智能合约技术黄皮书

Version 1.0

导论

智能合约（Smart contract）是一种旨在以信息化方式传播、验证或执行合同的计算机协议。智能合约允许在没有第三方的情况下进行可信交易，这些交易可追踪且不可逆转。智能合约概念于 1994 年由 Nick Szabo 首次提出，但在区块链技术出现之前，由于缺少可信的执行环境，智能合约并没有被广泛应用。随着区块链的普及，智能合约技术得到飞速发展。像以太坊，EOS 这些优秀的项目极大的促进了对区块链和智能合约底层技术的研究，使之朝着大规模商业化应用的方向推进。

但同时我们也清楚的认识到，现有的区块链系统和智能合约的设计机制还存在以下的几大缺陷使其还难以支撑大规模的商业化落地：1. 开发者不友好，e.g. 全新的开发语言无法复用广大传统开发者的原有开发技能，开发工具/SDK 的缺失和不完善导致开发效率的低下；2.智能合约的交互和表能能力不够丰富，e.g. 现有的智能合约的更新机制不够灵活，导致合约的开发迭代和部署不够敏捷；合约的运行周期和共识机制的强耦合对合约运行的时长有了极大的限制，实质上削弱了所宣称的图灵完备性的实用性；3.区块链系统运行智能合约的效率低下，e.g. 区块链的强安全性和去中心化特性都是通过大规模的计算和存储冗余来保证的，这同时也导致了整个网络系统的智能合约运行效率的低下。

在 Ultrain，我们对上述问题做了长久深入的思考和探索，在这个过程中积累了宝贵的经验并逐步形成了我们自己的创新解决方案。后面我们将在“**丰富灵活的智能合约交互形式**”“**动态自适应分片技术**”以及“**开发者友好的智能合约开发环境**”这几个章节一一展开来介绍我们的这些创新方案，从而形成一个整体的对开发者友好的，有丰富表达能力的，高效运行且安全的区块链和智能合约系统，以此来支撑我们 Ultrain 对于“构建人人可编程的商业社会”的愿景。

丰富灵活的智能合约交互形式

1. 合约的运行时长管理

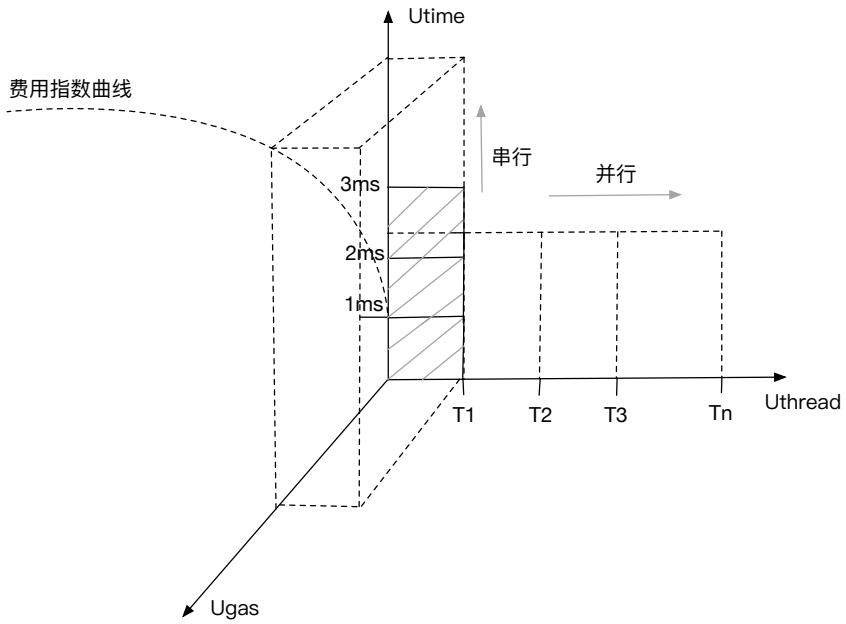
在讲智能合约运行时长之前，我们需要介绍一下交易的现状，因为区块链网络中智能合约是由交易驱动的，同时智能合约的运行结果关系到交易的确认。

首先看商业技术比较成熟的以太坊，在以太坊中交易类型只有一种，即普通的交易，其特性就是没有交易状态管理，交易驱动的合约需要同步运行，且合约运行时长受限于交易中的 Gas 数量，如果合约不能在 Gas 消耗完之前完成执行，节点会强行终止合约执行，并回滚合约运行的状态，导致交易确认失败。Gas 机制可以认为是对于图灵完备的智能合约和周期性共识算法融合时所面临的停机问题（Halting Problem）的一个 workaround.

我们再来看下 EOS 的方案. EOS 和以太坊的一个显著不同点是采用设置合约运行时间上限来解决停机问题，从而保证合约运行能够在共识周期内完成。另外一个不同点是 EOS 引入了延时交易（Deferred Transaction）的功能，允许当前交易/合约触发后续的交易/合约在

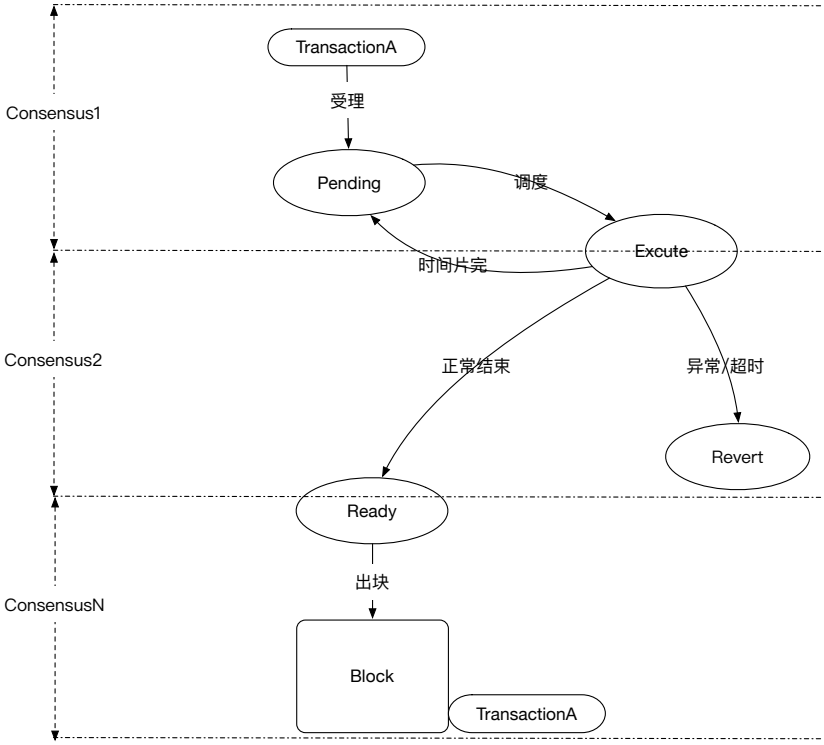
未来的共识周期内被调度。从而在一定程度上满足了合约长期执行的需求。但这样的设计最终所引导的编程范式极大的增加了开发者在合约设计上的负担，e.g. 长时间运行的合约必须被拆分成若干个可以在单个共识周期内能完成的子合约/功能；而且另一个灾难性的特性是 EOS 并不保证后续合约必定会被执行。

我们看到不论是 Gas 机制还是 Deferred Transaction 机制，试图解决的核心问题都是在智能合约有跨共识周期运行的需求的情况下的停机问题，但这两种方案都不令人满意。对此，我们提出如下的解决方案. 我们的 Ultrain 区块链网络中分为两种类型的交易，一种是弱时长交易（Short-Range Transaction），这个也是默认的交易类型，交易发起方对这种交易所触发的合约预计是能够比较快的执行完成的，系统对这种交易限制 1ms 运行时长，如果超过这个运行时间合约会被标记交易失败，系统状态会发生回滚。这种类型的交易因为可以做到在一个共识周期内完成，我们可以采用单机并行处理或者网络分片的方法进行加速，具体方案详见后续的“动态自适应分片技术”章节；另一种是强时长交易（Long-Range Transaction），交易发起方对这种交易所触发的合约预期是会执行比较复杂的应用逻辑，是无法保证在极短时间内完成的。对于这种实际的需求，我们在机制上允许这种不设时长上限的交易合约存在，只是在经济成本上对其进行管控。我们通过对强时长交易驱动的智能合约做有偿管理，建立合约运行的时长和 UGas 消耗数量呈指数关系，从而确保不会有人通过强时长交易来作恶（e.g. 恶意的长期占用系统计算资源），如下图描述 Ultrain 执行交易的合约运行的三维关系：

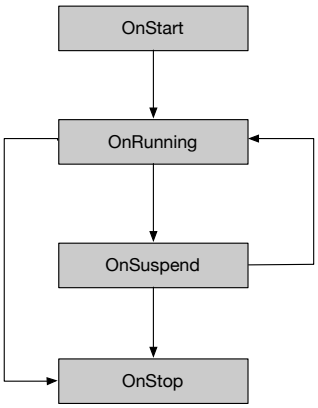


前面我们设计了合约的运行管理模型，保证这样的模型能在 Ultrain 这样完全开放的公有链网络中稳定高效的运行，我们必须解决一个世界性难题，即怎样在每次共识出块前保证全网超过 2/3 的节点能达成世界状态最终一致性。首先需要确保合约运行的基本要素确定性，如时间，随机数和数据源等，对于这些方面我们尽量吸收借鉴现有一些优秀项目成熟的技术方案，如时间的确定，我们只提供了基于区块时间戳的系统调用，可以将整个区块链看成一个时间戳服务器，并取得任意一个区块被构造时的时间戳，这样能完全保证分布式节点调用返回的结果都会一致。有了运行要素完全一致以及运行环境的基本一致，我们核心工作就是通过对（弱时长和强时长混杂的）交易和合约运行管理算法的创新，来保证在共识过程复杂合约能达成状态的一致。我们对强时长交易引入如下图中的状态管理，

能很好的做到合约运行周期和共识周期的解耦（i.e.允许跨共识周期的合约执行和状态管理），从而保证合约能通过自己生命周期的管理来达到在分布式系统中的状态一致性。



下面我们将展开讲合约怎样通过自己生命周期的管理来达到在分布式系统中的状态一致性，我们将在合约运行管理中引入如下四个生命周期，分别为 OnStart，OnRunning，OnSuspend 以及 OnStop。



假设我们有这样一笔强时长交易 TransactionA，其完整的执行过程是 TransactionA 受理时的状态为 Pending，并且加入到待处理的强时长交易集合，即 TransactionA $\in$ LRT（LRT = $\{ \mathrm{lrt}_1, \mathrm{lrt}_2, \mathrm{lrt}_3 \ldots \mathrm{lrt}_N \}$），在每轮共识周期开始前一小段时间，系统会先执行 LRT 集合中的强时长交易（通过 time-sharing 的方式来保证集合中的交易都有被执行的机会），在执行过程中系统会加载他们驱动的合约，合约启动进入生命周期 OnStart 执行。OnStart 的主要工作是锁定 TransactionA 关联账户的世界状态，并设置 TransactionA 到 Excute 状态，完成这些准备工作后合约进入生命周期 OnRunning 执行，OnRunning 主要工作是执行交易相关的业务逻辑，由于 OnRunning 执行时长不定，可能一个交易的时间片执行不完，遇到这种情况合约将进入生命周期 OnSuspend; OnSuspend 主要设置执行断点并进入 sleep，当进

入到下一个强时长交易的执行时间片的时候，合约恢复断点又进入 OnRunning 周期，开始剩余逻辑的执行，直到成功执行或 UGas 燃烧完，合约将进入 OnStop 周期；如果是正常执行完 OnStop 会设置交易状态到 ready，即等待下一个共识周期共识确认，如果由于 UGas 燃烧完导致的结束，OnStop 中将 revert 执行的所有状态，通知发送者交易 TransactionA 处理失败；OnStop 最后一步会解锁 TransactionA 关联账户的世界状态。

可以看到，强时长交易在其整个执行的生命周期中会锁住其关联账户的世界状态，即使是在等待调度的 sleep 状态下也是如此。在这期间，如果有互斥的其他合约交易，无论是弱时长交易还是强时长交易，都会被推迟执行直到关联账户的世界状态被解锁。可以看到，这种类型合约的运行和调度对系统资源的消耗是巨大的：不仅这个强时长交易本身消耗更长时间的计算资源，而且还隐含着系统调用其他潜在互斥合约的机会成本。这也是为什么我们对强时长交易设计了指数形式的合约运行的时长和 UGas 消耗数量的关系。我们认为这个模型是要优于 EOS 的 Deferred Transaction 模型，首先我们在语意上保证了长时间执行合约的可能，而相比较 EOS 不保证 Deferred Transaction 肯定被执行，会导致开发者必须要设计多种可能的合约执行结果的应对策略；在我们的有偿模型下，共识周期对上层合约开发者是透明的，他们只需要考虑如何简化代码复杂度来降低运行成本，而不像 EOS 模型需要考虑如何设计复杂度极高的合约系列来对齐每一个共识周期。

2. 合约的升级机制

在合约的升级方面，我们借鉴传统操作系统对进程的管理方式。操作系统中有的进程相对重要，而有的进程则没那么重要，所以可以通过不同的进程优先级来实现差异化管理。区块链上将迸发大规模的智能合约的应用，这些智能合约应用的场景以及合约开发者/服务提供者的诉求也是千差万别的 :有些智能合约的“合约”属性很强，合约双方信奉 “code is law”，他们希望合约一旦建立就不可修改和撤回；有些智能合约更像是传统意义上的构建在区块链这个分布式操作系统上的应用，合约提供商希望的是能够快速的进行服务迭代给到用户，他们所希望的是合约的快速升级能力；由此可见对于智能合约的特性的需求有时候是互斥的，不存在 one size fits all 的情况。 因此我们也在系统中设计了三种类别的合约来满足不同的应用场景，分别为 general, protected 和 restrict 类型合约：

- general 级别的合约，合约的 owner 可以在同一地址任意升级合约代码，这只是广义的合约，适合于弱合约强服务的应用场景。
- protected 级别的合约，合约的 owner 可以发起升级的 propose，只有全部或部分指定的用户签名才可以升级，适合于多方协作的应用场景。
- restrict 级别的合约，合约一经部署成功，合约的 owner 和其他任何人都无法修改升级，这是严格意义的智能合约。对于 restrict 级别的合约，开发工具比方说编辑器可以给出常见的安全错误的错误，而不仅仅是编译错误，适合于区块链上大部分的应用场景，这个也是默认的合约类型。

动态自适应分片技术（Dynamic-and-Adaptive Sharding）

朴素的智能合约计算模型是所有的参与节点保存所有的合约和账户状态(我们称之为世界状态)，并且串行执行所有需要验证的智能合约. 这个是一个大巧若拙的计算模型 :用极大的计算和存储冗余（但也意味着牺牲了系统的 Scalability）来保证整个系统的安全性（Security）和去中心化(Decentralization). 但这也意味着整网络丧失了可扩展性，单节点的计算性能成

为了整个网络的瓶颈. 正如当我们在解决大部分计算机科学领域的问题时都不可避免的面对各种设计上的 trade-off，e.g. 空间 vs 时间，一个好的区块链系统设计也应该争取在 Scalability, Security 和 Decentralization 这个 trilemma 中寻找一个平衡点. 我们很高兴的看到社区里有大量的资源投入到提升区块链性能 Scalability 上，比方说分片 sharding 和侧链 sidechain 技术都能够大幅度提高网络的 TPS (transactions per second). 这些技术本质上都是通过牺牲一定的 Security 和 Decentralization 来提高 Scalability. 我们也应该认识到，这些技术在过分追求高数值的 TPS 结果的时候，它并没有一个好的方法来量化这些方法在 Security 和 Decentralization 上的损失. 所以用 TPS 作为唯一的评判标准来衡量区块链系统的性能有时候是有失偏颇的. 在 Ultrain, 我们对这一领域的贡献在于 1) 首先我们设计了一个 state-of-the-art 的、对智能合约友好的、能大幅度提高系统吞吐量的计算分片 sharding 的架构 2) 在对智能合约进行分片计算的基础上，我们提出了一种动态自适应的分片设计理念，允许整个网络根据当前的网络状态来选择分片形式，以实现更灵活和动态的在 trilemma 中寻找适合当前网络状态的 anchor point; 我们称这个技术为 动态自适应分片技术 Dynamic-and-Adaptive Sharding. 接下来的章节我们会展开论述上面的两点创新.

分片的基本思想是计算机科学里面经典的分而治之 Divide-and-Conquer 算法. 我们把网络节点分成若干个子片区 shard, 每个片区执行被指定的一部分交易或者合约，所有片区并行执行然后把执行结果汇总. 这其中，计算的分片是显而易见的，但另外有一个需要考量的是存储是否分片---即每个分片是否都保存完整的世界状态还只是各自独立保存一部分世界状态. 在我们目前的设计中，我们不采用存储分片（即所有节点都保存完整的世界状态），因为存储分片带来的不可避免的跨分片通讯的额外开销即使在简单的交易场景下都还没有被验证可行，而在跨存储分片的智能合约执行场景下，系统复杂度更是急剧上升，我们认为社区现有的研究进展还不存在能够孕育出实用的跨存储分片智能合约调用方案的土壤. 所以以下的讨论都只限于计算分片.

我们先来看下简单交易场景下的分片. 我们的分片设计的核心是希望交易分片的结果不存在跨分片通讯，这样每个交易都可以在被分到的片区独立的被验证. 假设当前网络上所有的可交易账号集合为 A，有一批 C 个等待验证的交易 T，其中

$$T_i = (s_i, r_i)\ where\ i \in C, s_i \in A, r_i \in A$$

表示一个账户 s_i 对账户 r_i 发起的简单转账交易，然后我们定义一个无向图

$$G = (V, E),\ where\ V = s\ \cup r,\ E = \{(s_i\ r_i) : i = 1,2 \dots C\}$$

以 O(|V|+|E|) 的算法复杂度我们可以找到这个无向图 G 的联通分量 connected components, 这个联通分量集合可以作为我们的分片算法的基础. 只要我们能保证，落在同一联通分量里面的所有交易都在一个片区里面被验证执行，那就不存在跨片区通信带来的额外复杂度和通信开销. 具体的基于联通分量的分片算法我们稍后再做论述.

接下来我们看下当系统中存在智能合约的情况下的计算分片设计. 不同于简单交易的场景在处理交易前我们就已经知道这个交易会涉及的（两个）账号数据，智能合约的动态调用机制决定了合约执行过程中涉及的数据是运行时才能确定的，可能被调用的合约数量也是运行时确定的，因此我们无法在运行前执行类似上面所描述的针对简单交易所设计的避免跨分片通信的分片算法. 而允许跨分片通信的智能合约执行的算法又会带来极大的系统复杂度和运行时开销. 为了解决这个问题，一种直观的想法是引入开发者的干预，让开发者在开发智能合约的时候就静态的申明这个智能合约可能会调用的其他智能合约的地址，这样通过代码的静态分析，我们就可以在执行前确定这个智能合约所有可能直接或者间接调

用的其他的智能合约（通过递归的分析），我们称之为智能合约的调用集合. 这个给我们提供了对智能合约进行计算分片的基础. 这个设计的一个主要缺陷是牺牲了智能合约的动态更新能力，假设我们有这么一个合约调用链 ContractA --> ContractB --> …. --> ContractY --> ContractZ, 如果我们希望更新 ContractZ 的代码，重新部署 ContractZ 会导致他的部署地址发生了变化（这里假设我们遵循的模型是如果合约代码发生变化，重新部署会改变合约地址，即上一章所描述的 restrict 合约类型；general 类型合约的情况比较简单，这里就不做专门论述了），然后这个更新会造成一系列的链式反应，我们必须要重新发布和部署所有的调用链上的合约. 试想一下某个广泛使用的合约公共库发生了更新，所有依赖于它的应用合约都需要重新部署，这个在大规模的工程实践中是不能接受的. 我们希望的方式，而且也是现在被社区和开发者广泛接受的智能合约更新方式是: ContractY 中保存了指向 ContractZ 的地址的变量，当 ContractZ 由于重新部署而导致地址改变的时候，只要更新 ContractY 中指向 ContractZ 的地址的变量到新的地址就可以了，调用链上的其他合约比方说 ContractA 和 ContractB 都不受影响.

对此，我们的设计方案是: 通过在智能合约编程语言中对代表智能合约的数据结构做特殊的读写监控，能够在保证子合约的更新机制不受影响的情况下，仍然能在任一时间点把合约的调用集合静态的分析出来. 为了方便论述，我们首先做如下一些定义:

1. 在我们的智能合约编程语言中假设用 ContractInstance 类来表示合约对象.
2. 合约调用有且仅有唯一的语法入口，e.g. Call ContractInstance, 这个操作会调用 ContractInstance 当前值所代表的地址的合约.
3. 如果一个合约的某次执行对任意的 ContractInstance 变量有创建/销毁/赋值更新操作，我们称这样的操作为 TC-DIRTY 操作，否则我们称之为 TC-CLEAN 操作.
4. 对于触发合约运行的合约交易，我们有一个标志位，如果调用者预计这个合约交易会包含 TC-DIRTY 操作，调用者设置这个标志位为 TC-SHOULD-DIRTY, 否则调用者设置这个标志位为 TC-SHOULD-CLEAN;
5. 当且仅当标记为 TC-SHOULD-CLEAN 的合约调用交易在执行过程中触发了 TC-DIRTY 操作，我们称之为发生了 TC-BREAK 状况.
6. 假设当前时间有一批 N 个等待触发调用的智能合约集合 $T = \{t_1, t_2 \dots t_N\}$，其中标记为 TC-SHOULD-CLEAN 的个数为 NC的合约集合为 $TC = \{tc_1, tc_2 \dots tc_{NC}\}$,标记为 TC-SHOULD-DIRTY 的个数为 ND的合约集合为 $TD = \{td_1, td_2 \dots td_{ND}\}$
7. 我们定义映射 $F_LIST_ONE(t)$为合约 t 所包含的所有 ContractInstance 所指向的合约的集合 (且包含 t 本身)
8. 我们定义映射$F_LIST_ALL(t)$ 为:
   8.1 $TEMP = F\_LIST\_ONE(t)$
   8.2 $TEMP\_EXPAND = \{F\_LIST\_ONE(tmp_1) \cup F\_LIST\_ONE(tmp_2) \dots : tmp_i \in TEMP\}$
   8.3 如果 $TEMP == TEMP\_EXPAND$，则$F_LIST_ALL(t) = TEMP$，退出流程
   8.4 $TEMP = TEMP\_EXPAND$，转到 8.2

通俗的讲，$F\_LIST\_ALL(t)$ 表示了 TC-SHOULD-CLEAN 合约 t 在不发生 TC-BREAK 的执行过程中所有可能直接或间接调用的合约的总集合.

接下来我们描述下智能合约可做计算分片的条件:

1. 对任意两个 TC-SHOULD-CLEAN 合约 tc_i 和 tc_j，如果 $F\_LIST\_ALL(tc_i)$ 和 $F\_LIST\_ALL(tc_j)$ 没有交集，则这两个合约可以被分到不同的片区并行执行，因为他们读写的是完全不同的合约存储空间.
2. 我们定义一个无向图$G = (V, E)$，其中

$$V = \{F\_LIST\_ALL(\ tc_1) \cup F\_LIST\_ALL(\ tc_2) \cup \ \dots \cup F\_LIST\_ALL(\ tc_{NC})\}$$
$$E_{tc_i} = \{all\ edges\ of\ Complete\ Graph\ of\ F\_LIST\_ALL(\ tc_i)\}$$
$$E = \{E_{tc_1} \cup E_{tc_2} \cup \dots \cup E_{tc_{NC}}\}$$

同样的，以 O(|V| + |E|) 的算法复杂度我们可以找到这个无向图 G 的联通分量 connected components, 这个联通分量集合可以作为我们的分片算法的基础. 只要我们能保证，落在同一个联通分量中所代表的智能合约都在同一个片区里面被执行，那就不存在跨片区通信带来的额外复杂度和通信开销，且不同片区可以并行执行，因为不同片区的智能合约读写完全不同的合约存储空间.

3. 对于标记为 TS-SHOULD-DIRTY 的智能合约 t，因为它会动态的变更$F_LIST_ALL(t)$的集合元素，这些交易不能简单的被分片处理. 一个简单的方法是把这些交易独立出来，在统一的时间点做集中的非分片处理. 因为 TS-SHOULD-DIRTY 意味着合约代码的更新，它的发生频率应该非常低的，所以对这些交易做集中处理不会对网络的长期性能有大的影响.
4. 一个特殊情况是在分片计算中发生了 TC-BREAK 状况. 这种情况的发生会导致分片假设不成立，系统回退到顺序执行模型. 我们应该通过静态代码检查工具，经济上的惩罚措施等来发现这些情况并协助开发者来减少这种情况的发生.

到目前为止，不管是对于简单交易的场景，还是涉及智能合约执行的复杂交易场景，我们都能够得到基于图算法的联通分量集合，对于落在同一个联通分量里面的简单交易/合约执行，因为可能涉及相同的世界状态的读写，必须要顺序执行才能保证最终世界状态的结果的唯一性；对于落在不同联通分量里面的交易，则可以并行的执行而结果互相不干扰. 联通分量是我们分而治之的基础，假设我们把落在同一个联通分量里面的交易组合成一个交易组，所有的交易组集合我们称之为

$$TG = \{TG_1, TG_2, \dots, TG_m\},$$

理论上限是我们可以做 m 个并行计算. 接下来我们讨论单节点并行执行和多节点分片执行两种情况.

对于单节点并行的场景，我们考虑的问题是如何利用单节点的硬件能力(e.g. multiple-core, multiple-hyperthread) 做最大程度的并行计算. 假设我们的硬件的可并行计算单元数量为 N, 我们记为

$$\text{CORE} = \{CORE_1, CORE_2 \dots CORE_N\}$$

我们考虑如何把 m 个任务分配到 N 个计算单元上执行. 对于简单交易，因为每个TG_i的时间复杂度和交易数量成正比，如果我们记分配到$CORE_i$上的所有交易的复杂度之和为$S(CORE_i)$，那这个分配问题可以简化为求解 $min\frac{max}{i=1\dots N}\ S(CORE_i)$. 对于涉及智能合约执行的情况, 因为智能合约的时间复杂度是无法静态计算的, 在 N 个计算节点上进行 round-robin 算法是一种比较直观的分配方法.

对于多节点的分片 sharding 场景，我们记全网可用计算节点为

$$Node = \{Node_1, Node_2, \dots, Node_{N\_NUM}\}$$

记 shard 的数量为 $SN \in [1, N\_NUM]$. 不同于上一节里面 N 的值是正相关于单节点的硬件能力，SN 的数量选取会映射到网络中的 Security, Scalability 和 Decentralization 组成的 triangle 中的一个 anchor point，体现了当前状态下对 trilemma 的一个平衡. 我们的核心设计理念是这个平衡点 anchor point 是动态调整的，而不是静态设定的. 假设当前全网 sharding 的配置为

$$w_{ij} = \begin{cases} 1 & if\ TG_i\ is\ assigned\ to\ shard_j \\ 0 & otherwise \end{cases}$$

$$s_{ij} = \begin{cases} 1 & if Node_i\ is\ assigned\ to\ shard_j \\ 0 & otherwise \end{cases}$$

分片算法的衡量标准可以记为

$$\frac{min}{N\_NUM, SN, w, s}(SecurityLoss + \alpha * ScalabilityLoss + \beta * DecentralizationLoss)$$

在最复杂的模型下$SecurityLoss$，$ScalabilityLoss$ 和 $DecentralizationLoss$ 都是(N_NUM, SN, w, s) 的函数，但是这样会让我们的模型 untractable，所以在实践中我们可以做如下的简化（我们省略了N_NUM，默认它的影响通过SN来体现）：

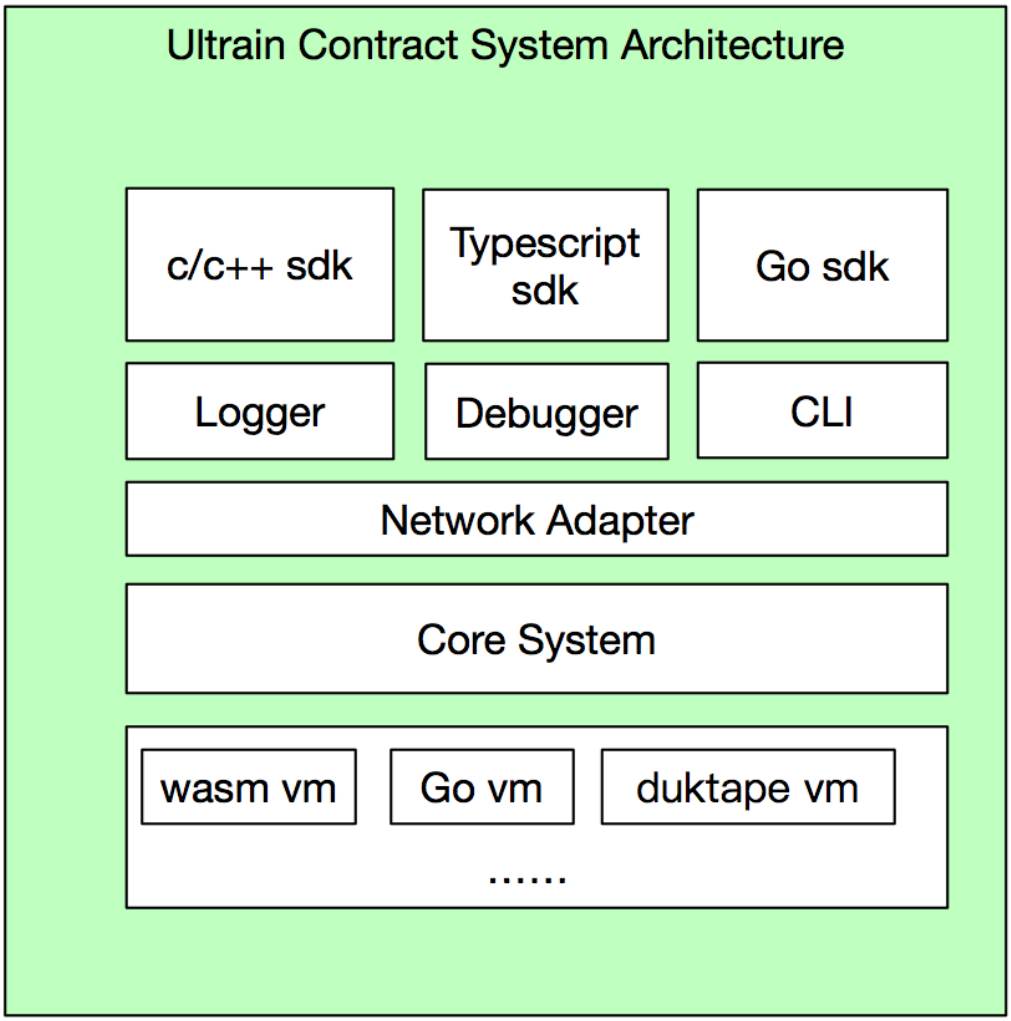
$$SecurityLoss \sim (SN)$$
$$ScalabilityLoss \sim (SN, w)$$
$$DecentralizationLoss \sim (SN, s)$$

直观上来看，我们假设 $SecurityLoss$ 受分片数量SN的影响最大，e.g.极端情况当 SN=1，我们有最强的安全性因为其实没有做分片．$ScalabilityLoss$ 主要取决于分片的数量以及每个分片里计算量的分配．$DecentralizationLoss$ 主要取决于分片的数量，以及各个计算节点如何划分到各个片区中．要指出的是，具体的$Loss$函数的设计是实现上的细节，我们设计的核心是权重项 α 和 β．一旦具体的 $Loss$ 函数确定了，权重项 α 和 β 才是调节我们对 trilemma 平衡的关键参数．我们认为，权重项 α 和 β 一定是动态的，能反应当前网络运行状态的，能对齐社区对网络的期望演进方向的．比方说，如果当前网络出现异常的拥堵，我们可能需要提高 α 的权重来增加分片属性从而提高网络的并行性能；如果当前网络出现恶意节点增多的迹象，威胁到我们共识算法中对 sharding 的安全假设，那我们可能要将 α 设小，临时切换到弱 sharding 状态(每个 shard 会有更多的节点参与)来抵御恶意节点的攻击．又比如当我们检查到网络中出现计算节点有被中心化控制的迹象发生时，我们会考虑增加 β 的权重从而得到一个更加体现去中心化特性的节点到片区的分布．α 和 β 的动态设定可以是个隶属于网络治理的投票过程，也可以是个根据网络状态指标输入的自我运行的智能合约，但是他的核心是动态变化的和适应于当前网络状态的，所以我们称之为动态自适应分片技术 Dynamic-and-Adaptive Sharding.

开发者友好的智能合约开发环境

为了实现人人可编程的愿景，Ultrian链提供一系列文档＆教程、SDK、调试工具、发布工具以及钱包、浏览器等工具。通过这些工具，可以方便快捷的开发、部署一份可以在Ultrain链上运行的智能合约。

我们的智能合约系统架构如下：



编程语言sdk的选择上，我们将支持目前最广泛使用的编程语言，智能合约可以使用C/C++、TypeScript/Javascript、GoLang进行开发。相比较于Ethereum的solidity语言，高级语言灵活性好、语法成熟度高，学习成本低，易于编写更安全、更易于审核的智能合约。高级语言对现代编程模型的支持度更好，有利于提高合约开发效率。

Logger系统，Ultrain链提供日志输出功能，但是不提供存储功能。合约开发者可以配置日志输出方式、地址，例如Post日志到指定的URL。日志不保证实时性。

Debugger系统，在debug模式下，合约执行不受共识算法限制，也不受其它诸如gas限制、运行时间片限制、系统资源(e.g. IO/RAM) 限制。

1. 为方便调试，我们提供usc-mocker工具，它可以在本地模拟运行节点环境， 并且有GUI界面和CLI两种形态。启动mock环境之后，它会自动产生模拟测试帐户，且每个帐户中默认具有预存储的测试token；

2. 我们提供一个本地钱包，通过usc wallet命令，就可以启动一个本地钱包。在本地钱包中，可以方便的进行帐户操作：导入usc-mocker中帐户、新建帐户、切换帐户，帐户间转账等操作；
3. 我们提供 MyUltrainWallet工具， 方便钱包管理，合约部署与调用，同时支持生产环境与测试环境两个版本，可以方便地在这两个环境中切换。在部署合约时，会自动计算出合约执行时需要的UGas，方便检查合约是否能被顺利执行。调用合约时，通过合约地址，可以查找到合约中的所有操作，只需要传入参数就可以分别调用相应的操作。

CLI工具。我们提供一套丰富的框架usc-cli，提供init，compile，migrate， test， wallet等基础操作。通过这些操作，可以选择合约类型模板，设置语法检查，代码格式化，单元测试及测试覆盖率统计；这套工具也支持自动生成合约文档。

Network Adaptor。Untrain链提供测试网络，作为预发布运行环境，测试网络中的结点具有强一致性，同时能够模拟生产网络情况中的平均结点情况。Network Adaptor负责测试测试与生产网络切换和配置。我们提供ultrain-explorer工具， 在浏览器中提供查询blocks，accounts， transactions， api等功能。浏览器的线上版本会集成开发者社区，方便开发者之间进行交流。

多虚拟机运行环境。在不同的侧链中，根据应用场景的实际需求，提供高级语言的原生执行虚拟机。

Ultrain 共识黄皮书

Husen Wang [*]

May 23, 2018

[*]Ultrain team

Contents

Abstract

Ultrain 的目的是为了在不损失安全度的情况下，通过 sharding、并行等一些列创新，提高区块链底层的性能。同时，针对通过对上层智能合约的思考，重新定义应用的架构。

1 前言

对于区块链应用场景，需要共识算法具有如下特性

- Throughput：transaction per second 意味着区块链系统每秒能处理的交易笔数，为了能够满足全球用户，交易处理速度必须足够大，否则会造成交易积压，回复时间长。比如 bitcoin 的交易拥堵和以太坊因为 crytokitties 造成的网络拥堵。
- Latency：交易从提交到回复成功与否的时间。在 PoW 网络中，因为有分叉可能，一般要等待几个块后确定。在 BFT 共识系统中，因为正确节点都在共识结束后就确定性有相同结果，所以在交易不拥堵的情况下，响应速度与共识时间相同。
- Scalability：一般默认区块链系统参与的节点数越多，系统安全度越高。对于基于 PoW 共识的系统，安全度与参与的诚实算力相关。对于基于 BFT 共识的系统，参与节点数越多，能容纳的恶意节点越多，恶意节点少于 1/3。但是对于 BFT 系统，节点越多，通信和计算开销越大。Algorand 提出利用 Verifiable Random Function 随机选出一部分节点形成 committee 参与共识，保证了无论总节点数多大，最后的 committee 数目一定，共识速度就不变。

- Security：对于 BFT 共识，安全度包含 consistence，liveness 和 fairness。Consistence 是系统内所有诚实节点要最终达到相同的状态。Liveness 需要系统在任意情况下都能收敛到确定状态，并且能持续接受交易，产生正确的共识结果。Fairness 在于对于系统的用户，任意合法的交易都不会被拒绝。

对于传统的 PBFT[1, 2] 系统，需要假设网络处于弱同步状态，通过超时和换主来保证 liveness。但是因为换主是确定性地换到预先设定的下一个节点，而且每次换主导致的节点消息同步耗时长，主节点会被连续不断的网络攻击导致瘫痪，最终系统持续换主，永远无法共识交易，相当于停滞状态。对此，Tendermint[3] 提出每一轮共识完成后确定性由下一个节点提出共识请求，从而避免了一直由主节点提交请求，避免了攻击的薄弱点。但是这依然无法避免提交请求节点的可预测性，导致被依次攻击，系统每轮共识都为空。Honey Badger[4] 提出了基于 RBC (Reliable Broadcast) 和 BA (Binary Agreement) 的协议。核心在于任意节点都可以提出共识消息，通过 RBC 可靠传播到所有节点，并且为了减小广播带宽，通过 erasure code 分割消息为多份。同时所有节点都通过 BA 协议共识所有消息，BA 协议的执行会在任意情况下快速收敛到 1 或者 0，表示对共识消息的接受与否。该协议相当于每一轮，所有节点并行地提出交易、共识，最终得到交易的子集作为共识结果。所以对任意一个节点的攻击，都不会造成整个网络的崩溃，只是会影响网络部分性能。而且能充分利用带宽，适合全球部署。该协议的缺点在于交易响应延时比较高，因为每轮要共识多个节点的交易。同时协议需要预先确定节点的集合，不能动态添加节点，不能支持大规模节点。为了解决节点添加和扩充节点的问题，Algorand[5] 提出将 VRF (Verifiable Random Function) 和 BA 结合起来。无论节点数目多少，通过 VRF 和持有的代币数随机选出特定数目节点，然后节点通过 BA 相互发送交易，并对优先级最高 (VRF 随机数最小) 的节点发出的交易共识。为了提高安全性，共识的每一步都通过 VRF 选出新一轮共识节点，从而让攻击者无法预测下一个攻击目标。该协议的主要有点在响应时间短，缺点在于无法做到高吞吐率和小带宽。而且 VRF 假设了正确节点是平均分布在全网中。

Dfinity[6] 主要是通过 threshold 签名来保证 VRF 的每一轮都能确定性地收到 signature。结合 Random Beacon 和 Notaries 来使每一轮的成员都随机选择，并且提交的块按照本轮随机数进行排名。但是一个潜在的经济学博弈问题是，threshold 签名可以通过多个成员的合谋的方法来预测，合谋的成员可以协同计算出群私钥，快速预测出下一轮的随机数。通过作恶 Dos 攻击下一轮的成员的目的，合谋者可以破坏网络的公平性。因为这种攻击是不容易被发现的，所以可以做到零成本收益，因而在现实世界必然会出现。

Casper，基于 RANDAO[7]，类似 Dfinity 的随机数生成方式。

Ouroboros[8]，基于类似 Dfinity 的方式，但是对博弈有更好的理解。

2 贡献

Ultrain 共识具备如下特性

- 通过 VRF 和 (硬件指纹 + 持有代币数) 筛选出参与共识的节点 (侧重硬件性能) 和验证节点 (侧重代币数和信誉评分)，提高了系统的安全性。任意观察到在同一轮给出不同签名的节点，都可以提交该冲突的签名对，区块链将会接收该格式的签名，自动共识移除一定比例的来自该节点的代币。
- 通过利用拜占庭共识过程中的并行性，提高了共识的效率，同时保证在少数共识提交者被攻击的时候，网络依然保持活性。
- 引入硬件独特的计算能力评分和历史记录评分，生成无法篡改的硬件指纹，保证参与共识的节点都是高性能和良好信誉的。
- 通过优化密码学签名验签模块，在共识过程采用轻量级的密码学模块，同时持久化高安全度的密码学结果，提高了节点响应速度。
- 通过冗余编码保证了节点可以充分利用有限的网络带宽，达到最高的共识吞吐率。

- 通过阈值签名，随机将多个节点编成一个组，提高了节点的活性和响应速度，同时提高了公平性。
- 通过阈值加密，保证了每个节点在收到足够多消息之前无法知道消息的内容，所以没法做到有倾向地传递共识消息，提高了公平性。

3 名词解释

3.1 签名验证

Ultrain 选择 ED25519[9] 作为对消息的签名，主要是基于如下考虑

- ED25519 是开源的密码学算法，由 Daniel Bernstein 提出，经过众多密码学专家论证，最后入选 RFC7748 标准。曾经参与过同微软 FourQ[10] 共同竞争下一代更高效更抗侧信道攻击的椭圆曲线算法标准，后因 NIST 直接进入后量子算法的遴选而取消。
- ED25519 有比较高效的开源算法实现，并且汇编实现支持 Intel AVX 指令集。
- ED25519 相对基于 Weistrass 曲线的 NIST P-256 曲线，性能有比较大的提升。

在接下来的数学表达中，

- 签名标记为：$sig_{sk}(m) = ED25519_{sig}(sk, m)$
- 验证标记为：$res_{pk}(sig_{sk}(m), m) = ED25519_{ver}(pk, sig_{sk}(m), m)$

3.2 摘要函数

Ultrain 采取 SHA256 作为摘要函数，基于如下考虑：

- 虽然 SHA3 标准已经提出，但是目前尚没有明显针对 SHA256 的攻击。
- SHA256 性能相对 SHA3 比较高，而且有众多比较成熟的开源实现。
- SHA256 相对 SHA3 或者其他摘要算法，有更多市场可用的实现。

在接下来的数学表达中，摘要函数标记为：$SHA(m) : hash = SHA256(m)$

3.3 可验证随机函数

可验证随机函数要求函数满足如下特性：

- Deterministic，一旦有随机数被引入，节点即可不断尝试新的随机数，直到 VRF 的结果对己方有利为止。
- Non-malleable，一旦签名结果可被变换为不同的，那么用户可以通过修改签名，达到想要的结果。

在接下来的数学表达中，标记为：$proof = VRF_{sk}(seed)$

3.4 后量子算法

量子计算机主要对密码学中的非对称算法和密钥交换影响较大，比如 RSA 基于的大数分解困难问题可用 Schor 算法高效解出，同理离散对数问题也可以解决（如 GEECM ）。然而对于对称算法和单向函数影响不大，只需要为了增加安全性到两倍即可即可。目前 Ultrain 不以后量子算法为主，基于如下考虑：

- 量子计算机的发展仍然有相当的时间长度 [12]，一些关键性的问题如多个量子的叠加、量子纠错。
- 目前抗量子算法如密钥交换和非对称算法标准尚不成熟，NIST 第一轮的筛选刚刚完成。
- 抗量子算法目前有基于 LWE 和 code 的算法，效率普遍比较低，而且安全性的量化评估目前尚无定论（如 LWE 基于的 CVP 和 SVP 问题一直有效率更高的算法被提出）。
- 传统算法基于的困难问题，如基于 pairing 的 ABE/IBE、零知识证明，在抗量子的困难问题基础上构造比较困难。
- 区块链上的数据到后量子的迁移并没有很大难度。主要是因为有足够多的时间，从非抗量子的公钥钱包迁移到抗量子的钱包
- Merkle tree、存证等基于的 HASH 算法即使安全度降低，被碰撞的难度依然比较大。

3.5 区块

区块由如下结构构成

- 块高度 bheight，简写为 bh
- 上一个块的 hash，定义为 $hprev_{bh}$。
- 块包含的交易的 merkle 根 $hroot_{bh}$。
- 交易执行结果的 merkle 状态树根，即 $hstate_{bh}$。
- 交易的 proposers 的 merkle 树根，即 $hproposer_{bh}$。

总的区块格式为 $(bh, hprev_{bh}, hroot_{bh}, hstate_{bh}, hproposer_{bh})$ 考虑到创世区块比较特殊，我们定义为 $(0, 0, hroot_0, hstate_0, 0)$ 其中 $hroot_0$ 包含一个创世合约，其中约束了 ultrain 的规则。$hstate_0$ 包含创世成员的代币分布。

4 架构

4.1 身份层

每个自愿参与共识的节点都有一对密钥 (sk,pk) 作为身份标识。设备被选为共识成员的概率依赖以下三个评分的综合结果。

- 链上代币。所有公钥对应的资产在链上都可以查询到，所以持有代币的百分比作为 w_{coin}。有的节点可能持有大量代币，但是不愿参与共识，可以代理给其他可靠的硬件厂商。或者持有大量代币的节点可能同时分散在多个账户，这不会影响到公平性，因为多个账户的综合概率与一个账户相同（假设其他两个指标相同）。有的节点持有过少的代币，被选中的几率比较小，所以为了提高效率，我们我默认只有前 1000 名的代币持有者有资格参与 committee 筛选。

- 硬件特性。根据自愿参与共识的设备硬件指纹信息可以得出硬件特性评分。该评分可能有多个维度，比如 CPU/网卡/硬盘特性，分别适合 committee 中不同的身份。该权重暂定为 $w_{hardware}$。
- 声誉。根据设备历史参与记录，得到设备成功提交区块的次数，默认提交次数越多，公信力越高，被选中的概率越大。该权重为 $w_{reputation}$。

每个公钥对应一个权重 $w = r_1 * w_{coin} + r_2 * w_{hardware} + r_3 * w_{reputation}$，其中 r_1, r_2, r_3 为系统常量，用来设置各个权重的比率。

4.2 VRF 层

我们选择独立的 VRF，而不是相对类似 RANDAO 或者 Dfinity 的方式。基于如下两点：

- Dfinity 的方式，只要有足够多的节点联合起来，或者一个控制了大量代币的用户，只要超过了 threshold，那么就可以预测下一轮的随机数，这样可以有时间窗口去 DDOS 攻击下一轮的 proposer，以此影响系统的 fairness。
- Dfinity 的方式延时比较高，需要足够多节点响应，才能确定随机数结果。分析一下我们基于 Algorand 的理解。假如设计我们自己的算法，要保证如下几点：
- 可以通过系统参数控制被选中的节点数大概率在一定范围内。
- 即使拆分为多个用户，依然不会影响被选中的概率。即 p(x+y)=p(x)+p(y)
- 能够对系统的代币分布自适应

本层需要初始化的种子 seed，可以放在创世区块中。该种子没有特殊要求，可以是任意 256bit 字符串。VRF 层每次都会根据上一轮的种子，生成下一轮的种子，并包含在下一轮的区块中。假设上一轮 block 中的种子为 seedr，每个可能选为 committee 的节点 (pk, sk)，独立运行伪随机函数 f，输出 256bit，该输出归一化到 [0,1)，小于节点自身权重 w，则为被选中，可以广播消息。即

$$Th_{min} < f(sk, seed_r)/2^{256} < w \tag{1}$$

这里 thmin 为系统参数，可以动态调整，为了提高效率，过滤掉小概率的用户。目的是为了结合 w 的分布，共同设置一个窗口，保证足够多的节点被选中。TODO：为了保证 VRF 层的正确性，我们要对我们的 VRF 函数进行一定的设计，类似 algorand 的方法

- 每一个持有代币数为 s 的节点，通过 VRF 的选举机制，被选中的概率为 p(s)
- 对于任意数 s_x, s_y, $p(s_x) + p(s_y) = p(s_x + s_y)$
- 该设计是为了保证公平性，即一个代币持有者通过拆分为多个持有少量代币的身份，获取更大的被选中的概率；同时这也会增加系统的安全性，否则持有小于 1/3 代币的人可以通过合谋获取大于 1/3 的投票概率，从而影响系统的安全性。
- 节点不能控制 VRF 的结果，而且 VRF 每轮的结果必须是唯一，所以节点每次计算 VRF，类似于对 Random Oracle 的访问。
- 系统可以通过静态或者动态的参数设置，控制被选中的节点数

一个比较直接的例子是 $f(s) = B^{-1}(r; s, p)$，其中 s 为节点持有代币数，p 为被选中的概率，r 为 [0,1] 的随机数，B 为二项分布。假设 r 是均匀的 [0,1] 概率分布，那么期望 $E(f(s)) = s * p$。

因为，$B(v_1 : s_1, p) + B(v_2 : s_2, p) = B(v_1 + v_2 : s_1 + s_2, p)$，所以将同一个节点的代币 s 拆分为两个节点 s_1 和 s_2，得到的投票权期望分别是 $E(f(s_1)) = s_1 * p$ 和 $E(f(s_2)) = s_2 * p$，总的为 $s_1 * p + s_2 * p = s * p$，和一个节点单独采样是一样的；同理，所有节点总的票数为 $\sum s * p$。

4.3 共识层主链 (BFT)

Ultrain BFT 共识通过结合消息可靠广播和 Las Vegas 算法, 在半同步网络下达到多轮大概率收敛。

5 共识过程

5.1 委员会身份

- Proposer: 共识每轮提起交易批
- Voter: 对交易进行确认，发出投票同意打包请求
- Listener：对消息进行传递，收到足够票数后执行

5.2 消息类型

- PROPOSE 消息：提议共识中需要包含的交易，带被选为 proposer 的 proof, 格式为

$$(PROPOSE, shard, txs, bh, round, txshash, proof_{proposer}, sig_{proposer}, pk_{proposer}) \tag{2}$$

  其中 $sig_{proposer} = SIG(bh, round, txshash)$

- ECHO 消息：对共识 PROPOSE 消息进行投票，传递当前 round 优先级比较高 (proof 比较小) 的消息，带 proposal 的 hash, 格式为

$$(ECHO, shard, bh, round, txshash, proof_{voter}, sig_{voter}, pk_{voter}, proof_{proposer}, sig_{proposer}, pk_{proposer}) \tag{3}$$

  其中 $sig_{voter} = SIG(bh, round, txshash)$。

  对于轮次大于 1 的，不需要带 $(proof_{proposer}, sig_{proposer}, pk_{proposer})$。

考虑到共识过程中，除了 PROPOSE 消息外，ECHO 消息比较小，但是占用时间长，为了不浪费带宽，Ultrain 共识同时存在 16 个 shard，每个 shard，包含不同的交易类型，或者不同的 hash 值的交易。为了并行化共识，我们设置了固定 16 个 shard，每个 shard 只选最小 proof 的 proposer, 多个 shard 并行共识，最后一起执行。

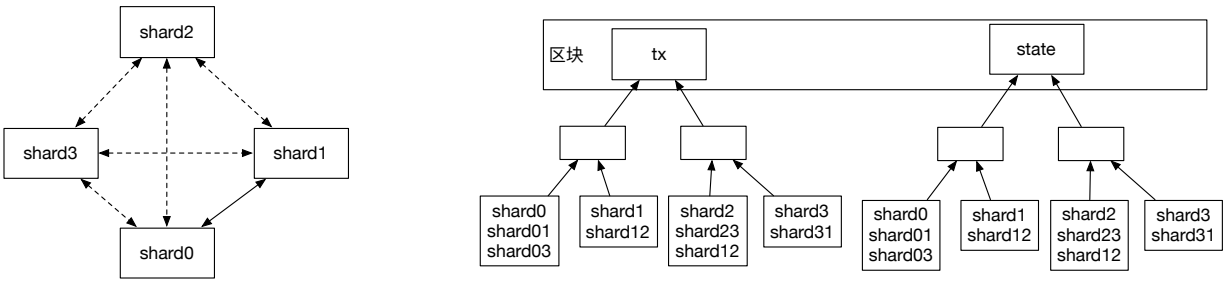


Figure 1: 存储和执行的 shard

对于存储和执行的 shard 我们有以下措施：

- 对于只参与 $shard_i$ 的节点，可能无法对其他 $shard$ 的状态进行验证，这时只需要监听网络中的消息，对其他 $shard$ 的共识执行结果接受即可。
- 当然假如是跨 shard 执行的合约，我们平均分配给跨 shard 执行 C_N^2 交易给网络中不同的 shard，因而其实每个 shard 也应该有能力执行本 shard 和其余 C_N^2/N 个 shard 的的交易，这样一个节点需要存储 $((C_N^2)/N + N)/N$ 个 shard 的状态。

- 不允许一个交易跨多个 shard。即使将来支持，这种交易需要在多个共识周期才能完成，每个共识周期完成一次 shard 的跨越。

用户自主地选择参与到某个的 shard 中。下面只针对一个 shard 中的行为进行描述。

5.3 延时

传统的 BFT 系统采取两种方式：

- (tx 执行 + 提议 +tx 验证 +BA)，这样延时比较大，因为对执行结果的验证相当于重复执行，总延时包含了两次执行过程。好处在于可以避免打包非法交易进入区块。一般基于 PoW 的公链采取这种形式。
- (tx 提议 +BA+tx 执行) 采取共识后执行的方法，避免不了打包非法交易。同时也对 deterministic 有比较强的要求。这里 fabric 分析比较到位。
- (tx 部分执行 + 共识排序 + 共识状态验证和转移) 这是 hyperledger fabric 的方式，好处在于将 non-deterministic 的部分放在共识前，将 deterministic 的验证签名和状态转移，同时可以并行执行无依赖的状态转移。缺点在于，节点不对执行过程进行验证，依赖受状态影响的各方签名校验。

为了减小节点的延时，我们考虑如下几种方式：

- 多轮 tx 共识 + 一轮执行结果共识，这样或许可以将多轮的 tx 批量执行，效率有一定程度的提高。诚实的客户端，只需要收到 tx 被打包进区块的收据，就可以对执行结果有足够的信心，不必等到执行结果被打包成块。
- Tx 共识 + 上一轮的执行结果共识。流水线处理，共识和执行相对独立，执行落后共识一个成块周期。不会减少延时，但是可以做到在同一时间点上可以并行处理共识和执行。缺点在于，会将非法交易打包成块，浪费区块存储。
- (tx 提议 + voter 执行 + voter 以执行结果 hash 响应 + 对执行结果投票)，延时也可以做到最小。优点在于可以避免打包非法交易，同时识别出哪些交易批没有 deterministic 地执行，因为这些交易批收集不到足够多的票数。从经济学博弈的角度，提议者为了获得激励，不会故意提交 non-deterministic 的程序。

我们的共识采用第三种方式，主要是考虑到方便上层智能合约的设计。

- propose 轮。为了保证消息确定性地由 proposer 到全网所有节点。节点基于上一个 block，独立判断是否被选为 proposer 或 voter.
  - proposer 默认也为 voter，在广播 PROPOSE 消息的同时为了加速传播，广播带交易 hash 的 ECHO。为了最小化带宽消耗，PROPOSE 消息被冗余编码后，分别传给多个节点。每个 proposer 只能广播一次，多次广播被视为恶意节点。
  - voter 等待 T_{PROP} 时间后，对收到 PROPOSE 消息校验格式，选择 proof 最小的通过后广播 ECHO。
  - 节点收到 TH 个消息后，即使在未收到 PROPOSE 消息的情况下，也广播 ECHO 消息。
  - 节点收到 TH2 个 ECHO 后完成此轮，返回。
  - 节点判断超时后，将交易批设置为空。
- BA 轮，正常情况 1-2 轮，最大 9 轮。为了保证全网对 proposer 的 hash 达成一致。
  - 全网基于上一个 block 和轮数，独立判断是否被选为 voter，是则广播对 hash 的决定。节点收到 TH2=2/3*N=40 个 ECHO 后完成此轮。超时则成空提议，返回。

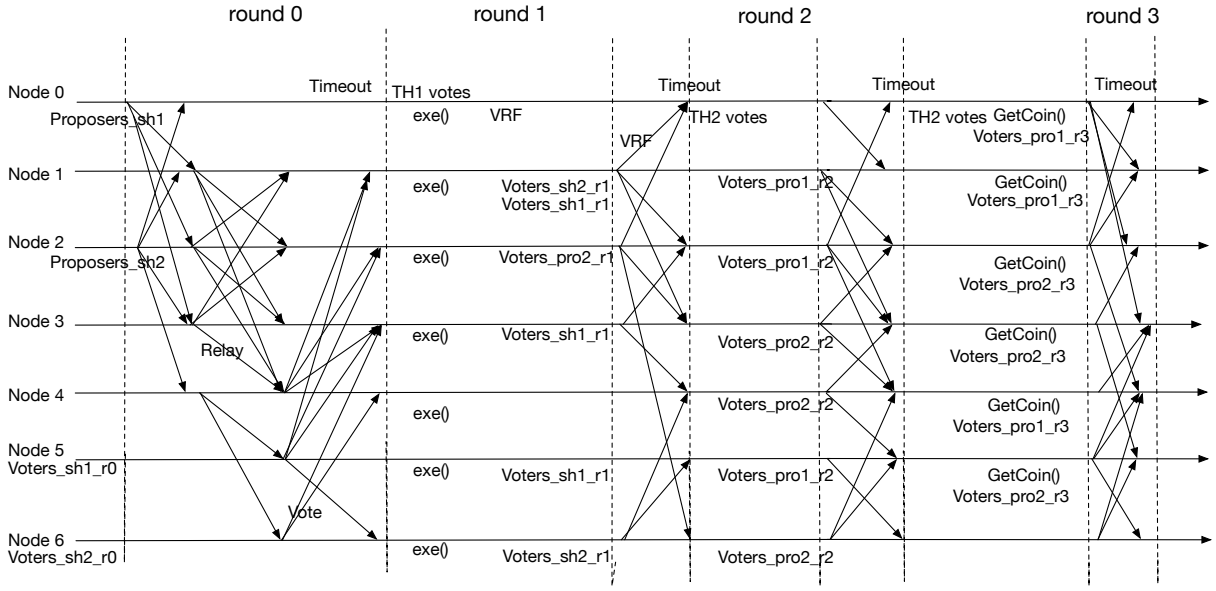


Figure 2: consensus overview

  - 若上轮超时，则新开始一轮，广播。重新判断是否为 voter，是则基于 CommonCoin 决定是否提起共识还是空块。
  - 节点收到 TH2=2/3*N=40 个 ECHO 后完成此轮。超时则成空提议，返回。超过最大共识轮数则返回，成空块。
  - BA 中每一轮时间固定为 T_{Bar}，节点判断超时后返回空。

- EXE 轮
  - 节点对共识结果执行，并广播包含执行结果 hash 值的 ECHO 消息到全网。
  - 节点收到足够多的共识执行结果后，确定为共识输出。

在整个共识过程中，Listener 会接收到跟 Voter 和 Proposer 同样的消息，所以除了传递消息外，Listener 同样会校验、处理消息、成块。

节点对不同 shard 的共识输出打包成块。对于只参与部分 shard 的节点，可以先打包成块，然后再向其他节点拉取其他 shard 的交易批次。具体实现过程中，节点会在条件满足的情况下，提前为下一轮做准备，同时为了提升效率，对已经满足的条件做忽略处理。具体算法描述如下：

TODO：BA 的过程中，接收到超过阈值的 vote 的人，在接下来的轮数中，要坚持发；只有接收到 vote 数不够的人才要确定发什么。

Algorithm 2: Ultrain 节点操作

Data: 初始代币分布 Istake，成员列表 Imember
Result: 区块链状态

Initialization
  $bh = 1$ /* 初始块高度 */
  $state_{bh}, block_0 = (Istake, Imember)$ /* 初始状态和块 */
  $N_{shard} = 16$ /* shard 个数 */
  $(sk_i, pk_i) = KeyGen()$ /* 节点身份 */
  $queue_{txs} = set_{EMPTY}$ /* 初始交易队列为空 */
Upon incoming transaction tx
  /* 节点自愿加入网络某个 shard，并对符合该 shard 的交易进行接收、处理 */
  if *check(tx)* **then**
    $queue_{txs} \cup = tx$
Upon start consensus
  /* last block */
  $round = 0$;
  /* 提议轮 propose，最后获取投票结果和交易批 hash，以及交易批 */
  $res, hbatchtx, batchtx =$
  $ROUND(PROPOSE, hblock_{bh-1}, shard, queue_{shard1}, block_{r-1}, round)$
  /* BA */
  $round++$;
  if $res == TIMEOUT$ **then**
    /* 若上轮收集不到足够选票，超时，则提议空交易批 */
    $res = ROUND(ECHO, shard, set_{EMPTY}, block_{bh-1}, round)$
  else
    /* 保存上一个状态 $state_{bh-1}$ */
    $save(state_{bh-1})$
    /* 执行交易批 batchtx，获得新状态，计算摘要为 $hstate_{bh}$ */
    $hstate_{bh} = exe(state_{bh-1}, batchtx)$ exe(batchtx)
    $res = ROUND(ECHO, shard, hbatchtx, block_{bh-1}, round)$
  while $res == TIMEOUT \& round < MAX_r$ **do**
    /* at most $MAX_r = 7$ rounds */
    $round++$
    $CC = SHA(res \parallel hblock_{r-1} \parallel round)$
    /* 取 CC 的最低位来决策下一步动作 */
    if *LSB(CC)==1* **then**
      $res = ROUND(ECHO, shard, hbatchtx, block_{bh-1}, round)$
    else
      $res = ROUND(ECHO, shard, set_{EMPTY}, block_{bh-1}, round)$
  /* 收集其他 shards 的 tx root and state root，构建块 */
  $build\_block(state_{bh}, hstate_{othershards}, hbatchtx_{othershards})$

Algorithm 4: Node i round action

Data: *Type*, *shard*, 交易批 *batchtx*, 块摘要 $block_{bh-1}$, *round*
Result: 返回值 res, 交易批摘要 hbatchtx, 交易批 batchtx

初始化
  $set_{ECHO} = emtpy$ /* 响应 hbatchtx */
  $set_{txs} = empty$ /* HASH(batchtxs) : batchtxs */
  $sent_{ECHO} = empty$ /* sent ECHO HASH(batchtxs) : batchtxs */
Upon Start
  start timing
  $proof = VRF_{sk}(block_{prev} \parallel shard \parallel round \parallel role)$
  if $proof[193:256] < score$ /* score is 64-bit Uint */
  then
    $sig = SIG_{sk}(hbatchtxs)$ /* can be paralleled */
    if $round! = 0$ /* BA */
    then
      broadcast (ECHO, *shard*, height, round, batchtxs, proof, sig, pk)
      break
    hbatchtxs = SHA(batchtxs)
    broadcast (PROPOSE, *shard*, height, round, hbatchtxs, batchtxs, proof, sig, pk)
    /* accelerate vote */
    broadcast (ECHO, *shard*, height, round, hbatchtxs, proof,sig, pk)
/* Never process or relay the same message */
/* relay before process to optimize? */
Upon receive PROPOSE
  if *HASH(PROPOSE.batchtxs)!= PROPOSE.hbatchtxs* **then**
    break
  /* have received such ECHO or not */
  if $PROPOSE.pk \in set_{ECHO}$ **then**
    $set_{txs}[PROPOSE.hbatchtxs] = PROPOSE.batchtxs$
    return 1
  if *receive_ECHO(PROPOSE\ PROPOSE.batchtxs)* **then**
    $set_{txs}[PROPOSE.hbatchtxs] = PROPOSE.batchtxs$
Upon receive ECHO
  /* 2 verification if no relay */
  if *VER(msg)* **then**
    $set_{ECHO}[ECHO.hbatchtxs] \cup = ECHO.pk$
    /* $TH2 = Threshold2 = 2/3 * N_{voters}$ */
    if $count(set_{ECHO}[ECHO.hbatchtxs]) == TH2$ **then**
      /* If propose, should get the txs; If not, fine */
      if $round! = 0 \parallel ECHO.hbatchtxs \in index(set_{txs})$ **then**
        return hbatchtxs
      /* wait for some time to get the batchtxs */
    /* $TH1 = Threshold1 = 1/3 * N_{voters}$ */
    if $count(set_{ECHO}[ECHO.hbatchtxs]) == TH1$ **then**
      if *not* $ECHO.hbatchtxs \in sent_{ECHO}$ **then**
        add_sig_then_send(ECHO, ECHO={ECHO, hbatchtxs, sig_{msg}, proof, pk)}, sk, pk)
        $sent_{ECHO} \cup = ECHO.hbatchtxs$
Upon TIMEOUT
  return TIMEOUT

Algorithm 5: append signature to original message

Result: $Type, msg_{in}, sk, pk$

/* 在原消息添加节点签名 */

$send(Type, msg_{in}.msg, msg_{in}.sig \cup SIG(msg_{in}.msg, sk), msg_{in}.proof, msg.pk \cup pk)$

Algorithm 6: 消息验证

Result: $msg_{in}, sk, pk, block_{prev}, pillar, role$

/* parallelized if possible */

if $msg_{in}.proof[193:256] >= score$ **then**

  return 0

/* use batch verification to accelerate it */

if $!VER_{msg_{in}.pk}(proof, block_{prev} \parallel pillar \parallel role)$ **then**

  return 0

if $!VER_{msg_{in}.pk}(msg_{in}.sig, msg_{in}.msg)$ **then**

  return 0

return 1

5.4 证书保存

证书的保存对新加入的节点同步，有比较大的影响。

- 聚合签名
- 采用 DAG 的方法，任意一个节点，都要对现存的两个 signature 进行验证。然后只要锁定了 DAG 的根，那么就可以对证书进行确定。

5.5 新节点加入

新节点需要经过如下几个步骤：

- 观察并接收网络正在传播的消息，确定块高度和共识的轮数，当前块共识结果。为了防止被恶意节点通过网络控制的方式攻击，新节点需要尽量从多个网络地址监听共识消息。
- 根据当前成块，向网络中其他节点拉取历史区块，并验证链的正确性。考虑到链的单向性，新节点可以保证历史区块的正确性。

5.6 网络通信

以太坊的网站上提到，为了实现真正的 sharding，p2p 底层网络也需要修改，要不每个节点都要接受和发送 $o(n)$ 的消息，这与节点自身 $o(c)$ 的带宽不符。

6 安全性证明

6.1 定理

Theorem 6.1 (FLP 不可能性) 异步系统下，即使只有一个节点崩溃或者作恶，不可能构造在确定有限时间内的共识算法。

Theorem 6.2 (容错上界) 对于部分同步网络，最多容忍 *1/3* 的恶意节点。异步网络，确定性共识算法不能容错。强同步网络可以达到 *100%* 容错。

Theorem 6.3 (CAP 定理) 对于网络分区，无法同时保证一致性和可用性。

6.2 安全假设

Assumption 6.4 (网络连通性) 网络不会被超过 *1/3* 的恶意群体控制，任意两点的消息传播通过 *gossip* 协议，保证消息在确定时间范围内最终大概率到达。同时消息的源头不会被轻易识别出来，被攻击。

Assumption 6.5 (节点恶意行为) 网络上的节点可以发送任意恶意消息或者串谋、拒绝响应，只要总持币数不会被 *1/3* 的恶意节点控制，那么区块链系统的安全性和活性就能得到保证。

Assumption 6.6 (VRF 算法) *VRF* 算法保证，持有比例为 x 的代币，大概率不会产生超过比例 x 的投票权。

Assumption 6.7 (弱同步时钟) 各个节点依赖本地时钟和网络消息判断何时进入下一轮。不同节点的本地时钟可能存在差异，需要定时跟网络时钟服务（如微软）进行矫正。我们的每轮是固定时间宽度，考虑了消息在不同网络拓扑下的传播时间分布和节点处理消息的时间。

Assumption 6.8 (弱同步时钟) 通过将用户的共识权利和花费代币权利进行独立和绑定，我们保证了用户身份的切换，同时支持类似 *dpos* 的方式。但是在 *dfinity* 中，重新形成一个 *distributed key* 来组成 *threshold* 签名是比较困难的，而且其 *key setup* 过程容易收到 *dos* 攻击。

Assumption 6.9 (恶意节点数) 假设网络上不可能有两个相同的集合，每个集合包含了相同的签名；即假设了足够的恶意节点持有的代币数。

6.3 推论

虽然拜占庭共识能保证全网系统的最终一致性，但是因为我们在公网上，情况比较特殊：

- 为了扩展性的要求，考虑的是通过委员会 + 阈值的方式，而不是传统的每个节点接收到其他全部节点的消息后做响应。
- 节点间的信息传输非点对点传输，而是通过公网 gossip 方式传播，恶意节点可能会将消息只广播到部分网络分区，能在超时范围内，影响该网络其他节点的接收消息的完整度。

恶意节点可能会有以下几种攻击的可能性：

- Case1: 作为 proposer，有意向一部分网络节点发送 PROPOSE 消息，向另外一部份节点延迟发送，或者根本不发送。
- Case2：作为 voter，有意向一部分网络节点发送 ECHO 消息，向另外一部份节点延迟发送，或者根本不发送。

Corollary 6.9.1 在假设 *A0* 的情况下，*Case1* 不会影响区块链的一致性和活性。

证明：VRF 保证了 voter 分布的不可预测性和随机性，所以假如 PROPOSE 消息没有大于 TH1 的网络上传播，大部分节点将无法在足够的超时范围内收到足量 ECHO 消息，从而以空交易批的方式进入第二轮；对于少部分能在超时范围内接收到 ECHO 消息的节点，在第二轮将无法得到足够多的对应交易 hash 的 ECHO 消息，而会在第三轮收到空交易批的 ECHO 消息，所以最后所有节点都会以空的交易批结束。网络的一致性得到保证。同理，假如 proposer 对不同的网络分区广播不同的 PROPOSE 消息，也会导致所有消息都接收不到足量的 ECHO 消息，最终所有节点以空交易批结束。活性：VRF 会选足够多的 proposer，并且有不同的 shard，另外引入公平的随机性，从而保证所有 proposer 都是恶意节点的概率非常小，系统会大概率获取一定的交易批次。

Corollary 6.9.2 在假设 *A0* 的情况下，*Case2* 不会影响节点的一致性。

证明：voter 在 case2 主要是干扰网络的一致性，VRF 保证了 voter 分布的不可预测性和随机性，所以其实 voter 无法确定其他 voter 的位置，假设全部 voter 在网络的分布是均匀的，恶意 voter 很难通过只广播小部分分区，影响大部分 voter 的目的。即使假定 voter 能控制足够量其他 voter 接收的消息，从而控制分区 1 超过阈值收敛为交易批，分区 2 超时，但是因为我们超时后的行为是随机的，继续提交交易批，或者提交空交易批，所以这种不可预测的行为让 voter 无法控制网络，最后超出固定轮数后，所有 voter 最后以空交易批结束。但是，这属于极端情况，因为恶意节点不能一直作为 voter 存在，所以最后网络还是不受恶意节点控制。

Corollary 6.9.3 基于假设 *A2.2* 和 *A2*，强同步网络下，*Ultrain* 共识第二轮结束后即完成；半同步网络下，*Ultrain* 需要多轮可以大概率收敛。

证明：在强同步网络下，任何节点在超时范围内，都能收到其他节点发过来的消息。基于假设 A2.2 和 A2，网络恶意投票数不会超过 1/3，所以所有节点在第一轮都可以收到所有的共识提议，无论是恶意还是诚实节点发送的。在第二轮过程中，只有诚实节点的提议会收到足够多的票数，所有节点都确定性地执行，并达到相同的结果。在半同步网络下，基于 [P0] 和 [P1]，所有节点依然可以保持确定时间收敛和一致性。

Corollary 6.9.4 *Ultrain* 可以防御女巫攻击。

女巫攻击主要是利用系统的不公平性，为某一实体谋取利润的方式。我们有如下防御方式

- 针对利用大量的假名，构造多个伪身份对应一个真实身份的攻击，Ultrain 采用硬件指纹的方式，唯一地对网络上的代币持有者进行标示。
- Ultrain 采用满足假设 A2.2 的 VRF 函数，保证即使用户拆分为多个伪身份，依然不能产生超出总持有代币的投票权，所以不能进行女巫攻击。

Corollary 6.9.5 *Ultrain* 可以抵御 *Selfish-mining* 攻击

Ultrain 基于的 BFT 系统是保证所有节点的一致性，所以即使外部节点独立生成一条链，也不会被网络诚实节点接受，从而无法影响节点一致性。同时为了保证节点

Corollary 6.9.6 *Ultrain* 可以抵御 *Nothing-at-stake* 攻击

Ultrain 中的见证节点相当于投票节点，每个代币持有者的投票能力与代币成正比，低于 1/3 的持有者不会有能力产生分叉。通过硬件指纹的唯一性，Ultrain 也会对双重投票的节点进行一定比例的惩罚。

Corollary 6.9.7 *Ultrain* 可以抵御 *DoS* 和 *DDoS* 攻击

DDOS 是通过消耗资源使被攻击者的服务机不工作，DOS 是直接利用对方的漏洞。即使共识代码出现了一定的 bug，系统被攻击崩溃也是有一定难度的，主要基于如下几点：

- 攻击成本是 $o(n^2)$，所以同时攻击多个节点的成本是很高的。
- 基于 gossip 的话更难被锁定信号源进行攻击。
- 假如是放在 sandbox，签名放在外面的话，或者用 sgx，那么即使实现有 bug 不会影响到单个节点的安全性，比如是代币被花；应该采用不同的 vote key 和 tx sig key；sandbox 不会让系统其他模块被污染，所以我们需要隔离好用户的钱包和共识代码，特别是针对 PoS 场景，不能采用同样的账户。
- 假如是有足够的时间窗口，能快速启动的话，那么能保证节点快速启动，不会影响到整个系统的 livenss。

网络分区将导致区块链系统的可用性和一致性不能同时满足。一般情况，区块链系统都选择中止（比如基于），或者分叉（比如比特币）。为了及时发现分区，我们的 VRF 函数考虑了在不同地域的采样，比如要求 voter 必须在不同的国家内有一定的比例。因为我们不想在中国和美国都出现脑裂的情况，系统也出现脑裂。

References

[1] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[3] Jae Kwon. Tendermint: Consensus without mining. *URL http://tendermint. com/docs/tendermint {_} v04. pdf*, 2014.

[4] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. SOSP, 2017.

[6] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series consensus system, 2018.

[7] Randao: Verifiable random number generation.

[8] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[9] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[10] Patrick Longa. Four and four lib.

[11] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[12] Scott Aaronson. The limits of quantum computers. *Scientific American*, 298(3):62–69, 2008.