种情况下，其执行是一个手动练习。

为了在 Tendermint 中启用某些操作，例如更改验证集或升级软件，实施了一个称为 Governmint 的治理模块。Governmint 是一个最低限度的可行治理应用，支持多组实体，每个实体可以在内部对提议进行投票，其中一些可能导致程序性执行操作，例如更改验证集，或升级 Governmint 本身（例如添加新提议类型或其他投票机制）。

该系统利用数字签名认证投票者，并可能使用各种可能的投票方案。特别感兴趣的是二次投票方案，其中投票的成本是投票权的二次方，这已被证明具有满足投票者偏好的优越能力。

6.2 验证集变更

验证集变更是现实世界一致性算法的关键组成部分，以前的许多方法都未能明确定义或被遗留为黑色艺术（black art）。Raft 艰难地阐明了验证集变更的良好协议，这需要使用新的消息类型通过一致性。Tendermint 采用了类似的方法，尽管它通过使用 EndBlock 消息的 TMSP 接口进行标准化，该消息在所有 AppendTx 消息之后运行，但在 Commit 之前运行。如果交易或一组交易被包括在具有更新验证集的预期效果的区块中，则应用程序可以通过响应 EndBlock 消息指定其公钥和新的投票权力来返回验证者列表以进行更新。可以通过将其投票权设置为零来删除验证者。这为应用程序提供了更新验证集而不必指定交易类型的通用方法。

如果在高度 H 处的区块返回已更新的验证集，则高度 H + 1 处的区块将反射更新。但是请注意，H + 1 区块中的 LastCommit 必须使用与 H 处一样的验证集，因为它可能包含已被删除的验证器的签名。

投票权的更改适用于 H + 1，以便下一个提议者可以受到更新的影响。否则的话，应该是下一个提议者的验证者可能会被删除。轮流算法应该优雅地直接地转到下一个提议者来处理。由于相同的区块在至少三分之二的验证者上被复制，并且轮流是确定性的，所以它们将全部都进行相同的更新，并期望同一个下轮提议者。

6.3 惩罚拜占庭验证者

比特币设计的一个显著特点是它的激励结构，目前为止，协议的目标是通过奖励验证者来激励验证者的正确行为。虽然这在比特币一致性协议的背景下是有道理的，但优越的激励可能会提供强烈的抑制效果，使得验证者具有真正的 skin-in-the-game，而不是一个软机会成本（soft opportunity cost）。

在 Tendermint 中可以使用 Vitalik Buterin 提出的方法作为所谓的 Proof-of-Stake 协议来实现消除负面效果。实质上，验证者必须提供保证金（“他们必须保证一定的赌注”）才能参与一致性。如果他们被发现签署双重提议或投票，其他验证者可以以交易形式发布违规的证据，应用程序状态可以通过删除违规者来修改验证集，从而烧毁其存款。这具有将明确的经济成本与拜占庭行为联系起来的效果，并且能够通过将三分之一或更多的验证者贿赂为拜占庭来估计违反安全性的成本。

请注意，一致性协议可以指定惩罚更多的行为，而不仅仅是双重签名。特别是，我们有兴趣惩罚不公正的任何强烈的信号行为—通常为任何不是基于其他人报告的状态报告的状态变化。举个例子，在 Tendermint 的一个版本中，所有预提交必须伴随 polka 来证明它们，验证者可能会因广播为被裁定的预提交而受到惩罚。但要注意的是，我们不能惩罚所有未预期到的行为—比如说，一个验证者在不是它提议的回合做出提议可能是优先考虑异步和节点崩溃优化的基础。

实际上，沿着这两条线概况 Tendermint，1）更为宽松的判定形式，2）允许验证者在其任期之前提出建议，从而产生一个性质与 Vlad Zamfir 所提出的相似的协议族，以 guise Casper 作为未来以太坊版本的一致性机制。关于协议之间的关系和反拜占庭判定的特征的更正式的说明仍然是未来的工作。

6.4 软件更新（Software Upgrades）

Governmint 也可以作为在可能分散的网络上协商软件升级的自然手段。在公共互联网上的软件升级是一个非常具有挑战性的操作，需要仔细规划来维护不再升级的用户的向后兼容性，并且不会通过引入漏洞，删除功能，增加复杂性，或者可能最糟糕的，未经许可自动更新来使软件的忠实用户沮丧。

比特币尤其明显地提高了分散式一致性系统的更新难度。以太坊由于其强大的领导力和统一的社区已经管理了一个成功的非向后兼容的升级版本，而 Bitcoin，放下软件工程中过多的弊病不说，由于一个恶性分裂的社区和缺乏强有力的领导，而无法进行一些必要的升级。

就变更的范围而言，区块链的更新一般分为 soft forks 和 hard forks。Soft forks 旨在向后兼容，并且使用协议中可能被尚未升级的用户忽略的自由度，为用户提供新功能。另一方面，hard forks 是非向后兼容的升级，在比特币的例子中，可能会导致违背安全性，而在 Tendermint 的例子中，会导致系统终止。

为了应对，比特币软件的开发人员推出了一系列 soft forks，验证者可以通过在新的区块中发信号来投票。一旦验证器的某个阈值是更新的信号，至少对于支持更新的软件版本的用户，它将自动在网络上生效。由于这些 soft forks，比特币系统的效用大大增加，预计在未来将而继续这样做。有趣的是，社区未能成功地 hard fork 软件，一方面引起了人们对该系统长期稳定性的担忧，另一方面引发了人们对该系统对腐败治理—它的不可治理性—的恢复力的兴奋和鼓舞。

鉴于今天世界上过多的政府腐败，有很多理由采取后一种立场。然而，密码学和分布式一致性算法提供了一套新的缺乏强大的身份验证系统的工具，可以在现代政府的“纸、笔、握手”的世界中甚至是传统网络的数字世界中产生不可想象的一定程度的透明度和责任感。

在使用 Governmint 的系统中，开发人员将是区块链上的标识实体，可以提交软件升级提议。该机制与 Github 的 Pull Request 非常相似，只有它被集成到现场运行系统中，共识通过一致性协议传递。客户端应写入可配置的更新参数，这样他们可以指定是自动更新还是更新前需要通知。

当然，任何未经彻底审查的软件升级都可能对系统构成危险，所以通常应该采取保守的方式升级。

6.5 危机恢复（Crisis Recovery）

在发生危机的情况下，比如交易日志中的 fork，或者系统陷入停顿，传统的一致性系统很少甚至没有提供保证，通常需要人工来干预。

Tendermint 保证，违反安全性责任的可以被识别，使得任何可以访问至少一个诚实的验证者的客户端都可以识别不诚实的验证者的加密确定性，从而选择将诚实的验证者跟踪到具有一个不包括拜占庭的验证集的新链上。

举个例子，假设三分之一或更多的验证者违反锁规则，导致在高度 H 处有两个区块要被提交。诚实的验证者可以通过传播所以投票来确定是谁双重签名的。在这一点上，由于基本的错误假设被违反，它们不能使用一致性协议。注意，能够在这一点上为 H 累积所有的投票意味着关于在危机期间网络连接和可用性的重要假设，如果他不能由 p2p 网络提供，它可能需要验证者们使用替代方案，比如社交媒体和高可用性的服务来交流证据。一旦其中至少有三分之二收集了所有的证据，全套的余留诚实节点可以启动一个新的区块链。

或者，修改 Tendermint 协议，以便预提交要求 polka 将确保负责 fork 的可以立即受到惩罚，并且不需要额外的发布期。这个更改工作留给以后来做。

使用更复杂的 Governmint 来适应各种特定的危机是可行的，比如永久性崩溃故障和私钥的危害。但是，这些方法必须仔细考虑，因为它们可能会破坏基础协议的安全性保障。我们以后将对这些方法进行调查，但是要注意，理解其从危机中恢复的能力的嵌入区块链的社会经济背景的重要性。

不管危机如何恢复，其成功取决于与客户端的整合。如果客户端不接受新的区块链，则该服务实际上处于脱机状态。因此，客户端必须了解特定区块链所用的恢复规则。在上述违背安全性的情况下，他们还必须收集证据，确定要删除哪些验证者，并用剩余的验证者计算新的状态。在违反活性的情况下，必须保持 Governmint。

6.6 结论

治理是分布式一致性系统的一个关键因素，尽管有关治理系统仍然知之甚少。Tendermint 以一种叫做 Governmint 的 TMSP 提供治理，旨在促进在分布式系统的基于软件的治理中的增强实验。

Chapter 7

客户端注意事项（Client Considerations）

本章回顾了与托管在 Tendermint 上的应用程序交互的客户端的一些注意事项。

7.1 发现

网络发现只需通过 TCP 连接一些种子节点即可。p2p 网络使用经过身份验证的加密，但是验证者的公钥必须以某种方式从 band 中验证，即通过一种替代的媒介，而不是在协议的范围内。实际上，在这些系统中，起源状态本身必须要被传出 band 以外，理想情况下，也是唯一必须要被通信出去的东西，因为它还应该包含验证者使用的用来验证加密的公钥，与在一致性中用于签署投票的方式是不同的。

对于可能随时间变化的验证集，通过 DNS 注册所有验证者以及在新的验证者实际成为验证者之前注册它们是有用的，并在其以验证者身份被删除后将其删除。或者，验证者位置可以注册在另一个容错分布式数据存储中，可能包括另一个 Tendermint 集群本身。

7.2 广播交易

作为通用应用平台，Tendermint 仅为广播交易的客户端（clients）提供简单的接口（interface）。一般范例是客户端通过代理连接到 Tendermint 的一致性网络，该代理可以在其本地机器上运行，也可以托管在其他提供商。代理功能作为网络上的非验证者节点，这意味着它可以跟上一致性并处理交易，但不会签署投票。该代理使客户端交易能够通过传播层快速广播到整个网络中。

节点只需要连接到网络上的另一个节点来广播交易，但默认情况下会连接到许多节点，从而最小化交易不会被接收的机会。交易被传递到内存池，并通过要在所有节点的内存池中被缓存的内存池反应器传播，以便最终它们其中的一个可以将它包含在一个区块中。

请注意，交易在进入一个区块之后才会执行它的状态，所以客户端不会马上得到结果，而不是确认它被接受到 mempool 并广播给其他对等点。客户端应该在一个区块的提交过程中计算出来的时候对代理进行注册，以接收其作为推送通知的结果。

客户端连接到当前提议者并不是必需的，因为最终任何在其内存池中具有交易的验证者都可以提出它。然而，在网络处于高负荷下的某些情况下，对下一个提案的优先广播可能会降低交易的延迟。不然的话，交易应该迅速被传播到每个验证器。

7.3 内存池（Mempool）

内存池负责在内存被包含在区块中之前缓存交易。它的行为很微妙，对整个系统架构构成了一系列的挑战。首先，在内存池中缓存任意数量的交易可以直接拒绝可能会使网络瘫痪服务攻击。大多数区块链使用其原生货币来解决这个问题，并且只允许花费一定费用的交易留在内存池中。

在一个更广泛的，不必须一种货币支付的系统中，比如 Tendermint，系统必须建立更严格的过滤规则，并依靠更智能的客户端重新提交被丢弃的交易。然而，这种情况更为微妙，因为在 mempool 中过滤交易的规则集必须是应用程序本身的一个功能。因此，TMSP 的 CheckTx 消息，mempool 可以使用来针对应用程序的瞬态状态（transient state）运行交易，以确定是否应该保留或丢弃。

尽管在许多示例应用程序中提供了例子，但处理瞬态是不重要的，并且可以把它留给应用程序开发人员。在任何情况下，客户端必须监视内存池的状态（即未确认的交易），以确定他们是否需要重新广播他们的交易，这可能发生在高度并发的，其中一个交易的有效性取决于已处理的另一个交易的设置中。

7.4 语义（Semantics）

Tendermint 的核心一致性算法只提供至少一次（at-least-once）的语义，也就是说系统常受到重放攻击（replay attacks）的影响，即同一个交易可以被多次提交。然而，许多用户和应用程序期望能从数据库系统得到一个更强的保证。Tendermint 系统的灵活性把这些语义的严格性留给应用程序开发人员。通过利用 CheckTx 消息，和充分管理应用程序中的状态，应用程序开发人员可以提供适合他们和他们的用户的需求的数据库语义。比如说，如第 5 章所述，使用基于具有序列号的帐户系统可以减轻重放攻击，并将语义从至少一次（at-least-once）更改为恰好一次（exactly-once）。

7.5 读取（Reads）

客户端将读请求发送到用于广播交易（写入）的同一个代理节点。即使网络停止了，代理始终都可用于读取。然而，在分区的情况下，代理可以与网络的其余部分分开，继续制造区块。在这种情况下，代理的读取可能会过时。

为了避免过时的读取，读取请求可以作为交易发送，假定应用程序允许这样的查询。通过使用交易，保证读取返回最新的提交状态，即在下一个区块中提交读取交易时。这当然比查询状态的代理更昂贵。可以使用启发式方法来确定读取是否过期，例如代理与其对等点是否良好连接，并且正在制作区块；或者是否它被卡在有三分之一或以上的验证者的投票的回合中， 但没有替换执行实际的交易。

7.6 Light Client Proofs

在传统数据库中，区块链的主要创新之一是他们有意识地使用 Merkle 哈希树来支持系统子状态的紧凑证明的成果，即所谓的 light-client 证明。一个 light client proof 是一个允许客户端验证一些键值对是在有一个给定跟哈希的 Merkle 树中的 Merkle 树的路径。状态的 Merkle 根哈希包含在区块的头（header）中，这样就足以让客户端只有最新的头来验证状态的任何组件。当然，要知道 header 本身是有效的，它们必须已经验证了整个链，或者只保持最新的验证集更改，并且依赖于状态转换是正确的经济保证。

7.7 结论

Tendermint 网络功能的客户端与任何其他分布式数据库的相似，尽管必须要考虑到基于区块的提交的性质以及内存池的行为。另外，客户端必须考虑到以特定的应用设计。 虽然这增加了一些复杂性，但是它具有极大的灵活性。

Chapter 8

实现

Tendermint 的参考实现是由 Go 编写的，托管在 Github 上。Go 是一种具有丰富标准库的，轻量级大规模并发执行的并发原语，以及为简化和高效率优化的开发环境的，类似 C 的语言。

代码使用了大量足够模块化来隔离成自己的库的包。这些包大部分是由 Jae Kwon 编写的，包括漏洞修复，测试，以及一些作者贡献的偶然特性。这些包中最重要的部分在下面的小节中描述。

8.1 二进制序列号（Binary Serialization）

Tendermint 使用了一种为简单性和决定性优化的二进制序列化算法。它支持所有整数类型（包括使用一个字节长度的前缀编码的变体），字符串，字节数组和时间（毫秒精度的 unix 时间）。它还支持任何类型和结构体的数组（编码为有序值的列表，忽略键）。在某种程度上，它灵感来自 Go 的类型系统，特别是它使用的接口类型，可以实现为许多具体类型之一。可以注册接口，并且每个具体实现都在其编码中给出了一个领先的类型字节。

详情参见 https://github.com/tendermint/go-wire。

8.2 密码学

一致性算法，如 Tendermint 使用三个主要加密原语：数字签名，哈希函数和验证加密。虽然存在这些原语的许多实现，但为企业软件选择加密库亦不是轻松的任务，特别是世界上最常用的安全库 OpenSSL 的深刻的不安全性。

导致加密系统不安全的原因是，有像与 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的 NSA，这样的政府机构有意的破坏其安全属性。鉴于这些机构明显不合法，例如爱德华斯诺登（Edward Snowden），以及试图破坏公共密码标准的历史，许多密码学社区更倾向于使用在开放的学术环境中设计的算法。同样，Tendermint 也只使用这种算法。

Tendermint 使用了 RIPEMD160 作为它的加密哈希函数，它产生 20 字节的输出。它用于交易和验证签名的 Merkle 树，并用于计算块散列。Go 在其扩展库中提供了一个实现。在从公钥派生的地址中，比特币也使用 RIPEMD160 作为两种哈希函数中的一种。

作为其数字签名方案，Tendermint 在 ED25519 椭圆曲线上使用 Schnorr 签名。ED25519 是由丹伯恩斯坦（Dan Bernstein）开源设计的，其目的是高性能，易于实现，但不引入漏洞。伯恩斯坦还引入了 NaCl，一个高级库，用于使用 ED25519 曲线进行经过身份验证的加密。Tendermint 使用其扩展库中提供的实现。

8.3 Merkle 哈希树

Merkle 树的功能非常类似于其他基于树的数据结构，还有额外的功能，它可以生成树中一个键的成员身份证明，而树的大小是对数的。这是由递归地连接和哈希键组合完成的，直到只剩下一个哈希，树的根哈希。对于树上的任何叶子，从它到根的一条哈希的痕迹作为它的成员的证明。这使得 Merkle 树特别适用于 p2p 文件共享应用程序，在该应用程序中，可以将大型文件的片段作为属于文件的内容进行验证，而不需要所有的片段。Tendermint 将此机制用于网络上的传播块，其中根哈希被包含在区块提议中。

Tendermint 还提供了一个自我平衡的 Merkle 二叉树，以 AVL 树为模型，作为一个名为 Merkleeyes 的 TMSP 服务。IAVL 树可以用于存储动态大小的状态，允许在对数时间内查找、插入和删除。

8.4 RPC

Tendermint 公开了 HTTP API，用于查询区块链、网络信息、一致性状态以及广播交易。同样的 API 可以通过三种方法获得：使用 URI 编码参数的 GET 请求，使用 JSONRPC 标准的 POST 请求，以及使用 JSONRPC 标准的 websockets。Websockets 是高交易吞吐量的首选方法，也是接收事件的必要条件。

8.5 P2P 网络

第 4 章更详细地描述了 Tendermint 使用的 P2P 子协议。

8.6 反应器（Reactor）

Tendermint 节点由多个并发反应堆组成，每一个都管理一个状态机在网络上给对等点发送和接收消息，如第 4 章所述。反应器通过锁定共享数据结构来同步，但是同步的点被保持在最小值，这样每个反应器就会和其他反应器同时运行。

8.6.1 内存池（Mempool）

Mempool 反应器管理 mempool，在它们被打包在区块上并提交之前缓存交易。Mempool 使用应用程序状态机的一个子集来检查交易的有效性。交易保存在一个并发链表结构中，允许安全写入和许多并发读取。新的、有效的交易被添加到列表的末尾。每个对等点遍历列表，将每个交易只需要一次有序地发送到对等点。该列表还被扫描为一个新的提议收集交易，并在每次提交一个区块时更新：提交了的交易被删除，未被提交的交易通过 CheckTx 被重新运行，以及那些已经失效的交易被删除。

8.6.2 一致性（Consensus）

一致性反应器管理一致性状态机，它处理提议、投票、锁和实际提交区块。状态机使用一些持久化的例行程序（go-routines）来管理，这些例行程序接收到消息，并使它们能够确定地回放，以调试状态。这些例行程序包括 readLoop，用于读取接收到的消息队列和 timeoutLoop，用于注册和触发超时事件。

在一致性状态机中，当收到完整的提议和区块时，或者在给定的回合中收到超过三分之二的预投票或预提交时，则会进行转换。转换导致了建议、区块数据或投票的广播，这些数据在内部队列（internalReqQueue）上排队，由 readLoop 在串行上处理从对等点接收到的消息。这将内部消息和对等消息放在平等的基础上，并将其输入到一致性状态机中，但允许内部消息被更快地处理，因为它们不像来自对等点那样处于相同的队列中。

8.6.3 区块链（Blockchain）

区块链反应器采用比一致性反应器更快的技术来同步区块链。也就是说，验证者请求区块的递增高度，直到他们的对等点没有任何更高的高度的区块。区块被收集在一个区块池（blockpool）中，并通过一个工作例程同步到区块链，它周期性地从池中取区块，并对当前链进行验证。

一旦区块链反应器完成同步，它就会启动一个一致性反应器来接管。

8.7 结论

Tendermint 的 Go 语言实现充分利用了语言的并发原语、垃圾回收和类型安全性，以提供一个清晰的、模块化的、易于阅读的，包含许多可重用组件的代码。如第 9 章所示，该实现获得高性能，并对许多不同类型的错误具有强大的鲁棒性。

Chapter 9

性能和容错

Tendermint 被设计为一种拜占庭容错的状态机复制算法。只要不到三分之一的验证者是拜占庭，它就保证安全性，类似地，只要网络信息最终和用来传播协议的有关网络同步的弱假设被传送，它就保证了活性。在本节中，我们通过注入宕机故障和拜占庭故障来经验性地评估 Tendermint 的容错能力。目标是，表明在发生这种故障的情况下，Tendermint 一致性的实现并不会影响安全性，因此可以最小化性能受到的影响，并且可以很快地恢复。

Tendermint 算法的性能可以通过几种关键方式进行评估。最明显的措施是区块提交时间，它是最终延迟的度量，以及衡量网络容量的交易吞吐量。我们为每个分布在全球范围内的验证者收集数据，验证者的数量范围都是 2 从 2 到 64 的倍数。

9.1 总览

本章的实验使用存在 https://github.com/tendermint/network_testing 的 repository 来复制。所有的实验都在 docker 容器中进行，这些容器运行在 t2.medium 或 c3.8xlarge 类型的 Amazon EC2 实例上。t2.medium 有 2 个 vCPU 和 4 GB 的 RAM，c3.8xlarge 有 32 个 vCPUs 和 60 GB 的 RAM。实例分布在横跨五大洲的 7 个数据中心。另一个负责生成交易的 docker 容器在每个实例上运行。交易

的大小是 250 字节（一个包括 32 或 64 字节的哈希和签名的合理的大小），并被构造成可调试的，可以快速生成的，并包含一些随机性的。因此，前面的字节是表示该实例的交易号和验证者索引的 Big-Endian 编码整数，后面的 16 个字节是从操作系统中随机生成的，中间的字节全是零。

使用网络监视工具来维护每个验证者的 Tendermint RPC 服务器的活动的 websocket 连接，并且在第一次接收到新的提交的区块作为该区块的官方提交时间时，使用其本地时间。实验首先在没有监视器的情况下运行，通过复制验证者中的所有数据来进行分析，并使用 2/3th 验证者提交区块的本地时间作为提交时间。使用监视器的速度要快得多，能够在线监测，而且被发现只要区块头信息（不是整个区块）通过 websockets，就不会影响结果。

远程机器上的 Docker 容器很容易使用 Docker-machine 工具进行管理，而网络测试存储库提供了一些工具，可以利用 Go 的并发特性，同时在许多远程机器上执行 Docker 容器上的操作。

每个验证者都直接连接彼此，以避免网络拓扑的混淆效应。

对于涉及宕机故障或拜占庭行为的实验，故障节点的数量由 $N_{fault} = [(N - 1) / 3]$ 给出，其中 N 是验证者的总数。

9.2 吞吐量和延迟（Throughput and Latency）

本节描述了在非对抗性条件下测试 Tendermint 的原始性能的实验，其中所有节点都在线并同步，并且不为异步提供任何设施。也就是说，使用了一个人为的高的 TimeoutPropose（10 秒），所有其他超时参数均设置为 1 毫秒。此外，所有的 mempool 活动都是禁用的（在提交之后不会传播交易或重新检查它们），而在进程内的 nil 应用程序则用于绕过 TMSP。这可以作为一个控制场景，用于评估在故障和/或异步的情况下的性能下降。

实验运行在大小为 2 的从 2 到 64 倍的验证集和大小为 2 的从 128 到 32768 被的区块上。交易被预先加载到每个验证器上。每个实验运行 16 个区块。

从图 9.1 可以看出，Tendermint 可以轻松在大约一秒钟的区块延迟里处理每秒数千个交易，尽管每秒大约有一万个交易的容量限制。16384 个交易的区块的大小约为 4 MB，网络带宽分析显示每个连接很容易达到 20MB / s 以上，尽管日志分析表明，在高的区块大小时，验证者可以花费高至两秒钟时间等待区块部分。此外，如图 9.2 所示，在单个数据中心进行的实验表明，在更大的机器上进行的实验可以得到更高的吞吐量的可能，而在更大的机器上的实验显示出更一致的性能，从而消除了容量限制，如图 9.3 所示。我们将在未来进一步进行调查这一容量限制的工作。

在随后的实验中，注入了各种形式的故障，并给出了延迟的统计数据。每个实验都是在大小为 2 的从 4 到 32 倍的验证集，不同的 TimeoutPropose 值，和大小为 2048 个交易的区块上运行的。

9.3 宕机故障（Crash Failure）

为了评估一个常遭受宕机故障的网络的性能，每 3 秒钟 N_{fault} 个验证者被随机选择，停止，并在三秒钟后重新启动。

在表 9.1 中的结果表明，在这个宕机故障场景下的性能下降了大约 50%，而更大的 TimeoutPropose 值有助于调节延迟。当平均延迟时间增加到大约 2 秒时，中间值接近 1 秒，延迟可能高达 10 到 20 秒，但在一个情况下，它高达 70 秒。

很有可能修改时间的提议是稍微不确定的，这可能会降低这种极端的延迟的可能性。将 TimeoutPropose 修改为稍微不确定可能会降低这种极端延迟的可能性。

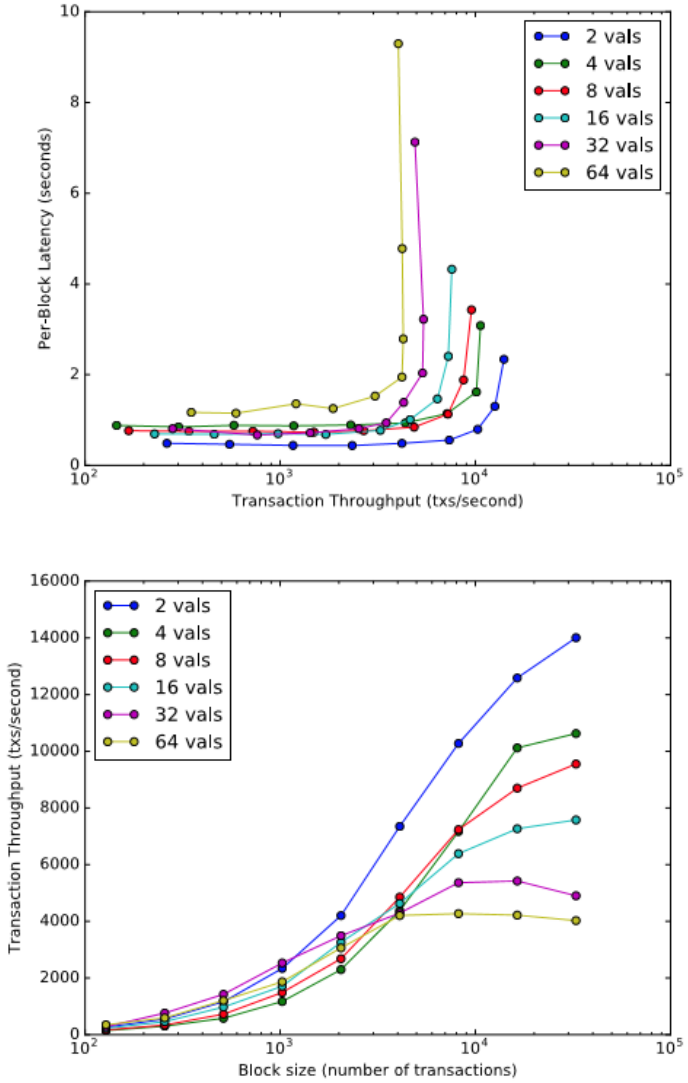
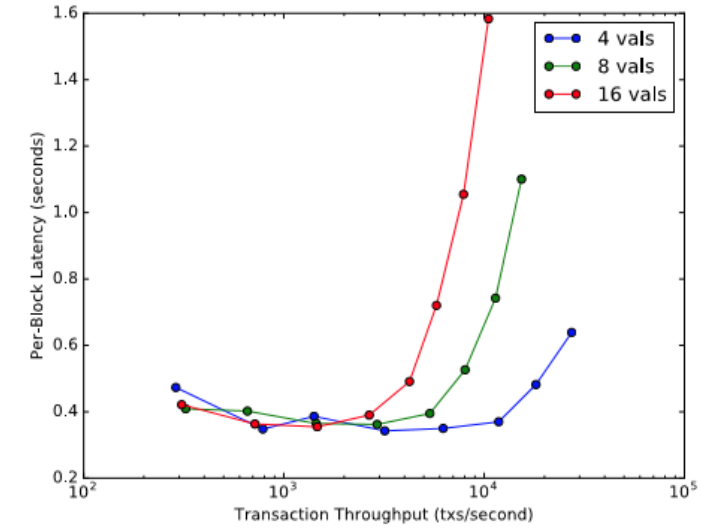


Figure 9.1: Latency-throughput trade-off. Larger blocks incur diminishing returns in transaction throughput, with an ultimate capacity at around 10,000 txs/s



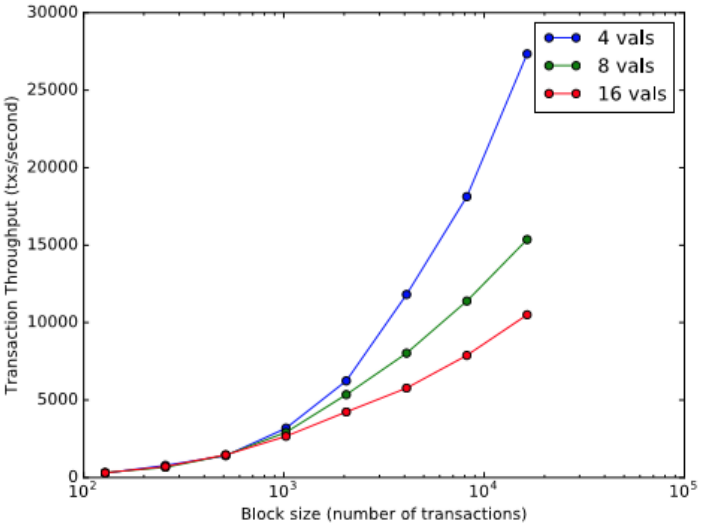


Figure 9.2: Single datacenter. When messages don't need to cross the public Internet, Tendermint is capable of tens of thousands of transactions per second.

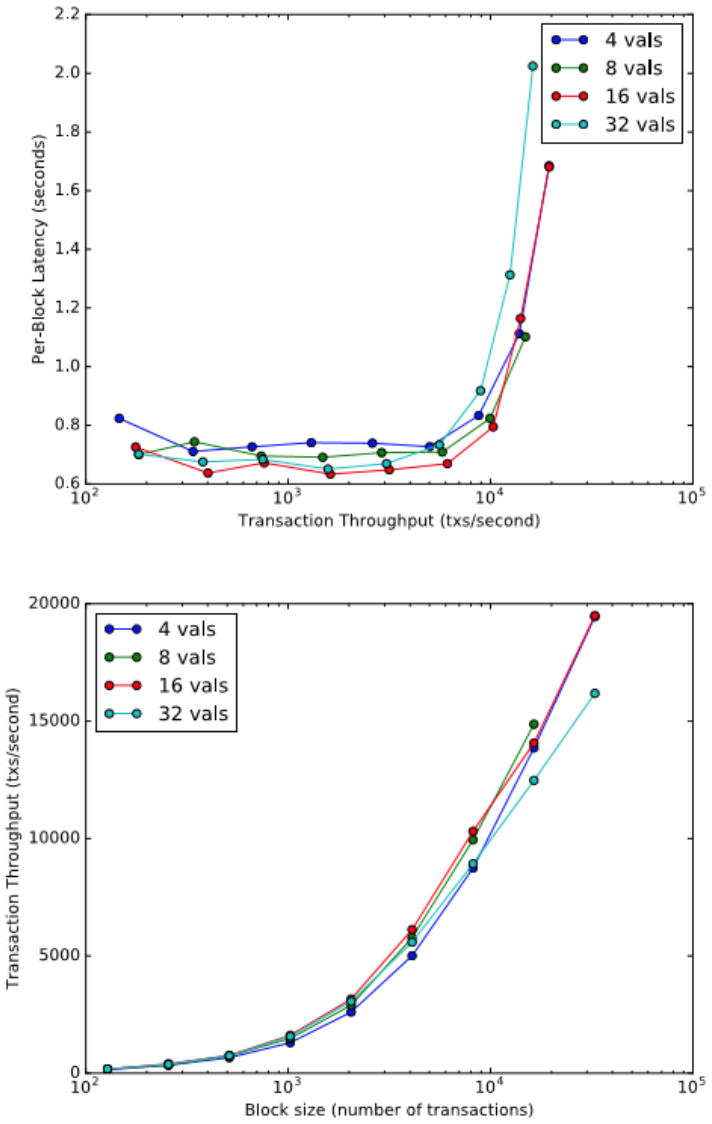


Figure 9.3: Large machines. With 32 vCPU and 60 GB of RAM, transaction throughput increases linearly with block-size, relieving the capacity limits found on smaller machines.

TimeoutPropose	Min	Max	Mean	Median	95^{th} % − *ile*
500	434	15318	2179	1102	5575
1000	516	18149	2180	1046	5677
2000	473	15067	2044	1049	5479
3000	428	9964	2005	1096	5502

(a) 4 Validators

TimeoutPropose	Min	Max	Mean	Median	95^{th} % − *ile*
500	618	126481	2679	990	5589
1000	570	9832	1763	962	5835
2000	594	8869	1658	968	5481
3000	535	10101	1633	959	5485

(b) 8 Validators

TimeoutPropose	Min	Max	Mean	Median	95^{th} % − *ile*
500	782	21354	1977	1001	5930
1000	758	12659	1761	981	5642
2000	751	21285	2041	1005	6872
3000	719	72406	2395	991	5987

(c) 16 Validators

TimeoutPropose	Min	Max	Mean	Median	95^{th} % − *ile*
500	760	24692	2591	1087	14025
1000	755	19696	2328	1119	9321
2000	852	21044	2178	1141	6514
3000	763	25587	2289	1119	6707

(d) 32 Validators

Table 9.1: Crash-fault latency statistics. Every three seconds, a random selection ofN_{fault} validators were crashed, and restarted three seconds later. This crash-restart procedure continued for 200 blocks. Each table reports the minimum, maximum, average, median, and 95^{th} percentile of the block latencies, for varying values of the TimeoutPropose parameter.

9.4 随机网络延迟（Random Network Delay）

可能归因于拜占庭行为或网络异步的另一种形式的故障是在每次读取和写入网络连接时注入随机延迟。在这个实验中，在每个网络连接读取和写入之前，N_{fault} 个验证者睡眠 X 毫秒，其中 X 被均匀地绘制在（0，3000）上。 从表 9.2 可以看出，延迟与宕机故障的情况类似，尽管 TimeoutPropose 的增加有相反的效果。由于并非所有验证者都有故障，所以小 TimeoutPropose 值可以允许快速地跳过错误的验证者。如果所有验证者都受到网络延迟的影响，因为不会有无故障的验证者跳过，则较大的 TimeoutPropose 值会减少延迟，并且将会提供更多的时间来接收延迟的消息。

9.5 拜占庭故障（Byzantine Failures）

通过以下修改可以将更明确的拜占庭故障注入状态机：

• 冲突提议（conflicting proposals）：在提议时，拜占庭验证者会签署两个冲突的提议并且广播两者，并附上预投票和预提交，以将其连接的对等点分为两半。

• 无 nil 投票（no nil votes）：一个拜占庭验证者从不签署一个 nil 投票。

• 签署每个提议（sign every proposal）：一旦一个拜占庭验证者看到一次提议，它就为每个它看到的提议提交一个预投票和一个预提交。

综上所述，这些行为明确地违反了双签名和锁规则。然而，请注意，这种行为主要是由相互矛盾的建议的传播和最终它们其中之一的提交所主导的。更复杂的拜占庭战略安排留给未来的工作。

从表 9.3 可以看出，尽管被注入了将导致许多系统立即彻底失败的拜占庭故障，但是，Tendermint 仍然保持着令人预期的延迟。由于这些故障与异步无关，所以在 TimeoutPropose 上没有实际的影响。性能也会随着更大的验证集而消失，这可能是处理拜占庭投票的简单算法的结果。

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	873	2796	1437	1036	2627
2000	831	4549	1843	1180	4036
3000	921	5782	2273	1251	5491
4000	967	6875	2700	1413	6781

(a) 4 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	870	2840	1449	1040	2786
2000	957	4268	1848	1076	4148
3000	859	5724	2156	1100	5649
4000	897	11859	3055	1093	11805

(b) 8 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	914	5595	1821	1135	5466
2000	950	7782	2490	1165	7650
3000	978	10305	3049	1163	9890
4000	1018	6890	2808	1174	6813

(c) 16 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	1202	8562	2219	1349	5733
2000	1196	7878	2549	1365	7579
3000	1164	10082	3003	1382	9805
4000	1223	17571	3696	1392	12014

(d) 32 Validators

Table 9.2: Random delay latency statistics. N_{fault} validators were set to inject a random delay before every read and write, where the delay time was chosen uniformly on $(0, 3000)$ milliseconds.

9.6 相关工作

本章中的吞吐量实验建模在[67]中，其基准是 PBFT 实现的性能和称为 HoneyBadgerBFT 的新的随机 BFT 协议。在其结果中，PBFT 在四个节点上实现了每秒超过 15000 次交易，但随着节点数量的增加而呈指数下降，而 HoneyBadgerBFT 每秒大概达到 10000 到 15000 次交易。然而，HoneyBadgerBFT 中的区块延迟要高得多，对于大小为 8，16 和 32 的验证器集，更接近 10 秒，对于更大的验证集延迟甚至可能更大。

Jepsen 是研究一致性实现的一个著名工具，它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。用 Jepsen 测试 Tendermint 是留给未来的一个令人兴奋的领域的工作。

面对持续的拜占庭故障，作者并没不了解任何的吞吐量实验，就像其他在这里描述的那样。

9.7 结论

由作者和Jae Kwon编写的Tendermint的实现轻松地在分布在全球范围内机器上的多至64个节点上达到了每秒数千笔的交易，延迟主要在1到2秒范围内。这比其他解决方案更具竞争力，尤其是在区块链的目前状态，例如，比特币的上限为每秒7笔交易。此外，我们的实现被证明对崩溃故障、消息延迟和故意的拜占庭故障都很鲁棒，能够在每个场景中保持每秒超过1000个交易。

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	868	3888	1450	1086	3320
2000	929	4375	1786	1272	4166
3000	881	4363	1224	1099	1680
4000	824	8256	1693	1272	2607

(a) 4 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	771	3445	1472	916	3288
2000	731	3661	1426	902	3339
3000	835	6402	1912	962	6155
4000	811	4462	1512	964	3592

(b) 8 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	877	15930	2086	1024	5844
2000	808	5737	1580	1027	4155
3000	919	10533	1801	1110	4174
4000	915	5589	1745	1095	4181

(c) 16 Validators

TimeoutPropose	Min	Max	Mean	Median	$95^{th}\ \% - ile$
1000	1594	11730	2680	1854	5016
2000	1496	17801	3430	1874	11730
3000	1504	15963	3280	1736	9569
4000	1490	24836	3940	1773	12866

(d) 32 Validators

Table 9.3: Byzantine-fault latency statistics. Byzantine validators propose conflicting blocks and vote on any proposal as soon as they see it. Each table reports the minimum, maximum, average, median, and 95^{th} percentile of the block latencies, for varying values of the TimeoutPropose parameter.

Chapter 10

相关工作

拜占庭一致性算法有着丰富的历史，包括密码学、分布式计算和经济学，但其部署在工业上的产品的社会经济背景直到最近才出现，至少在传统的关键实时系统之外，比如飞行器控制。一方面，比特币的发明和“区块链”一词的产生，使一种不受单一实体控制的分布式分类帐的概念得到普及，使用密码学和一致的

经济激励措施来保护面对拜占庭故障的安全性。另一方面，服务器的持续商品化，以“云（The Cloud）”的形式，以及 Raft 的发明，在主流开发者文化中推广了分布式计算，并再次把关注点引向分布式一致性算法使得成为在大规模部署中的协调中心。

在这个十字路口，有一系列的解决方案，通常适用于银行和金融应用，也适用于治理，物流和其他一般的协调形式，借鉴经典学术 BFT 修改的和现代化的各种方式。本章回顾了这些想法的历史和多样性，目的是提供一个丰富的背景来理解区块链现象。

10.1 开始

分布式算法首先出现在 19 世纪后期的电信和铁路行业，试图有效地处理传输中的多个并发消息流，或者是同一组轨道上的多个列车。

关于这个问题的学术研究似乎是由 Edsger Dijkstra 在互斥问题上的开创性工作和 Tony Hoare 在描述通信进程的模型发起的。

在这段时间里，一系列有好记名字的并发性问题都得到了推广，其中包括吸烟者的问题，吸烟者围坐在一张桌子周围，每个人都有不同的原料，并且必须成功地卷一支完整的香烟；哲学家就餐问题，坐在桌子周围的哲学家必须轮流吃饭和思考，但每个人只能在他的邻居都在思考时才能吃饭；两军问题或协同攻击问题，两名将军必须同时从远处协调进攻一个敌人的城市。

这些问题将重点放在如如信号量、互斥和通信信道的同步原语上，并将为未来几十年的许多改进打下基础。

10.1.1 故障（Faulty Things）

在 70 年代末，容错分布式计算有效地出现在了利用微处理器进行飞机控制的努力中，从而产生了许多早期的系统。今天，它成为为了 NASA 对 BFT 研究的标准和商用飞机对 BFT 系统，如 SAFEbus，的使用。

然而，许多系统不需要容忍拜占庭故障，因为它们在受控环境中运行时，可能不存在恶意行为，而且代码是正确编写的。在这种情况下，在像谷歌或 Amazon 这样的大公司管理的数据中心中，容错计算被用来防御无论是网络链接中断，还是服务器机架的电源故障，亦或是是硬盘报废的各种错误是很常见的。

10.1.2 时钟（Clocks）

然而，在 Leslie Lamport 在他的“时间、时钟和分布式系统中的事件排序”中引入了分布式一致性的问题时，才正式出现。在那项工作中，Lamport 演示了如何从基于通信的因果关系的定义中产生局部的事件排序。也就是说，发生在在通信事件之间的并发进程中的事件，实际上发生在同一时间，因为它们不能相互影响。因此，一个逻辑时钟系统可以根据个人的顺序进程和消息在接收前发送的事实来定义。然后，可以通过在局部排序之上分配任意但一致的总排序来完全排序事件，例如，通过为系统中的每个进程分配一个索引和排序事件，这些事件发生在同一逻辑时间内，由它们所发生的进程的索引进行。该算法相当简单，要求每个进程从彼此的进程中获得消息，以确定事件的顺序。

Lamport 的工作确立了时间作为设计容错分布式系统的主要障碍，因为在地理位置上同步时钟需要信息的通信，而这些信息最终受到光的速度的限制。这个

问题的形成与现代物理学的相对论有密切的联系，其中参考系和观察者相关，而光的速度限制了信息传播。

10.1.3 FLP

如第 2 章所述，设计一致性算法的主要因素之一是关于网络和/或处理器同步的假设。同步网络是指在固定的已知时间内传递消息的网络。类似地，同步处理器的时钟是保持在某些固定的、已知的相互运转数目之间的。在早期的一致性研究中，尽管在异步和宕机故障之间的密切关系是很明显的，这一区别并没有被很好的定义。Lamport 最初的协商一致算法能够在异步环境中运行，只要所有的消息最终都可以从每个进程中传递。然而，该算法显然不是容错的，因为仅仅一个进程的失败就可以永久地终止该算法。

由 Fischer、Lynch 和证明了在异步环境中即使有个单独的进程失败，决定性分布式一致性的都是不可行的 Patterson，正式地提出了在一个单独故障阻挠的一致性协议基础上的直觉。结果并不适用于同步上下文，因为关于网络同步的假设允许处理器使用超时来检测故障，这样如果某个进程在给定的时间内没有响应，则假定它已经崩溃。此外，这个结果只适用于决定性一致性协议，因为它的证明依赖于网络从一个二价状态，即不是所有的进程都持有相同的值，到一种单价的状态，即它们都持有相同的值，变成决定性的时刻。由于转换点是时间上的一个决定性的点，如果一个单独的进程恰好在那个时刻崩溃，那么一致性就会失败。

10.1.4 常见币（Common Coin）

FLP 的结果成为了分布式系统科学家的警钟，在新兴领域的心脏地带建立了一个明显的不可能的结果。之后，这种方法将推广到更多不可能的结果，并将花费大量的学术研究来放松同步或决定论假设，以推导出绕过结果的算法。

特别地，在简短的说明中，Ben Or 演示了包含简单量的非确定性的算法可以如何规避 FLP 结果。该算法容忍异步环境中多达一半进程的故障。从本质上说，为了在单个比特的值上达成一致，如果一个进程没有从同一个价值的多数获得投票，那么它会随机地改变它下一轮投票的值。随着每个人都在改变价值观，最终一半以上都会投同样值的票。由于与公有地翻转硬币以获得一个共享的值的过程相似，这种方法被认为是一种常见币（common coin）。

Ben Or 的 common coin 的问题在于，在异步情况下，算法需要在验证者数量上以指数形式来表示。这被 Rabin 的后续行动迅速纠正，Rabin 表示，如 Shamir 首创的，common coin 可以用秘密共享的方式构建。该方法对于 BFT 也是有用的，并且在后面的章节中将更详细地讨论。

10.1.5 交易处理（Transaction Processing）

同步开发容错一致性算法是第一个商业数据库系统的出现形式。虽然他们没有首先使用正在开发的一致性协议，但他们是在分布式计算和并发的不断增长的工作体之上构建的。特别是 Jim Gray 的开创性工作，他将这个术语，交易（transaction），作为一个数据库系统中的原子工作单元引入。也就是说，一个交易要么完全应用，要么根本不应用。

Grey 还介绍了其他现代数据库的经典特征，如原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）的原则，也是交易

（transaction）概念的一部分和包装，以及为了在被执行前把交易日志记录到磁盘上以便从在交易执行时才是的错误中恢复而使用的 write-ahead-logs。

在分布式数据库设置中，这种处理交易、原子性和一致性的工作导致了一系列以原子提交（atomic commit）的概念为中心的数据库复制的方法，其中交易在所有机器上都以原子方式复制。这些方法被称为两阶段提交（two-phase-commit），以及它的非区块替代，三阶段提交（three-phase-commit）。

两阶段提交和三阶段提交协议都只在同步设置中工作，在那里可以检测到宕机故障，并利用协调进程作为协议的领导者。

10.1.6 广播协议（Broadcast Protocols）

第 2 章介绍了两个最重要的广播协议，RBC 和 ABC。在[27]中提供了对该问题解决方案的分类和调查。

10.2 拜占庭（Byzantine）

许多容错协议只关注宕机故障，因为它们是最常见的，而对于潜在的任意的问题，包括恶意软件的行为，却很少关注。这个更普遍的问题被称为拜占庭容错。

10.2.1 拜占庭将军（Byzantine Generals）

Lamport 在[78]中引入了拜占庭容错问题，但是在后来的论文中，因为面对拜占庭军队协调攻击敌方城市的问题做出分析，才给它起了名字。军队由多个部门组成，每个部门由一个将军领导。将军之间的交流只能通过信使进行。如果一名或几名将军是叛徒，那么将军如何达成共同的行动计划？

原始论文提供了能够容忍 f 个拜占庭的故障的第一种证明，系统必须具有至少 3f + 1 个节点。这一结果基础上的直觉如图 2.2 所示，并在第 2 章和第 3 章中进行了讨论。这两篇论文中提供了许多算法作为问题的第一个解决方案，尽管它们被设计为仅在同步情况下工作，其中可以检测到不存在消息。

10.2.2 随机一致性（Randomized Consensus）

异步拜占庭式一致性以 Ben Or 和 Rabin 介绍的 common coins 的形式出现了第一个解决方案。然而，这两种解决方案都不能达到 3f +1 个机器中有 f 个故障的最佳拜占庭容错能力。Ben Or 的解决方案需要 5f + 1 台机器，而 Rabin 需要 10f + 1 台机器。该解决方案被迭代地改进，以达到以低的开销优化拜占庭。

10.2.3 局部同步（Partial Synchrony）

BFT 的下一个重大进展是所谓的 DLS 一致性算法的形式，是以作者 Dwork，Lynch 和 Stockmeyer 命名的。DLS 的创新是定义称为局部同步（partial synchrony）的在同步和异步之间的中间地带（middle ground）。局部同步的秘诀是假设以下之一：

- 消息保证在一定固定但未知的时间内传送。
- 从未来的某未知的时间开始，消息保证在一段已知的时间内交付。

DLS 算法通过一系列回合进行，每一回合分为尝试（trying）和锁定释放（lock-release）阶段。每一回合都有一个相应的提议者，如果他们认为提议者会提出这个值，进程可以在这一轮锁定这个值。一回合开始于进程广播它们可以接受的值。如果提议者从至少 N – f 个可接受该值的进程中收到来信，它即可以提

出这个值。任何接收提议值的进程都应该锁定它，并发送一个确认消息表示它已经这样做了。如果提议者接收到 f + 1 个进程的确认，它就会提交该值。

对基本协议的变化进行了不同的假设组合的讨论，并提供了许多证明。然而，尽管它取得了成功，但 DLS 算法从未被广泛用于 BFT。Tendermint 最初的设计是基于 DLS 的，更确切的说采用的是假设一个局部同步网络但是全部同步处理器时钟的版本。实际上，由于使用了像网络时间协议 Network Time Protocol (NTP)这样的协议，同步时钟可能是一个合理的假设。然而，NTP 很容易受到一系列攻击，而且假设同步时钟的协议从宕机故障中恢复得很慢。在 2015 年夏天，核心的 Tendermint 一致性协议被重新设计为更加完全的异步，如第 3 章所描述的，因此它变得更加接近于另一种 BFT 算法，即实用拜占庭容错 Practical Byzantine Fault Tolerance (PBFT)。

10.2.4 PBFT

PBFT 在 1999 年被引入，被广泛认为是第一个实用的 BFT 算法，适合在异步网络中使用，尽管它实际上会产生微弱的可能会被一个谨慎的对手所违背的同步假设。PBFT 通过一系列视图进行，每个视图都有一个提议者，称为主视图（primary），它是按回合顺序选择的。主视图从客户端接收请求，分配它们一个序列号，然后广播一个签署过的预准备（pre-prepare）消息到其他包含视图和序列号的进程。如果副本还没有接受相同的视图和序列号，它们就接受预准备消息，假设消息是针对当前视图并由正确的主视图签名的。

一旦一个 pre-prepare 被接受，一个副本广播一个签署过的准备消息（prepare message）。当一个副本收到一个给定客户端的具有相同视图和序列号的请求的 2f 个 prepare messages 时，这个副本就可以说是准备好的了（prepared）。预准备（pre-prepare）和准备（prepare）的组合根据其序列号确保单个视图中的请求的总序性（total order）。一旦一个副本已经准备好（prepared），它会广播一个签署过的提交消息（commit message），只要它被正确签名并且视图是正确的，它就被接受。当一个副本接受提交消息时，它会针对状态机运行客户端请求，并将结果返回给客户端。

PBFT 采用了一种额外的机制，以方便在主视图故障时帮助视图更改。副本维持一个超时，每次接收新客户端请求时重新启动，并在收到该请求的 pre-prepare 时终止。如果没有收到 pre-prepare，则副本超时，并触发视图更改协议。视图的变化是微妙的，而且有些复杂，因为它需要视图应该进行更改的一致性，并且自从上次提交的所有客户端请求都必须被引入新的视图。

Tendermint 通过使用区块并更换每个区块的提议者来回避了这些问题，允许使用与提交提议的区块相同的机制来跳过一个提议者。此外，区块的使用允许 Tendermint 在下一个区块中包含上一个区块预提交的消息集，消除了显式提交消息的需要。

10.2.5 BFT 改进

自从 PBFT 被发表以来，许多改进方案已经被提出来。其中一些关注在所谓的乐观执行（optimistic execution）上，这样交易可以在它们被提交以前执行以便为客户端提供低延迟和乐观的回复。这些方法的问题在于，管理不一致性的责任被转移到客户端，然而可能他们在一开始使用一致一致性协议（consistent consensus protocol）的原因就是为了避免这种责任。或者，在低故障情况下，这

可能是一种有用的方法。这种现象在比特币中被称为 zero-conf 交易，并受到广泛的警告，鉴于接受交易的不安全性已经在它们之前提交了足够的工作。

其他人则专注于同时运行独立交易的可能性，以实现更高的吞吐量。这是在区块链社区中开始研究的方法，尤其是以太坊，以便产生可扩展的区块链结构。

10.3 非拜占庭

与 BFT 算法并行，出现了一些非 BFT 算法，并且已经建立了一些重要的高可用性互联网服务。

10.3.1 Paxos

在一致性科学中，人们通常认为，只有一种一致性算法，即 Paxos。这一方面说明了 Paxos 算法对该领域的重要性，另一方面是对共识一致性协议的普遍基础的反射，这在每种情况下都是“类似 Paxos（Paxos-like）”的。

Lamport 在九十年代初期引入了 Paxos，尽管那篇文章直到将近十年以后才被接受发表。许多人已经指出，该算法实际上与上世纪八十年代末期出版的 Viewstamped Replication 极为相似，并且两者都代表了相同协议的独立发现。

这些协议与之前的 PBFT 非常相似，但只需要 2f + 1 台机器就能够容忍 f 个故障，如同它们不是 BFT。另一个类似的协议，Zookeeper 原子广播协议（ZAB）是为 Apache Zookeeper 分布式键值存储开发的。在[99]中，每个算法的相似性和差异性都有所阐释。

10.3.2 Raft

引进 Raft 后，非 BFT 一致性科学得到了重大改进，这一设计是从根本上可以理解的，通过用户调查证明了它甚至比 Paxos 更容易理解。

Raft 在精神上类似于 Paxos 和 Viewstamped Replication，但它强调复制一个交易日志，而不是一个单个的比特，并引入随机化来进行更有效的领导者选举。此外，Raft 的安全性保障已经使用 Coq 证明助手（Coq proof assistant）和一个在 Coq 上面构建的以正式验证分布式系统的框架，Verdi，正式地证明过了。Verdi 将如何与基于过程演算的方法进行比较还有待观察。

10.4 区块链（Blockchain）

本文的主要动机是介绍以比特币形式出现的区块链技术，和从此看到的许多迭代。 直到最近才有人成功地将区块链置于经典一致性科学的背景下。

10.4.1 比特币（Bitcoin）

在[71]中介绍的，比特币是第一个区块链。它通过巧妙地利用经济学在公开的对抗设置中解决了原子广播问题。特别地，交易顺序是由解决部分哈希冲突的人提出的，其中被散列的数据是交易的区块。由于计算部分哈希冲突是昂贵的，需要在大空间中进行强力搜索，所以这种努力通过每个区块的发行货币，比特币，进行补贴。该协议已经非常成功，货币已经达到十亿美元市值，原有的克隆产品市值也达数百万。

然而，比特币并非没有问题。一些设计缺陷使应用程序开发人员很麻烦也很难使用 它。此外，一些学术著作也阐明了协议中的激励不相容问题，削弱了人们对该协议的安全性的普遍假设。

很多方法被提出以改善比特币，包括那些对部分哈希碰撞函数性质的改变，那些为了改善许多经济学和潜在性能的特征而对协议中领导者选举的性质的改变以及那些旨在实现可伸缩性对协议进行的讨论。

10.4.2 以太坊（Ethereum）

以太坊由 Vitalik Buterin 引出，作为 Bitcoin 之后的一种加密货币的扩散的解决方案，具有不同的功能。以太坊寻求更纯粹的任务：没有功能。 相反，以太坊提供了一个图灵完整虚拟机（Turing complete virtual machine），即以太坊虚拟机 Ethereum Virtual Machine (EVM)，用于在一致性之上执行交易，并为用户上传代码到可以在将来的交易处理中执行的 EVM 上提供了一种手段。所谓的智能合同（smart contracts）使用强大的密码学和 BFT 复制提供了在公共设置中自动执行代码的承诺。以太坊项目在迄今为止规模最大的众基金（crowd-funds）之一中取得了成功，超过了 1800 万美元，而它的本地 token（用于支付交易执行和代码上传的费用）的市值已经达到 10 亿美元。

Ethereum 目前使用一种叫做贪婪最终观察子树 Greedy Heaviest Observed Sub Tree (GHOST)的 Proof-of-Work 的修改形式，但正计划转向围绕 Proof of Stake 建模的更安全的经济一致性算法。

10.4.3 Proof-of-Stake

Proof-of-Stake (PoS)最开始作为在 PPCoin 中使用 Proo-of-Work 的一种替代方案被提出来。在 PoS，提议由那些可以证明网络中硬币奖金的所有权的人提出并投票。虽然消除了 PoW 的过高成本，但对于 PoS 的朴素实现很容易受到所谓的“nothing-at-stake”的攻击，在给定的高度，验证者可以在多个区块上提议和投票，从而导致了严重的安全性的违背，没有任何激励以收敛。虽然朴素 PoS 的问题众所周知，但许多流行的加密货币仍在使用它。

Nothing-at-stake 问题可以通过一个名为 slasher 的机制进行纠正，即验证者必须交一个安全保证金，以便有资格验证块，这样可以在发现验证者提议或投票支持冲突区块时削减它的押金。Tendermint 是这种方法的第一个实现，尽管其他的 BFT 算法也可以发挥作用。

10.4.4 超级账本（HyperLedger）

比特币、以太坊和其他加密货币的成功激发了社会越来越多元化的领域，包括监管机构、银行家、企业高管、审计人员、客户经理、物流师等等。尤其是，Linux 基金会（Linux Foundation）最近发起的一个项目，由 IBM 和一家名为“数字资产控股”（Digital Asset Holdings, DAH）的新成立的以区块链为基础的新公司，旨在为工业应用提供一个统一的区块链架构。这个项目被称为“超级账本”（HyperLedger），以一个同名的公司命名，该公司提供了一个基于 PBFT 的区块链的基本实现，已经被 DAH 收购。

对 HyperLedger 倡议的两项贡献尤为相关。第一个是 JP Morgan 团队运作的 Juno 和 Hopper 的合并。Juno 是 Tangaroa（一个 BFT 版本的 Raft）的实现。Hopper 是一个新的虚拟机设计，基于线性逻辑（linear logic）和依赖类型系统（dependent type system），它的目标是为对于得出和证明关于系统状态的陈述或者合同行为而配备的一个正式的逻辑的智能合同系统提供一个执行环境。Juno 和 Hopper 都是用 Haskell 写的。

另一个项目是 IBM 的 OpenBlockchain，一个由 Go 编写的基于 PBFT 的区块链，它的应用程序状态支持任意 docker 容器的部署。由于任意 docker 容器可能包含非决定性，它们的 PBFT 实现用在可能的非决定性执行的情况下保护安全性的额外步骤来修改。

IBM 的另一个相关贡献是最近的一篇评论文章，类似于本章。

10.4.5 HoneyBadgerBFT

尽管在异步环境中运行良好，所有与 Paxos-like 的一致性协议，包括 Raft、PBFT 和 Tendermint，都不是完全异步的。这是因为每个协议都在协议的某个地方使用超时，通常是为了检测故障领导者。另一方面，像 common coin 这样的随机一致性协议提供了在完全异步环境中，没有超时的解决方案。

所有一致性协议都以某种方式依赖于最终消息的传递。异步的假设简单地指出，没有消息被传递的上限。大多数时候，网络行为是同步的，在某种意义上，大多数消息是在一定范围内传递的。完全异步协议与有超时协议之间的差异在于，当网络同步运行时，异步协议总能取得进展。这一点在[67]中清楚地显示，其中引入了 HoneyBadgerBFT，第一个完全异步区块链设计，基于 common coin 一致性。

对网络进行任意控制的对手以及一次可以使得任何一个节点崩溃的能力可能导致 PBFT 任意长时间停止。这可以通过在网络同步时崩溃当前的主视图/提议者/领导者，并将其恢复为异步的时间段来完成。网络仍然最终传递消息，具有一些平均的同步性，但是精确的时序可以停止所有的系统进程。在 PBFT 的直接实验在[67]中，并且与 Tendermint 工作类似。

HoneyBadgerBFT 利用一系列加密技术，包括秘密共享，擦除编码和阈值签名，以设计出高性能异步 BFT 一致性协议来克服这种困难，因为它是完全无领导的而导致的没有任何同步的假设。然而，它需要一个受信任的经销商（dealer）进行初始设置和验证者变更，并且它依赖于尚未经得起时间考验的某些问题的难度的相对新的加密假设。

10.5 结论

Tendermint 出现并补充了丰富的一致性科学的历史，涵盖了同步和容错假设的范围。 区块链和 Raft 的发明重新启动了一致性的研究，并产生了新一代通过互联网协调的协议和软件。

Chapter 11

总结

拜占庭容错一致性提供了一个丰富的基础，用来构建不依赖于集中式、可信任方的以及可能被社会用来管理社会经济基础设施的关键组成部分的服务。本文中提出的 Tendermint，旨在满足这些系统的需求，并以可理解的安全性和易于高性能的方式以及允许任意系统按一致性协议和最少的无关事进行交易来进行设计。

在部署分布式一致性系统时，需要谨慎考虑，尤其是在没有中央授权的情况下，在危机发生时调解潜在的争端并重置系统。Tendermint 试图通过明确的治理模块和问责保障制度来解决这些问题，使 Tendermint 部署融入到现代法律和经济基础设施中。

现在仍然有许多工作要做。这其中包含算法保证的正式验证，性能优化，以及架构变更以使得系统用额外的机器增加容量。而且，当然，还有许多许多 TMSP 的应用要构建。

我们希望本文更好的解释了在分布式一致性算法和区块链架构中的问题，并且启发其他人建立一些更好的东西。