BTN 技术白皮书

[版本 V1.0]

2020 年 3 月 3 日

BTN 团队

目录

1 介绍：区块链与 BTN 核心价值 5

2 POE 共识算法 7

2.1 从难题开始：性能与去中心化的冲突 7

2.2 技术方法：核心技术详解 7

2.2.1 两级体制 7

2.2.2 备选记账人竞争 9

2.2.3 出块 13

2.2.4 链的形成 14

2.3 其他关键技术 14

2.3.1 主要编程语言和开发平台 14

2.3.2 网络通讯部分 15

2.3.3 通讯协议 16

2.3.4 远程异步 20

2.4 BTN 总体架构 22

3 市场优势 23

3.1 二级节点的数量优势 23

3.2 每个家用设备都有机会出块 23

3.3 核心优势总结 23

4 风险 24

4.1 技术风险 24

4.1.1 作恶的方法、可能性和后果 24

4.1.2 质押机制 24

4.1.3 Double spend 25

4.1.4 女巫攻击 25

4.2 竞争风险：与 Bitcoin, ETH, EOS 的比较 25

4.3 组织和资金风险 25

5 展望 26

5.1 完整应用 26

5.2 生态系统 26

6 答疑 27

6.1 量子计算 27

6.2 女巫攻击和 TPM 27

6.3 TPS 27

6.4 未能将交易及时打包 28

7 术语 29

摘要

BTN 在全球首创 **POE(Proof of Existence)** 共识算法机制, 创新自研非对称秘钥抽签, **多级封印**等核心解决方案, 以 RSA 密码算法与 IBC 身份标识密码算法为混合底层, 通过**非交互式零知识证明 (NIZK)** 和 **TPM 安全芯片**, 真正实现在图灵完备智能合约的高性能并发无限扩展的区块链可信平台。它整合了可信计算芯片硬件方案，在 OSI（Open System Interconnect）开放式系统互联中采用 IBC 身份标识密码算法（Identity-Based Cryptograph），实现了以 IBS 身份签密（Identity-Based SignCrypt）方式的点对点可信身份认证与数字资产交易, 进行灵活安全性和隐私保证。

本文中的内容为技术 V1.0 版本, 仅覆盖了核心共识算法、系统架构和组织机制等, 其它设计, 技术, 系统和应用会在后期版本逐步更新。除了区块链应用之外，BTN 所具有的吸纳二级节点的特性，我们可以借此打造内容、发行、私有云、边缘计算、分布式计算和存储等生态系统。

1 介绍：区块链与 BTN 核心价值

在数字转型的深刻影响下。一个并行的数字世界，已经与我们的物理世界越来越相似。我们正处于从物理世界向数字世界迁移的伟大时代！在这数字迁徙的浪潮中，区块链技术更像是催化剂，它在数字化转型中起着至关重要的作用。作为信任协议，可以将其视为物理世界和数字世界之间的链接。凭借其信任，分配和价值的特征，区块链使识别和识别连接物理对象及其数字表示。区块链其实更像数字世界中是钥匙与锁的信任游戏，而我们的存在本身正不知不觉地数字化。

2009 年出现的比特币是伟大的发明，它用巧妙的技术算法和代码，成功地开启了一次社会实验，成功地将货币价值映射进入数字世界，随着系统的去中心化爆炸式地生长，也带来了算力垄断和矿池化的局面，少数人在这野蛮的新世界中暴富，而多数人都与此无缘，很多人惊叹比特币是奇迹，但其算力竞争导致的系统生态的不公平性愈发的明显。经历了十余年的发展，野蛮的算力帝国崛起，算力越来越集中，本来应该去中心化的系统沦为一家或几家控制，随意操纵区块链网络，违背了区块链去中心化的初衷。同时不断的分叉事件，对于上千亿资金系统来说，糟糕透顶。

比特币出现的时代在移动互联网之前，节点设计为 PC，所有节点都是平等的用算力竞争。然而移动互联网，网络设备是多元化，移动便携化，网络底层架构也发了巨大的变化。区块链技术也不断的创新突破，采用了智能合约以太坊，虽然开启了发币高潮，但是仅一千多的 TPS 和严重的网络拥塞，也远不及真正商用的标准。而在之后的主流共识算法创新中的 POS 和 DPOS 虽然貌似达到了性能标准，但难以摆脱少数节点垄断的嫌疑。

区块链受到重视，是因为这种新的技术体系巧妙的利用了计算机、网络和数学方法，实现了去中心化、不可篡改等目标，急剧的降低了整个社会中的不透明性和不可靠性，从而大幅度的降低了实质上的交易成本，改变了社会的信任体系，会强烈的影响现有的金融、商业、中介和公正监督体系。这会涉及到我们生活中的现金、银行、保险公司、券商、交易所、国际与国内贸易、商业合同、股权凭证、各类证书（房产证、学历学位证、车辆行驶证等等）、产品认证、著作权登记等方方面面。

而另一方面，区块链作为一种复杂、不容易理解，又和金融以及我们的日常生活息息相关的新事物，吸引了广泛的注意力，甚至被用来作为地下金融和网络化销售的工具。区块链本就具有融资工具、交易工具等金融属性，而在初期野蛮生长的阶段又缺乏监管，这造成了不少问题。但这不是说区块链本身有问题，区块链是一种类似于水、电、电话、互联网这样的基础设施技术，相对于金融行业，这个特性尤其明显。由于这些特性，监管体系是必不可少的。最终，随着监管体系的成熟，大家对区块链的理解和应用变得成熟，区块链最终将成为整个社会的基础设施之一。

我们认为成为一个理想的区块链通用基础设施系统，应该需要满足以下要求：

1. 高并发低时延百万级别的 TPS
2. 不需要过多的算力竞争，零门槛参与
3. 网络节点除通信记账外还能合理资源利用，共享带宽与存储
4. 系统可以轻松无限扩展
5. 广泛去中心化，没有矿霸垄断

针对以上的要求，BTN 团队经过了长期努力，重新设计了区块链结构和各类底层协议，BTN 在全球首创 POE(Proof of Existence) 共识算法机制，创新自研非对称密钥抽签，多级封印等核心方案，以 RSA 密码算法与 IBC 身份标识密码算法为混合底层，通过非交互式零知识证明 (NIZK) 和 TPM 安全芯片, 真正实现在图灵完备智能合约的高性能并发无限扩展的区块链可信平台，旨在为网络上不同角色与机构提供了一个便捷、安全、可信的区块链应用入口，将一个一个的信任价值孤岛连接成价值互联网络。

“BTN” 是一个基于去中心化的区块链可信网络，整合了可信计算芯片硬件方案，在 OSI（Open System Interconnect）开放式系统互联中采用 IBC 身份标识密码算法（Identity-Based Cryptograph），实现了以 IBS 身份签密（Identity-Based SignCrypt）方式的点对点可信身份认证与数字资产交易，在域内和域间基于 CL - PKC 的双向认证和密钥协商协议, 进行灵活安全性和隐私保证。

区块链主要解决的问题是信任成本，但单独依靠区块链技术难以打造爆款应用。区块链本身并不是一项能够直接赋予人们权利的技术。真正的爆款和应用突破，应该是结合区块链，AI 和物联网技术。这也是 BTN 致力于发展的方向。

BTN 的目标是利用自己的核心技术和市场方法，有效的解决区块链走向实际应用中面临的性能、可靠性的问题。具体方法将在下一节阐述。

企业的盈利能力来自于其解决问题的能力以及企业在人的心智中的地位，也就是我们所说的品牌价值。BTN 所发明的方法，是一个适合规模化的方法，这一方面给 BTN 提供了更好的可靠性、容灾能力，另一方面，也摆脱了现有的区块链技术为人诟病的池化、垄断化的问题，更加容易进入到普通家庭，因而也更能够抢占人们的认知资源，随着 BTN 的成长，承载的应用的日益增多，它将成为承载各类信用应用的第一选择，就好像谷歌是人们搜索的第一选择，Facebook 是人们沟通的第一选择一样。BTN 将会成为信用应用的代名词。

2 POE 共识算法

2.1 从难题开始：性能与去中心化的冲突

区块链技术提供了一套简单可靠的信任机制，这是一个非常大的进步。信任和验证机制，是我们人类社会的重要组成部分。我们选择有信誉的产品、银行、酒店、保险公司，除了产品和服务本身，信任也是其中一个重要因素，而且我们还为此支付了溢价。在有的领域，由于高度的垄断或者专业性，对普通人来说，透明度很低，我们的信任成本更加巨大。而区块链技术，虽然也很复杂，但它的算法是公开的，专业人员都可以去校验，所以这是一次巨大的进步。

当然这一技术也并不完美。一个问题性能与去中心化的冲突。对于比特币和以太坊这样的系统来说，越多节点，意味着记账的人越多，账目越可靠，但反过来，节点越多，意味着通讯越多，系统的处理能力也就会随之下降。

这可以说是区块链技术目前的核心难题之一。但难题是挑战，更是机遇，我们就从这个难题出发。

2.2 技术方法：核心技术详解

我们怎么样才能解决这几个问题呢?

- 保持分散性，不被矿主垄断、操控。节点可以无限扩充，让更多的人参与我们，让我们的系统越来越可靠
- 保持良好的性能，不因为节点太多而处理能力越来越差

我们的核心方法就是抽签、分级和广泛验证。通过抽签和分级的结合，可以让系统拥有几乎不受限制的扩充能力，并保持很高的处理性能。下面这个图概括了分级机制和系统的运转过程。接下来我们将详细的讲解系统的各个关键点。

2.2.1 两级体制

在比特币和以太坊的系统中，所有的节点都是平等的，所有的节点必须具有点对点的通讯能力，所有的交易都在节点之间直接广播，所有的节点都有生成区块的能力。所有的节点都平等的竞争，通过 POW，来获得最终的记账权。这种机制是非常好的发明，让各个节点能够做到在独立运算无需投票过程的情况下就能够达成一致。但它有如下几个问题:

- POW 本身是没有意义的，只是为了求得一个特定的 Hash 值而作的无意义的运算。这是比特币和以太坊被人批判的一个点。
- 导致了投机者，也就是矿机的出现，矿机专注于 POW 运算，不做别的事情，这排挤了 PC 等通用型计算设备。

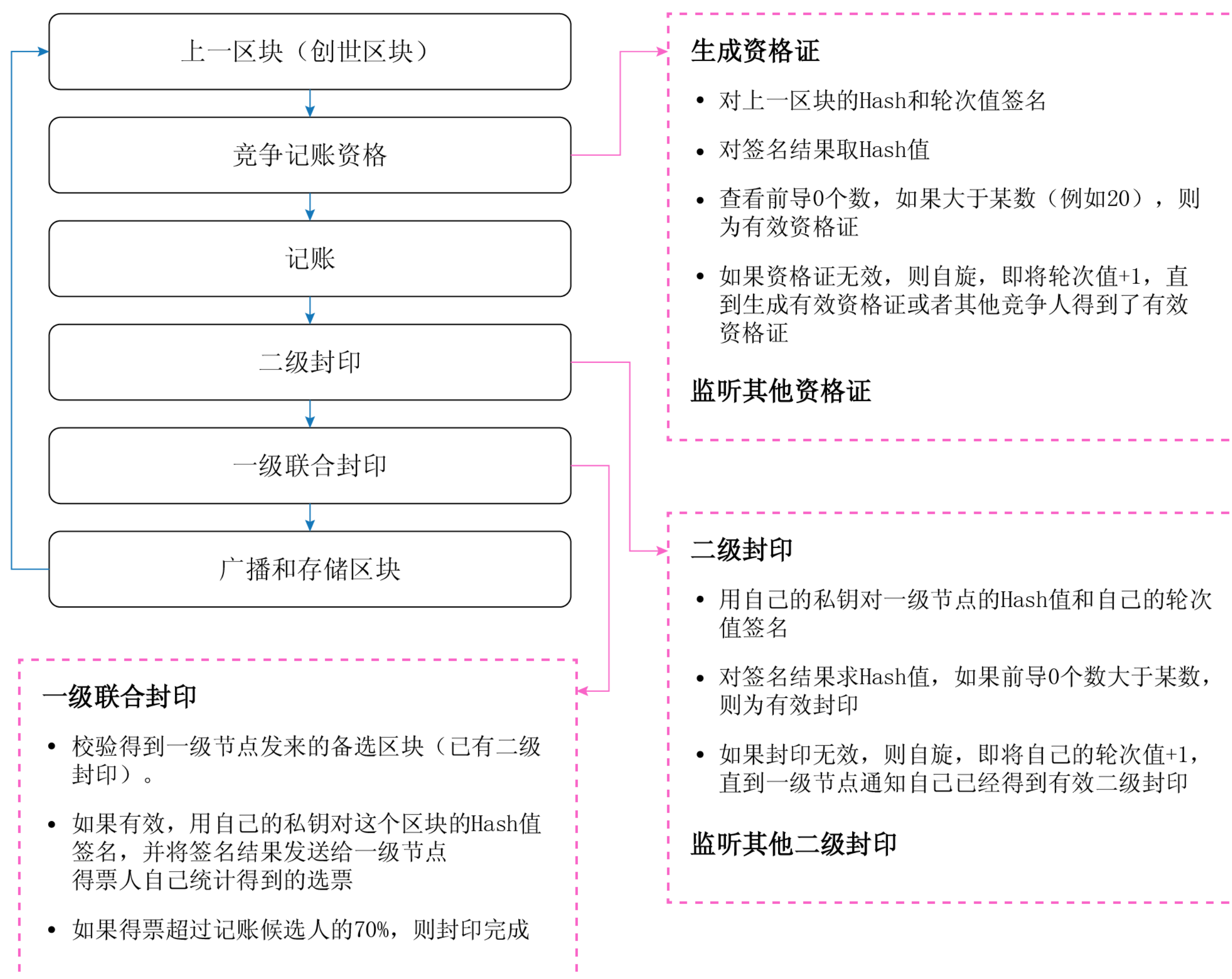


图 1: 技术方法

- 因为路由器和防火墙的限制，节点往往必须部署在机房，导致了矿池的形成。

我们采用两级机制，来避免这几个问题。这种分级机制不同于比特币、以太坊、EOS、Algorand。第一级的节点，是公网节点，这些节点主要承担记账和通讯的职责。第二级节点，放在家里就可以，它的作用在于见证和检验一级节点的记账结果。

2.2.2 备选记账人竞争

A. 抽签算法

比特币和以太坊采用 p2p 的技术，如果有一笔交易发生，需要广播到整个网络。其实并不需要这样做。EOS 等采用某些机制选取特定的节点，来集中处理所有交易，实现了 TPS 的迅速提高。但这样也带来了一个问题。EOS 通过所谓的 PoS 来选取记账节点，但有被垄断的嫌疑。只有真正的可靠的基于大基数的随机算法才能摆脱这种质疑。

我们所采的方法是用私钥对 hash code 签名，然后再取 hash 值 (256 位)，接下来判断前导 0 的个数。只需要一点简单的数学推导，就可以得到我们需要的结果，其依据如下：如果要求 1 个前导 0，其他位任意，概率为 0.5。如果要求两个前导 0，概率为 0.25。以此类推，容易得到，要求 n 个前导 0，概率是

$$\frac{1}{2^n}\,(n = 1, 2, 3, ..., 256)$$

例如，如果要求有 20 个前导 0，则概率为 1/1048576。我们可以根据这个规则，来选取潜在的记账节点，即可以根据整个系统中节点总数，调节前导 0 个数，来决定有多少人将获得记账资格。

如果我们的系统一共有 100 万个一级节点，如果我们要求有 20 个前导 0，则潜在的记账节点约有 1 个，依此类推：

如果我们只要求有 19 个前导 0，则潜在记账节点约有 2 个

如果我们只要求有 18 个前导 0，则潜在记账节点约有 4 个

如果我们只要求有 17 个前导 0，则潜在记账节点约有 8 个

……

这个过程中，用私钥对前一个区块的 Hash 值签名也是重要的步骤。用私钥对整个区块签名显然是不现实的。仅仅用私钥对 Hash 值签名，就可以产生一个只有这个节点能够拥有的，同时又和前一个区块相关的数据。这可以把这个节点的信息可靠的加入到历史区块中，极大的降低被伪造的可能性。

用私钥对 Hash 值加密之后，得到的是非定长数据。对这个数据再次取 Hash 值，可以把非定长数据再次变成定长数据，简化我们的数学计算，形成简单、一致的抽签规则。

以从 1 到 1 千万之间的所有数字（转换成字符串）的哈希值 (SHA1,160 位) 为例，前导 0 的个数 >20，只有 7 个数字，分别是：

```
4233739
leading zero count=21
4836969
leading zero count=22
5370398
leading zero count=23
5871620
leading zero count=21
6132845
leading zero count=25
8520920
leading zero count=21
9057261
leading zero count=22
```

本部分的相关代码如下：

```
fun hash(data:ByteArray):ByteArray{
  val md = MessageDigest.getInstance(SHA_KEY)
  val hash = md.digest(data)
  return hash
}

fun leadingZeroCount(hash: ByteArray): Int {
  var count = 0
  val zero:Byte = 0
  var curByte:UByte = 0xffu
  for ((index, b) in hash.withIndex()){
    if(b == zero)
      count += 8
    else{
      curByte = b.toUByte()
      break
    }
  }
  return count + curByte.countLeadingZeroBits()
}


fun sign(privBytes:ByteArray?,data:ByteArray):ByteArray{
  return rsaEncrypt(privBytes,data)
}
```

```
26
27  fun rsaEncrypt(privBytes:ByteArray?,data:ByteArray, keyIsPub:Boolean = false):
      ByteArray{
28    val key = getRsaKey(privBytes,keyIsPub)
29    val cipher = Cipher.getInstance(RSA_ALG)
30    cipher.init(Cipher.ENCRYPT_MODE, key)
31    return cipher.doFinal(data)
32  }
33
34  private fun getRsaKey(bytes:ByteArray?,isPub:Boolean): Key {
35    lateinit var keySpec: EncodedKeySpec
36    val keyFactory = KeyFactory.getInstance(RSA_KF)
37    return if(isPub){
38      keySpec = X509EncodedKeySpec(bytes)
39      val k = keyFactory.generatePublic(keySpec)
40      k
41    }else{
42      keySpec = PKCS8EncodedKeySpec(bytes)
43      keyFactory.generatePrivate(keySpec)
44    }
45  }
46
47  ...
48
49  val signedData = sign(privateKey,bytes)
50  val signedHash = hash(signedData)
51  val zeros = leadingZeroCount (signedHash)
```

B. 备选记账人竞争

所有一级节点用自己的私钥加密上一个区块的 hash code，并将加密结果 sha256，如果前导 0 数量满足要求（例如 m 个），则获得记账资格。

所有获得记账资格的节点向整个网络发起广播，所有要提交交易的节点，验证记账节点的善恶，通过则将自己的记账请求提交到记账节点。

C. 自旋

因为抽签算法是概率性的，有可能无法产生合法的候选人。此时每个候选人可以加入轮次，再重新签名求 sha 256，直到自己得到资格，或者收到更佳的竞选结果。

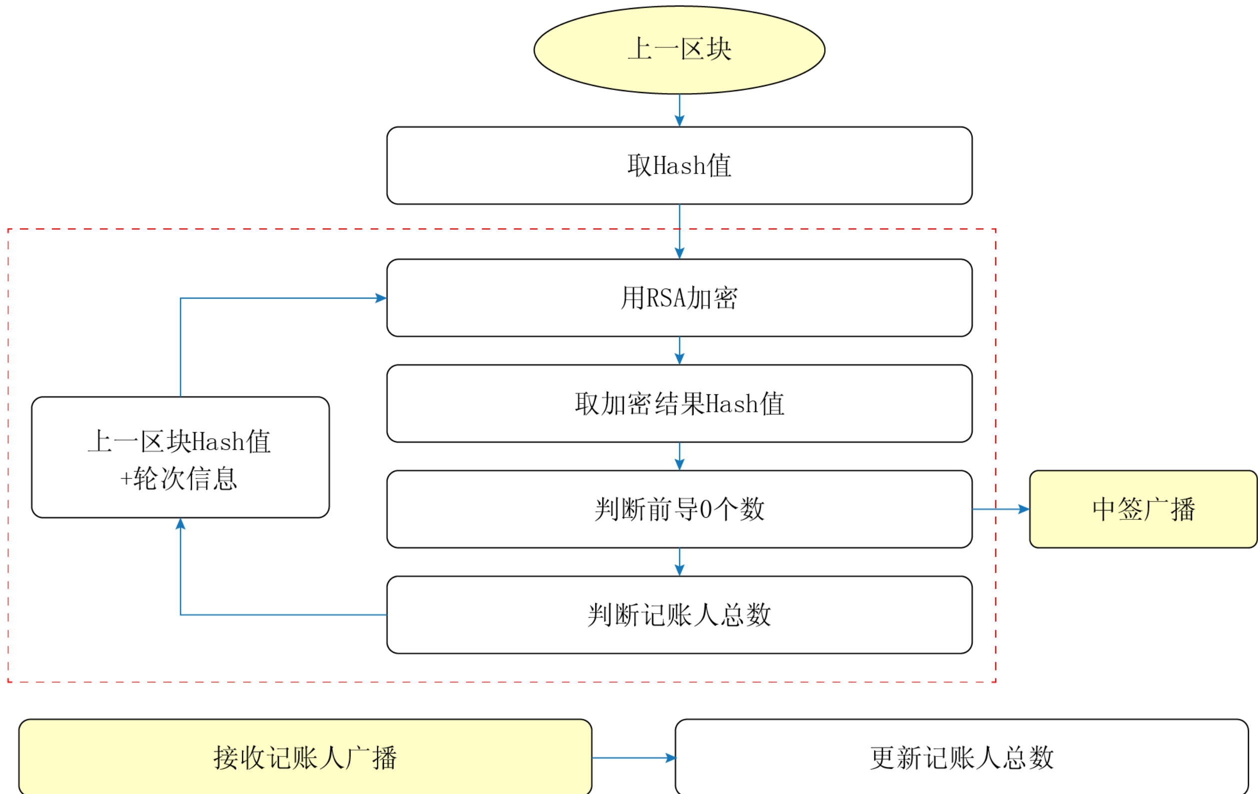


图 2: 抽签算法

自旋方法对一、二级节点都适用。

代码如下：

```
abstract class MasterNode:NodeInstance(){
    fun compete(block:Block){
        val hash = block.hashCode
        var round = 0
        while(true){
            val bytes = combineByteArray(hash,round)
            val signedData = sign(privateKey,bytes)
            val signedHash = hash(signedData)
            val zeros = leadingZeroCount(signedHash)

            if(btnNetwork.clerkIsReady)
                break

            if(zeros < btnNetwork.clerkZeros){
                round++
```

```
16                  continue
17              }
18
19              btnNetwork.broadcast(hash,signedData,publicKey)
20              break
21          }
22      }
23  }
```

2.2.3 出块

A. 第一备选状态

一级节点记账完毕后，区块进入第一备选状态，他将自己打包的区块广播到整个网络中。网络中的二级节点从自己关联的一级节点拉取并校验这些区块。

B. 第二备选状态：二级封印

如果区块合理有效，二级节点用自己的私钥对这个备选区块的 hash 值进行加密，再取加密结果的 sha256，如果前导 0 符合要求（例如 m 个），则此区块进入第二备选状态。

一级节点愿意将自己的区块交给别人检查的动力来自于数量。越多的人知道并认同它的区块，它越有可能得到二级封印。

代码如下：

```
1   abstract class SupervisorNode:NodeInstance(){
2       fun sealBlock(block:Block){
3           if(!block.verify())
4               return
5
6           val signedData = sign(privateKey,block.hashCode)
7           val hash = hash(signedData)
8           val precedingZeroCount = leadingZeroCount (hash)
9           if(precedingZeroCount <= btnNetwork.sealPrecedingThreshold)
10              return
11
12          block.clerk.seal(signedData,publicKey)
13      }
14  }
```

```
15
16  abstract class MasterNode:NodeInstance(){
17  fun onBlockSealed(sealedBlock:Block){
18          btnNetwork.broadcastSealedBlock(sealedBlock)
19  }
20  }
```

C. 第三备选状态：一级联合封印

所有的记账候选人，同时也是票选委员会的成员。当有一个记账候选人得到一个合格的封印过的第二备选状态的区块，它将会把这些信息发送给票选委员会。因为每个记账候选人都会向整个网络广播自己的记账资格，所以整个网络中的每一个节点，都有一个自己维护的记账委员会的名单。

投票委员会（原来的记账候选人）收到诸多第二备选状态的区块之后，计算各个区块的优先级（比较前导 0 个数），找到优先级最高的备选区块。投票人将自己对区块的 hash 值的签名发回给拥有第二备选状态的区块的记账人。拥有第二备选状态区块的记账人自己统计它收到了多少选票（即签名），因为它知道全网有多少备选记账人，所以它知道得到多少签名，自己就能进入第三状态。

这种算法比 PBFT 更简洁，不需要复杂的点对点通讯；同时，还可以避免像比特币一样经过漫长的等待才能最终确定交易。

2.2.4 链的形成

记账节点的区块进入第三备选状态之后，立即将这个区块广播出去。所有的一级节点都知道谁是备选记账人，所以所有的一级节点都能判断出一个区块是否真的能够进入第三备选状态。当一级节点判断自己收到的第三状态备选区块确实有效，就开始下一轮抽签，用自己的私钥签名这个区块的哈希值，看自己是否能成为备选记账人，这样就完成了一个区块链的链接过程。

2.3 其他关键技术

2.3.1 主要编程语言和开发平台

BTN 采用 Kotlin 作为主要编程语言，编译为 JVM 字节码运行在 Java 虚拟机上。

Java 提供了优秀的运行平台，但语法长时间没有较大的更新。Kotlin 是一个用于现代多平台应用的静态类型的编程语言，它比 Java 更安全，能够静态检测常见的陷阱。同时它比 Java 更简洁，支持 variable type inference，higher-order functions (closures)，extension function 和 first-class delegation。这会让我们更加安全高效的开发，也方便其他人加入 BTN 社区，改进基础代码或扩展 BTN。

2.3.2 网络通讯部分

BTN 采用 Netty 作为网络通讯的基础。Netty 是一个高性能、异步事件驱动的 NIO 框架，提供了对 TCP、UDP 的支持。作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于 Netty 构建，比如 RPC 框架、zookeeper 等。

Netty 的高性能、成熟可靠和灵活性，是 BTN 持数万 TPS 甚至更高的 TPS 的重要保障。

主节点启动代码如下：

```
fun main() {
    val mBoss = NioEventLoopGroup(1)
    val mWorker = NioEventLoopGroup(20)

    val bs = ServerBootstrap()
    try {
        bs.group(mBoss, mWorker)
            .channel(NioServerSocketChannel::class.java)
            .option(ChannelOption.SO_BACKLOG, 128)
            .option(ChannelOption.SO_REUSEADDR, true)
            .option(ChannelOption.SO_REUSEADDR, true)
            .childOption(ChannelOption.SO_KEEPALIVE, true)
            .childHandler(SupervisorChannelInitializer())

        val f = bs.bind(PRIMARY_PORT).sync()
        if (f.isSuccess) {
            println("Primary node is started")
            f.channel().closeFuture().addListener {
                mWorker.shutdownGracefully()
                mBoss.shutdownGracefully()
            }
        } else {
            mWorker.shutdownGracefully()
            mBoss.shutdownGracefully()
        }
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

2.3.3 通讯协议

BTN 使用 Json 作为底层通讯协议。上层封装为统一的对象化的命令 BtnCmd，由 BtnCmdEncoder 和 BtnCmdDecoder 负责编解码。

具体代码如下：

Part of the demo code as below:

```
open class BtnCmd(open val body:String ){
    val cmd:String
    val param:JsonObject?
    val paramString:String?
    init {
        val index = body.indexOf("{")
        if(index == -1){
            cmd = body
            paramString = ""
            param = null
        }else{
            cmd = body.substring(0,index)
            paramString =  body.substring(index)
            param = try{
                JsonParser().parse(paramString).asJsonObject
            }catch (e:Exception){
                e.printStackTrace()
                null
            }
        }
    }
    fun getStringParam(name:String):String?{
        return param?.get(name)?.asString
    }

    fun getLongParam(name:String):Long?{
        return param?.get(name)?.asLong
    }

    fun getIntParam(name:String):Int?{
        return param?.get(name)?.asInt
    }

    fun getJsonParam(name:String):JsonObject?{
```

```
35            return param?.get(name)?.asJsonObject
36        }
37        fun getBooleanParam(name:String):Boolean?{
38            return param?.get(name)?.asBoolean
39        }
40
41        override fun toString(): String {
42            return "BtnCMD:[$cmd]$body"
43        }
44        fun getJsonArray(s: String): JsonArray? {
45            return param?.getAsJsonArray(s)
46        }
47
48        fun getReturnCode(): Int? {
49            return getIntParam("r")
50        }
51    }
52
53    open class BtnCmdEncoder: MessageToByteEncoder<BtnCmd>() {
54        init {
55            enclog.info("BtnCmd Encoder Created")
56        }
57        override fun acceptOutboundMessage(msg: Any?): Boolean {
58            return msg is BtnCmd
59        }
60        override fun encode(ctx: ChannelHandlerContext?, msg: BtnCmd?, out: ByteBuf
          ?) {
61            enclog.info("try to encode message$msg")
62            if(out == null)
63                return
64
65            if(msg?.body == null){
66                enclog.info("read a message ,but body is null")
67                return
68            }
69
70            enclog.info("Encoding btncmd ${msg.body}")
71            out.writeInt(BTN_CMD_FLAG)
72            val bytes = msg.body.toByteArray(Charsets.UTF_8)
73            out.writeInt(bytes.size)
74            out.writeBytes(bytes)
75            enclog.info("Encode end")
76        }
```

```
77  }
78
79  open class BtnCmdDecoder:ChannelInboundHandlerAdapter(){
80      private enum class State {
81          INIT,
82          READ_COMMAND
83      }
84
85      private var state = State.INIT
86      private var buffer:ByteBuf? = null
87      private var byteRead = 0
88      private var messageLen = 0
89      private var readCount = 0
90      private var cmdId = 0
91
92      init {
93          this.resetState()
94      }
95      private fun resetState(){
96          this.buffer?.release()
97          this.buffer = null
98          this.byteRead = 0
99          this.state = State.INIT
100         this.messageLen = 0
101         this.readCount = 0
102     }
103
104     override fun channelRead(ctx: ChannelHandlerContext?, msg: Any?) {
105         if(ctx == null){
106             declog.info("read a message ,but context is null")
107             return
108         }
109
110         if(msg !is ByteBuf){
111             declog.info("read a message ,not byte buffer")
112             ctx.fireChannelRead(msg)
113             return
114         }
115
116         if(msg.refCnt() == 0){
117             declog.err("Command reference count = 0",Error("Wrong input message
    ,Command is not retained."))
118             return
```

```
119             }
120
121             while (true) {
122                 if(state == State.INIT){
123                     if(msg.readableBytes() < 8){
124                         ctx.fireChannelRead(msg)
125                         return
126                     }
127
128                     msg.markReaderIndex()
129                     declog.info("before try to read btn cmd flag")
130
131                     val flag = try{
132                         msg.readInt()
133                     }catch (e:java.lang.Exception){
134                         declog.err("message=${msg.hashCode()} channelId=${ctx.
        channel().id()}")
135                         declog.err("decode err",e)
136                         msg.release()
137                         throw e
138                     }
139
140                     declog.info("after read btn cmd flag")
141                     if(flag != BTN_CMD_FLAG){
142                         msg.resetReaderIndex()
143                         ctx.fireChannelRead(msg)
144                         return
145                     }
146
147                     declog.info("will change btncmd read state")
148                     messageLen = msg.readInt()
149
150                     state = State.READ_COMMAND
151                     cmdId++
152                     buffer = ctx.alloc().buffer(messageLen)
153                     if(buffer == null) {
154                         msg.release()
155                         throw Exception("Can not allocate buffer for BtnCmd")
156                     }
157
158                     declog.info("Ready for a new btn cmd=$cmdId")
159                 }
160
```

```
161                 var leftLen = messageLen - byteRead
162                 if(msg.readableBytes() <= leftLen)
163                     leftLen = msg.readableBytes()
164 
165                 readCount ++
166                 buffer?.writeBytes(msg,leftLen) //notice
167                 byteRead += leftLen
168                 if(byteRead >= messageLen){
169                     val btncmd = BtnCmd(buffer?.readCharSequence(messageLen,CHARSET
        ).toString())
170                     declog.info("read btn cmd finished cmd=${btncmd.body} remained
        bytes=${msg.readableBytes()}")
171                     ctx.fireChannelRead(btncmd)
172                     this.resetState()
173                 }
174 
175                 if (msg.readableBytes() == 0) {
176                     msg.release()  //notice wrong btng, not fired, must be released
177                     return
178                 }else{
179                     //notice continue until no byte.
180                 }
181             }
182         }
183 
184     }
185 
186     class BtnCmdCodec : CombinedChannelDuplexHandler<BtnCmdDecoder, BtnCmdEncoder
        >() {
187         init {
188             init(BtnCmdDecoder(), BtnCmdEncoder())
189         }
190     }
```

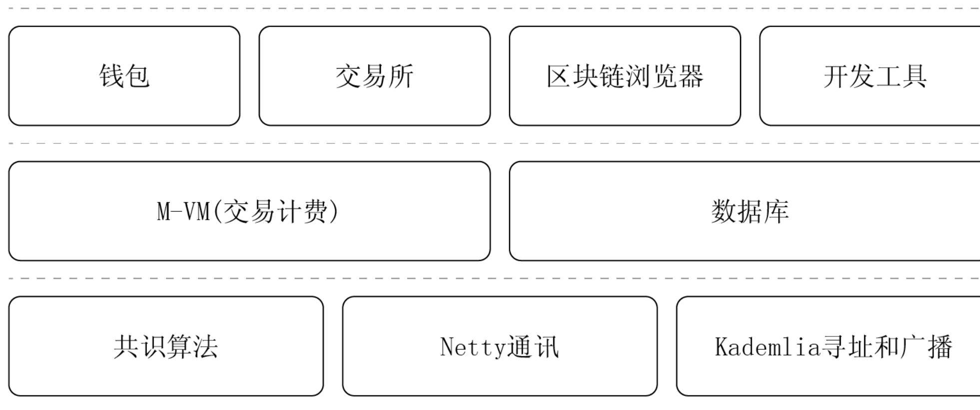
2.3.4 远程异步

在网络和多线程软件中，异步调用常常带来不少问题。等待异步结果，往往需要轮询、事件等手段。这回让代码变得很复杂，且容易出错。基于 Kotlin 提供的现代的语法特性和 Netty, BTN 团队设计了一种新型的支持 async/await 的远程调用方法。

代码如下：

```kotlin
private val requestMap = HashMap<String, CompletableDeferred<Any?>>()

suspend fun requestRemote(name:String, param:String? = null,paramIsString:
    Boolean = false):Any?{
        if(globalCtx == null)
            throw Exception("Global context is null, can not send remote
    request")

        val uuid = UUID.randomUUID().toString()
        val deferred = CompletableDeferred<Any?>()
        requestMap[uuid] = deferred
        var param1 = param
        if(param != null && paramIsString){
            param1 = "\"$param\""
        }

        val remoteRequest = """${ServerUtil.REMOTE_REQUEST}{"reqid":"$uuid","
    name":"$name", "param":$param1 }"""
        log.info("RemoteRequest=$remoteRequest")
        sendMsCmdStr(globalCtx, remoteRequest)

        val remoteResponse = withTimeoutOrNull(TimeUnit.SECONDS.toMillis(10L))
    {
            deferred.await()
        }
        requestMap.remove(uuid)
        if(remoteResponse == null){
            val remoteRequest1 = remoteRequest.replace("\"","'")
            return jsonResultString(2,"Remote api timeout. Remote request=
    $remoteRequest1")
        }

        return remoteResponse
    }
```

2.4 BTN 总体架构



3 市场优势

3.1 二级节点的数量优势

利用两级体制，我们可以部署庞大数量的二级节点，而不拖慢整个网络的性能。一二级节点的机制，容易组织销售力量。一级节点承担记账和中转的职责，放在机房内，由机构或者早期发起者持有，并保证服务质量，二级节点价格低廉，可以由一般的家庭用户持有，容易扩大数量。数量越多，我们的系统的验证者越多，公信力越高。

如果我们有 1000 个一级（记账和导流服务）节点，就一共可以有 100 万个二级（验证和签名）节点。有 10 万个一级节点，就有一亿个二级节点。没有一个其他公链可以和这个相比。因为对比特币和以太坊、EOS 这样的公链来说，节点越多，他们的通讯越复杂，处理能力会随之下降。

我们的一级也是鼓励扩容的。因为节点越多，存储的账目越多，越可靠。出了问题越能够选到备选节点，保证整个系统的可用性。同时由于记账池的概念，以及二级中转的概念，节点增加并不会导致处理速度降低。任何一个一级节点接到的记账请求，最多两级就记录了。

3.2 每个家用设备都有机会出块

挖矿不再需要矿场矿池。每个设备都有机会出块。用单个设备的私钥去给当前的块加密，看加密的结果是否符合特征值。可以有无数个设备竞争这个区块的生成，真正的分布式并发运算，这和 bitcoin 里面的 nonce 异曲同工，但这避免了作为投机者的矿机的出现。因为节点获得一次封印的机会，并不取决于它特定的运算能力，而之取决于它的身份信息，能够在 1 秒之内做 100 次运算和做一次运算并无显著差异，因为低轮次的封印优先级高于高轮次的封印。

3.3 核心优势总结

综上所述，结合技术和市场方法，Mass Chain 的核心优势如下：

1. 高度分散化，数量可以无限扩充
2. 保持高性能
3. 人人皆可参与，不需要机房，不需要矿机
4. 不会形成矿池和矿霸

4 风险

4.1 技术风险

4.1.1 作恶的方法、可能性和后果

任何一个想要交易的善节点，都很容易判断自己要提交交易的节点是善还是恶。核心就是，你不需要保证别人是好还是坏，只要保证自己是好的就行。

而交易发起者很清楚自己要站在那一边。步骤说明如下：

步骤	**比特币 （只有一级节点）**	**Mass Chain**	**后果、解决方案、备注**
发起请求	Full 钱包将请求提交给任意一级节点。 不区分善恶节点。善节点会广播交易。自己的转账、余额，会从自己验证过的链中获取。	二级节点的记账请求是直接发给记账人的。它可以鉴别记账人是否合法（根据区块和签名）。	发起请求这一步并不需要一级中转节点。 为什么我们比比特币多了鉴别这一步？因为我们不需要转发。
转发交易请求	收到交易的节点有可能是恶节点，可能不转发	恶一级中转节点有可能不转发，影响仅限于此	交易作废，没有进入有效的区块
记账：生成坏区块	把别人的钱划给自己 凭空创造 过高的手续费	同左	这些都是不合法区块，不会被认证，所以没有意义。
封印	对自己声称的坏区块进行 POW 封印	二级节点签名和投票委员会投票。 签名的二级节点、投票的一级节点，必须都是恶节点，否则无法出块。	无需 POW, 这是核心区别。 基于身份封印
后续交易	不会有善节点基于这个坏区块生成区块	不会有善节点基于这个生成区块	

4.1.2 质押机制

未来可以考虑结合质押机制。节点竞争记账或者封印，需要一定的质押，但量不需要太大，否则容易形成垄断。节点作恶，扣除它所质押的资产，从而进一步保证系统的可靠性。

4.1.3 Double spend

投票一致，没有 double spend 的问题。后续的区块没有能力翻盘。

4.1.4 女巫攻击

如果大量拥有节点，进行女巫攻击（伪装身份），恶意节点被选中的概率会提高。这个成立吗？如果成立，恶意节点可以做什么？对系统会产生什么样的影响？

凭空创造货币这件事情，其实比特币系统一样可以。只要连续作恶就可以。也就是大多数节点作恶，但这样的话，整个系统就没有价值了，失去了公信力。

一个诚实节点，其实很容易校验哪个是合法链，因为每个区块都是很容易校验的。所以好节点可以继续的生成好链。一个节点如果想要校验，只要校验它当前收到的区块就可以了。那些坏节点如果要作恶，可以在他们的坏链上持续作恶，但好链不论长短，会一直存在并生长。好一级节点会基于好链，好区块，继续生成下一个区块。

4.2 竞争风险：与 Bitcoin, ETH, EOS 的比较

	POE	Bitcoin/ETH(POW)	EOS(POS)
记账速度	✓ 委员会记账，TPS 接近 EOS	✗ 全网记账	✓ 委员会记账
分叉	✓ 无，投票解决	6 个块确认	✓ 无，PBFT
真实性，可靠性	✓ 基于随机数，网络越大越真实，而且验证是全网验证	✓ 基于 POW，全网验证	✗ 验证机制不透明，争议较多
网络基数与选举制	✓ 越大越好，越快越稳定。用可靠的随机机制抽取	✗ 越大越慢，全网记账，全网算力竞争	选举机制不透明，POS 难自圆其说
参与者	✓ 一二级网络，二级可以使家庭用户	✗ 家用不行，不能 p2p 组网	✗ 不能 p2p 组网
垄断性	真人身份认证，TPM 设备	矿场、矿池	委员会垄断，不透明
销售	一级私募，二级群众参与	矿场投资	私募，小范围
广播次数	2 次（竞选和成功广播）	账目广播和区块广播	账目无广播，区块广播

4.3 组织和资金风险

BTN 想要取得成功，需要充分的准备，一旦开始启动，就要组织足够的资金和市场力量，迅速达成共识，并形成一定的规模，从而建立自己的护城河，这种领先地位一旦形成，将具有非常强大的自我生长的力量，持续保持我们技术和规模以及组织上的优势。

5 展望

5.1 完整应用

本文中的内容，仅覆盖了核心共识算法、系统架构和组织机制，还没有包含虚拟机、钱包、区块链浏览器、交易所等内容，这些会随着生态系统的成长逐步完善，最终 BTN 会成长为超过以太坊的，一套真正的完整独立的世界计算机。

5.2 生态系统

Mass Chain 除了区块链应用之外，它所具有的吸纳二级节点的特性，还可以让我们有打造生态系统的机会。我们可以用区块链为突破口和基石，建立内容、发行、私有云、边缘计算、分布式计算和存储等生态系统。

我们的家用计算设备，除了能够校验和封印一级节点的区块，还可以作为一个正常的智能终端来使用，因为如前所述，运算并不是二级节点的核心任务，二级节点有足够的闲散算力和存储空间，做其他的应用。包括但不限于如下可能：

- 家庭智能终端：例如私有云、智能家庭主机
- 内容分发平台：利用二级节点提供影视等内容的分发
- 分布式计算：利用二级节点做诸如图像识别、大数据存储和检索等应用
- 分布式存储和带宽共享：我们数量庞大的二级节点和一级节点，在不影响核心业务的情况下，可以承载部分分布式存储和宽带共享业务
- 边缘网络：我们的二级节点具有 24 小时开机、自带存储等特性，非常适合作为家庭中的边缘存储设备，服务于家庭中的电视、手机、pad 等

另外，如果网络足够大，我们地位足够高，还可以虚拟的分解出来若干子网生成子链承载特定业务，更客制化的满足客户或者行业的需求。

6 答疑

6.1 量子计算

现有的绝大多数公钥密码算法（RSA、Diffie-Hellman、椭圆曲线等）能被足够大和稳定的量子计算机攻破，需要考虑量子计算的潜在影响。量子计算机很强大，但利用其强大算力的前提是：存在能高效解决问题的量子算法，否则量子计算机没什么用，反而因为其高昂的成本带来劣势。

美国国家标准技术研究所 (NIST) 早在 2012 年启动了后量子密码的研究工作，并于 2016 年 2 月启动了全球范围内的后量子密码标准征集。主要包括以下 4 种数学方法构造的后量子密码算法：

- 格 (Lattice-based)
- 编码 (Code-based)
- 多变量 (Multivariate-based)
- 哈希 (Hash-based)

在新的算法成为工业标准后，可以在保持整个系统架构不变的情况下，替换系统的 RSA 算法，保持所有的历史数据和业务逻辑不变。

关于对称密码算法和哈希函数（例如 AES、SHA1、SHA2 等），虽然有量子算法可以理论上攻破，但这个算法的影响有限，且有很多限制条件。著名的量子算法是 1996 年的 Grover's algorithm。这一部分把密钥长度或哈希的长度加倍即可，例如：AES-128 升级至 AES-256，SHA-256 升级至 SHA-512 等。

6.2 女巫攻击和 TPM

在对第二备选状态的区块封印的过程中，如果采用文件证书，恶节点可以用自己收藏的大量的合法证书对此区块进行封印，从而获取封印收益，这类似于女巫攻击。

可以用 TPM 来控制这种行为。因为 TPM 作为硬件，被私自制造的可能性太低。所有的 TPM 必须通过根节点的备案激活才生效。TPM 与量子计算的关系参见上一节。

TPM 不需要立即实施，因为所有的文件证书，依然是不可伪造的合法证书。作恶者需要花费大量成本获取这些证书。

6.3 TPS

区块链的核心是可靠，而不是 TPS，但 TPS 可以通过侧链和抵押机制解决。

6.4 未能将交易及时打包

如果一个二级节点发出了一个记账请求，而最终成功获得签名的一级节点没有及时打包，可以让请求者重新发送。

记账节点必须有一定数量，否则可能导致一些账目丢失。交给一个节点记账是不行的。

7 术语

- **TPS:** Trade per second 每秒交易数量
- **UTXO:** Unspent output from bitcoin transactions
- **TPM:** Trusted platform module

参考文献

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin. org/bitcoin. pdf. 2008.

[2] E. Foundation. Ethereum’s white paper. Technical report, https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. Technical report, Cryptology ePrint Archive, Report 2019/514, 2019. https://eprint.iacr.org/2019/514.

[4] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin’s peer-to-peer network. In *24th Security Symposium Security 15)*, pages 129–144, 2015.

[5] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd Security Symposium (Security 14)*, pages 781–796, 2014.

[7] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. *IACR Cryptology ePrint Archive*, 2018:377, 2018.

[8] Ethash. Ethereum’s white paper. Technical report, https://github.com/ethereum/wiki/wiki/Ethash, Accessed on June 27, 2017., version 23.

[9] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[10] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[11] SECG SEC. 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.

[12] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th security symposium (security 16)*, pages 279–296, 2016.

[13] Daniel J Bernstein. Multi-user schnorr security, revisited. *IACR Cryptology ePrint Archive*, 2015:996, 2015.

[14] Andrew Poelstra. Schnorr signatures are non-malleable in the random oracle model, 2014.