区块链作业

姓名：裴启智

学号：PB18111793

1、Merkle 树

请用伪代码实现下列算法：交易序列的Merkle树的生成，SPV证明的生成和验证。

附：Merkle 树 T 的根为 T.root，每个节点 x 包含 p，left，right，data 四个属性。其中 p，left，right 分别为指向父节点，左子树，右子树的指针，data 存储了该节点的哈希值。
一个包含三笔交易的 Merkle 树示例见图 1，其中使用了 SHA256 作为哈希函数，⊕表示拼接操作。

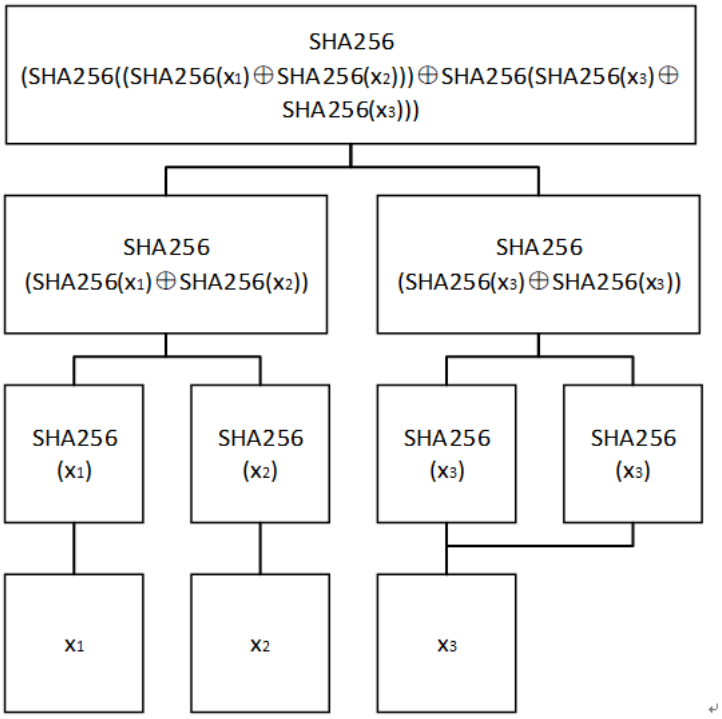


图 1 一个二叉 Merkle 树的示例

1.1

1.1、**生成给定交易序列的二叉Merkle树。**输入交易序列 $[x_1, x_2, \ldots, x_n]$，长度为n，输出该交易序列所对应的Merkle树T。

```
// Transaction 代表交易序列，是一个二维切片
// Transaction[i] = xi

func NewMerkleTree(Transaction [][]byte) *MerkleTree {
    var nodes []MerkleNode
    // 保证叶子节点为偶数
    if len(data) % 2 != 0 {
        // 复制最后一份数据
        data = append(data, data[len(data) - 1])
    }
```

```
    // 将所有数据读入所有叶子结点
    for _, dataTemp := range data {
        node := NewMerkleNode(nil, nil, dataTemp)
        nodes = append(nodes, *node)
    }

    // 逐层合并，由下到上，生成内部结点
    for i := 0; i < len(data) / 2; i++ {
        var treeLevel []MerkleNode
        for j := 0; j < len(nodes); j += 2 {
            node := NewMerkleNode(&nodes[j], &nodes[j+1], nil)
            treeLevel = append(treeLevel, *node)
        }
        // 每轮循环中都对 nodes 赋值，最终nodes只含有 Merkle 树的根结点
        nodes = treeLevel
    }

    var T = MerkleTree{&nodes[0]}
    return &T
}
```

1.2

1.2、**生成某笔交易的Merkle树证明。**输入Merkle树T，以及某笔交易x_i的序号i，输出该交易所对应的SPV证明路径$[H_1, H_2, \ldots, H_m]$，长度为m。

```
// 获得传入 node 的兄弟节点
func getBrotherNode(node *MerkleNode) *MerkleNode {
    if node.p == nil {
        return nil
    } else if node == node.p.left {
        return node.p.right
    } else {
        return node.p.left
    }
}

// 证明路径最终存储在 tempHash 中，包括了当前交易 xi 对应的 Hash 值
func MerklePath(i int, T *MerkleNode) {
    var result *MerkleNode
    // 获得 xi 对应的 Merkle 节点
    result = findMerkleNode(x[i])

    var tempHash [][]byte
    tempHash = append(tempHash, result.Data)
    // 自底向上填充证明路径
    for getBrotherNode(h) != nil {
        tempHash = append(tempHash, getBrotherNode(h).Data)
        h = h.Parent
    }

    for i := 0; i < len(tempHash); i++ {
```

```
        fmt.Print(tempHash[i])
    }

}
```

1.3

1.3、 **验证SPV证明。**输入Merkle树的根哈希值$\mathrm{merkle_root}$和SPV证明路径$[H_1, H_2, \ldots, H_m]$，长度为m，输出该SPV证明路径是否合法（true或false)。

```
// 证明路径存储在 tempHash 中
func Validate(merkle_root *MerkleNode, tempHash [][]byte) {
    for i := 0; i < len(tempHash); i++ {
        hash := sha256.Sum256(append(result, tempHash[i]...))
        result = hash[:]
    }

    return bytes.Equal(result, merkle_root.Data)
}
```

2、简述比特币钱包公私钥生成原理，并给出比特币钱包地址生成的伪代码实现。（注：分别使用助记词和私钥）

公钥加密与比特币地址

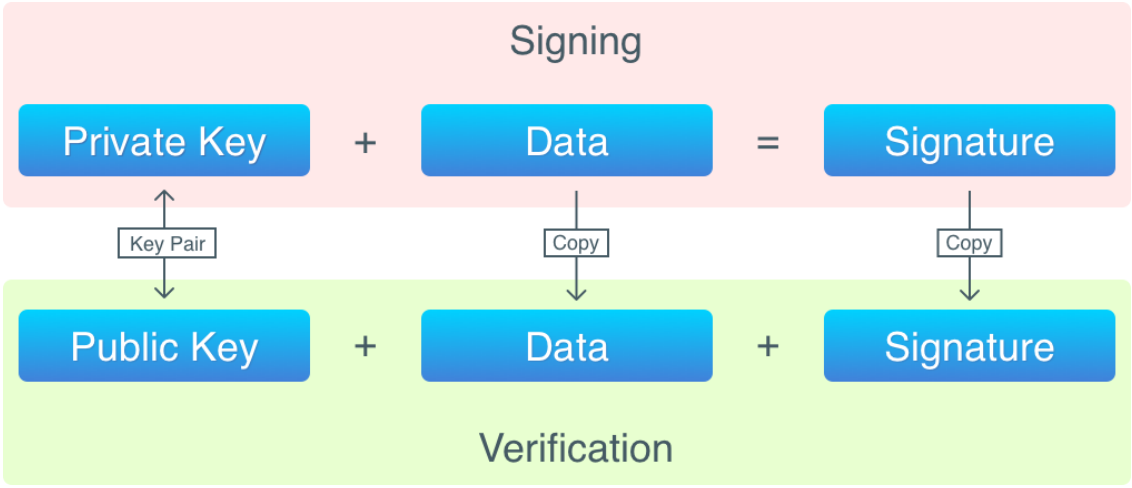
- 在比特币中，身份（identity）就是一对（或者多对）保存在电脑（或者你能够获取到的地方）上的公钥（public key）和私钥（private key）
- 所谓的比特币钱包地址，只不过是将公钥表示成人类可读的形式而已 。而比特币钱包本质上就是公私钥密钥对，他们通过公钥加密算法生成。在比特币中，谁拥有了私钥，谁就可以控制所有发送到这个公钥的币。也就是说私钥用来证明用户身份

助记词

- 比特币钱包应用很可能会为你生成一个**助记词**。助记词可以用来替代私钥，并且可以被用于生成私钥
- 助记词可以理解为私钥的另一种简单表示，最初是BIP39提案提出的，它有助于用户记住复杂的私钥(64位的哈希值)，并且具有与私钥相同的功能。记住64位随机数基本上是不可能的，因此助词有助于钱包用户有效地使用和支配自己的资产。助词一般由12、15、18、21个单词组成，这些单词都来自固定词库，生成顺序也是按照一定的算法生成的，所以用户没必要担忧随随便便地输入 12 个单词，就会生成一个地址。
- 助记词很重要，因为能通过助记词找到私钥，还能恢复钱包。目前，大部分钱包都需要备份助记词，以恢复钱包。

数字签名

- 数字签名保证
  - 当数据从发送方传送到接收方时，数据不会被修改
  - 数据由某一确定的发送方创建
  - 发送方无法否认发送过数据这一事实
- 通过在数据上应用签名算法（也就是对数据进行签名），你就可以得到一个签名，这个签名晚些时候会被验证。生成数字签名需要一个私钥，而验证签名需要一个公钥。
- 为了对数据进行签名，需要 要签名的数据数据和私钥，然后应用签名算法，且这个签名会被存储在交易输入中
- 为了对一个签名进行验证 ，需要 被签名的数据、签名、公钥。验证过程可以被描述为：检查签名是由被签名数据加上私钥得来，并且公钥恰好是由该私钥生成
  - 数据签名不是加密，因为无法从签名重构出数据
  - 签名和哈希的区别在于密钥对：有了密钥对，才有签名验证
  - 但是密钥对也可以被用于加密数据：私钥用于加密，公钥用于解密数据。不过比特币并不使用加密算法
- 在比特币中，每一笔交易输入都会由创建交易的人签名 。对数据进行签名和对签名进行验证的过程大致如下

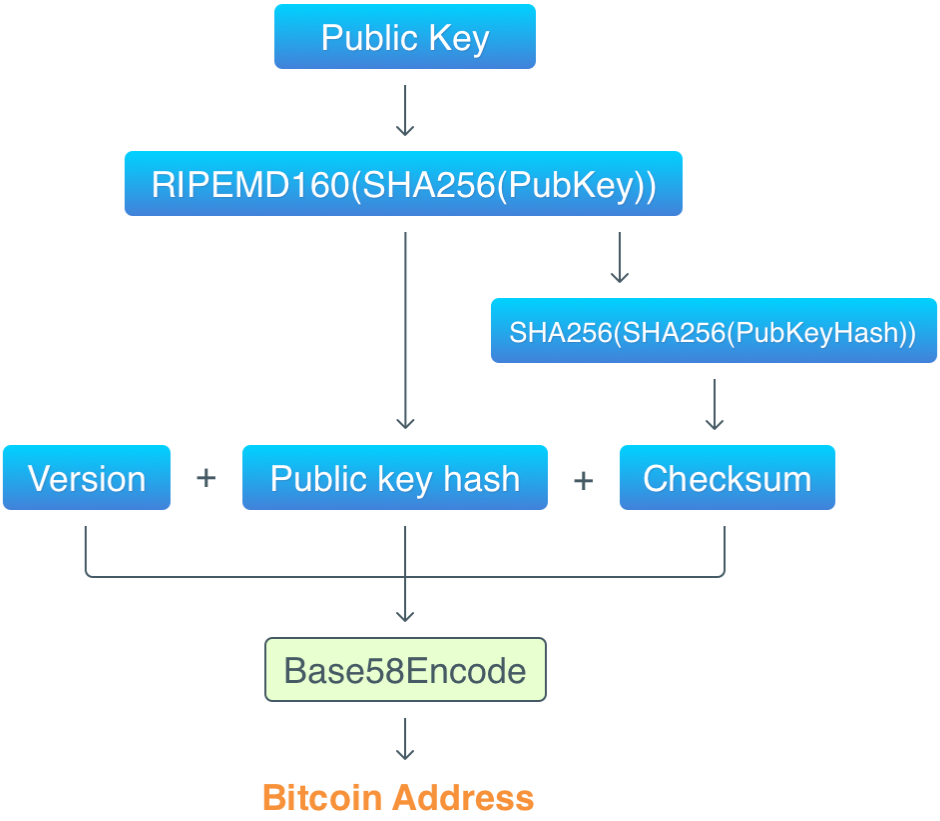


椭圆曲线加密

- 公钥和私钥是随机的字节序列。私钥能够用于证明持币人的身份，需要有一个条件：随机算法必须生成真正随机的字节。因为没有人会想要生成一个私钥，而这个私钥意外地也被别人所有。
- 比特币使用的是 ECDSA（Elliptic Curve Digital Signature Algorithm）算法来对交易进行签名

Base58

- 比特币使用 Base58 算法将公钥转换成人类可读的形式。
- 从一个公钥获得地址的过程



- 由于哈希函数是单向的，所以不可能从一个哈希中提取公钥。不过通过执行哈希函数并进行哈希比较，我们可以检查一个公钥是否被用于哈希的生成。

比特币钱包地址生成算法

- 比特币钱包结构及构造函数，这里用到了私钥

```
type Wallet struct {
    PrivateKey ecdsa.PrivateKey
    PublicKey  []byte
}

func NewWallet() *Wallet {
    private, public := newKeyPair()
    wallet := Wallet{private, public}

    return &wallet
}

func newKeyPair() (ecdsa.PrivateKey, []byte) {
    // 获得一个椭圆曲线
    curve := elliptic.P256()
    // 通过椭圆曲线生成一个私钥
    private, err := ecdsa.GenerateKey(curve, rand.Reader)
    // 通过私钥生成一个公钥
    // 公钥是曲线上的点。因此，公钥是 X，Y坐标的组合。
    // 在比特币中，这些坐标会被连接起来，然后形成一个公钥。
    pubKey := append(private.PublicKey.X.Bytes(),
private.PublicKey.Y.Bytes()...)

    return *private, pubKey
}
```

- 将公钥转换成比特币地址

```
func (w Wallet) GetAddress() []byte {
    // 使用 RIPEMD160(SHA256(PubKey)) 哈希算法，取公钥并对其哈希两次
    pubKeyHash := HashPubKey(w.PublicKey)
    //  给哈希加上地址生成算法版本的前缀
    versionedPayload := append([]byte{version}, pubKeyHash...)
    // 对于前面生成的结果，使用 SHA256(SHA256(payload)) 再哈希，计算校验和。校验和是
结果哈希的      前四个字节。
    checksum := checksum(versionedPayload)
    // 将校验和附加到 version+PubKeyHash 的组合中。
    fullPayload := append(versionedPayload, checksum...)
    // 使用 Base58 对 version+PubKeyHash+checksum 组合进行编码
    address := Base58Encode(fullPayload)

    return address
}

func HashPubKey(pubKey []byte) []byte {
    publicSHA256 := sha256.Sum256(pubKey)

    RIPEMD160Hasher := ripemd160.New()
    _, err := RIPEMD160Hasher.Write(publicSHA256[:])
    publicRIPEMD160 := RIPEMD160Hasher.Sum(nil)

    return publicRIPEMD160
}

func checksum(payload []byte) []byte {
    firstSHA := sha256.Sum256(payload)
    secondSHA := sha256.Sum256(firstSHA[:])

    return secondSHA[:addressChecksumLen]
}
```

函数`GetAddress`返回的地址即为比特币地址

3、简述区块链 P2P 网络协议中Gossip（比特币）和Kademlia（以太坊）原理，给出相关示意图，并加以比较。

Gossip

- Gossip协议是一个通信协议，一种传播消息的方式，灵感来自于：瘟疫、社交网络等。

原理

- Gossip 过程是由种子节点发起的。一个节点想要分享一些信息给网络中的其他的一些节点。于是，它**周期性**的**随机**选择一些节点，并把信息传递给这些节点。这些收到信息的节点接下来会做同样的事情，即把这些信息传递给其他一些随机选择的节点。一般而言，信息会周期性的传递给N个目标节点，而不只是一个。这个N被称为**fanout**
- 这个过程可能需要一定的时间，由于不能保证某个时刻所有节点都收到消息，但是理论上最终所有节点都会收到消息，因此它是一个最终一致性协议。

用途

Gossip协议的主要用途就是**信息传播和扩散**：即把一些发生的事件传播到全世界。它们也被用于数据库复制，信息扩散，集群成员身份确认，故障探测等。

Gossip类型

Gossip 有两种类型：

- Anti-Entropy（反熵）：以固定的概率传播所有的数据
- Rumor-Mongering（谣言传播）：仅传播新到达的数据

Anti-Entropy 是 SI model，节点只有两种状态，Suspective 和 Infective，叫做 simple epidemics。

Rumor-Mongering 是 SIR model，节点有三种状态，Suspective，Infective 和 Removed，叫做 complex epidemics。

其实，Anti-entropy 反熵是一个很奇怪的名词，之所以定义成这样，Jelasity 进行了解释，因为 entropy 是指混乱程度（disorder），而在这种模式下可以消除不同节点中数据的 disorder，因此 Anti-entropy 就是 anti-disorder。换句话说，它可以提高系统中节点之间的 similarity。

在 SI model 下，一个节点会把所有的数据都跟其他节点共享，以便消除节点之间数据的任何不一致，它可以保证最终、完全的一致。

由于在 SI model 下消息会不断反复的交换，因此消息数量是非常庞大的，无限制的（unbounded），这对一个系统来说是一个巨大的开销。

但是在 Rumor Mongering（SIR Model） 模型下，消息可以发送得更频繁，因为消息只包含最新 update，体积更小。而且，一个 Rumor 消息在某个时间点之后会被标记为 removed，并且不再被传播，因此，SIR model 下，系统有一定的概率会不一致。

而由于，SIR Model 下某个时间点之后消息不再传播，因此消息是有限的，系统开销小。

Gossip 通信模式

在 Gossip 协议下，网络中两个节点之间有三种通信方式:

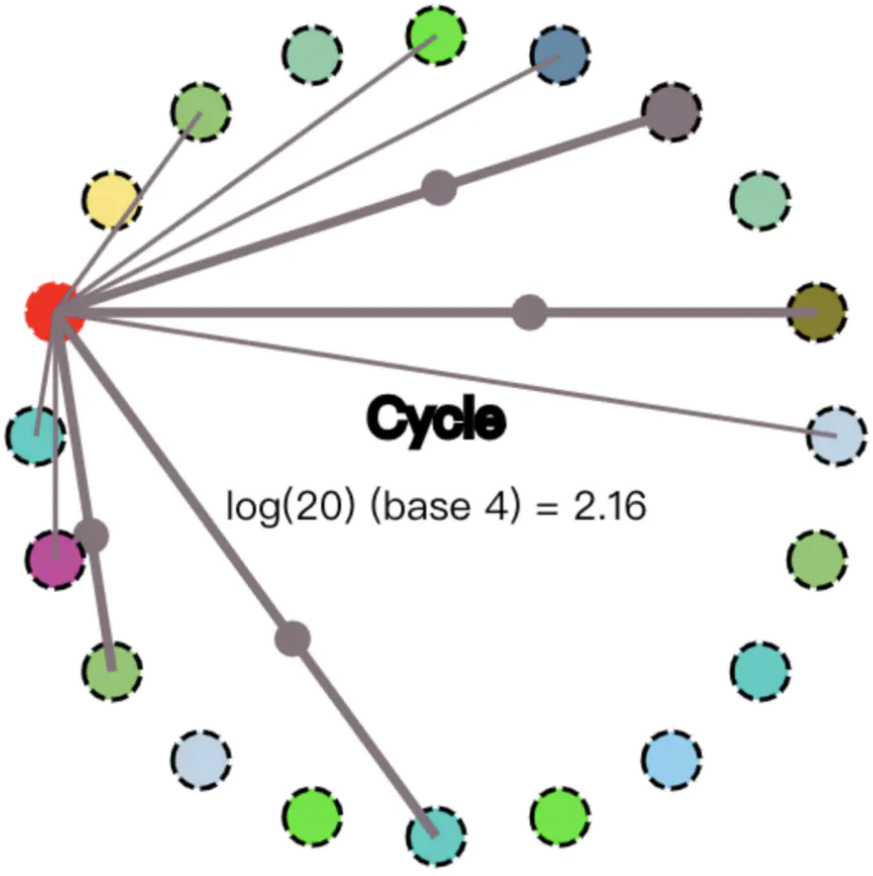
- Push: 节点 A 将数据 (key,value,version) 及对应的版本号推送给 B 节点，B 节点更新 A 中比自己新的数据
- Pull：A 仅将数据 key, version 推送给 B，B 将本地比 A 新的数据（Key, value, version）推送给 A，A 更新本地
- Push/Pull：与 Pull 类似，只是多了一步，A 再将本地比 B 新的数据推送给 B，B 则更新本地

如果把两个节点数据同步一次定义为一个周期，则在一个周期内，Push 需通信 1 次，Pull 需 2 次，Push/Pull 则需 3 次。虽然消息数增加了，但从效果上来讲，Push/Pull 最好，理论上一个周期内可以使两个节点完全一致。直观上，Push/Pull 的收敛速度也是最快的。

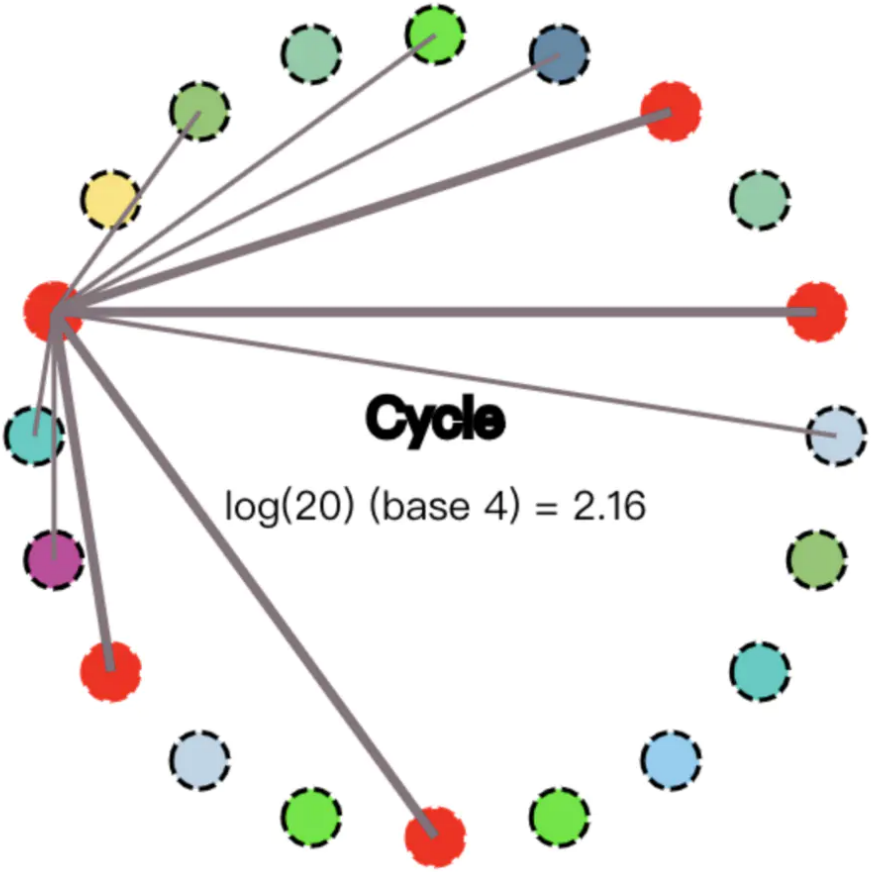
图解

如下图所示，Gossip协议是周期循环执行的。图中的公式表示Gossip协议把信息传播到每一个节点需要多少次循环动作，需要说明的是，公式中的20表示整个集群有20个节点，4表示某个节点会向4个目标节点传播消息：

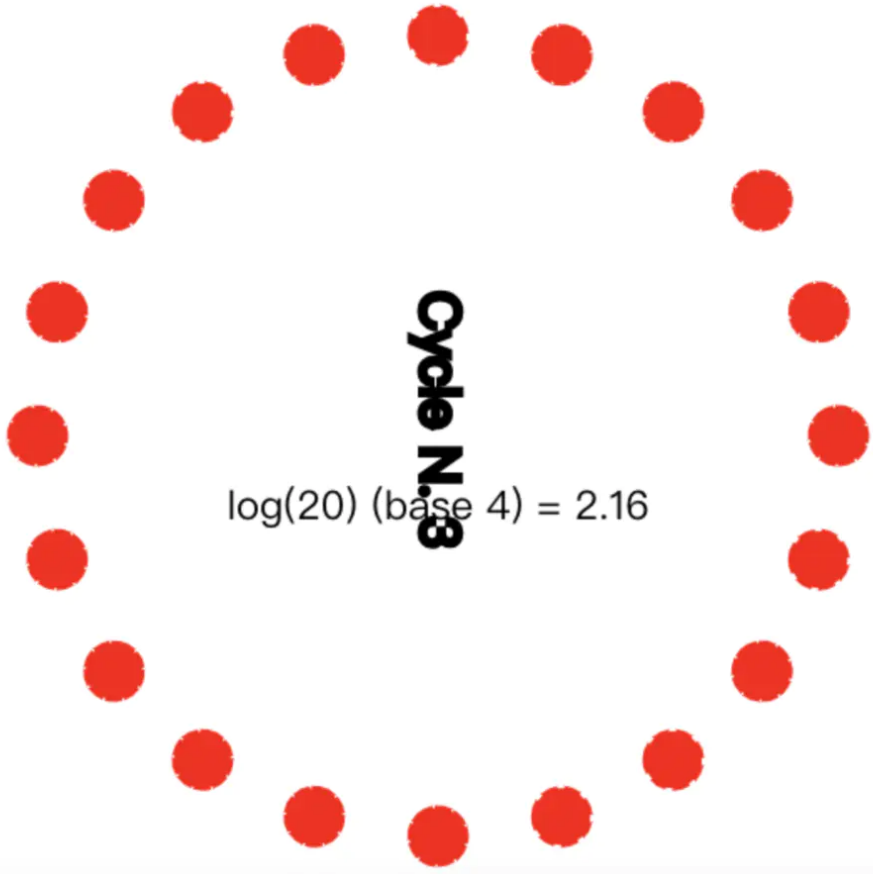
如下图所示，红色的节点表示其已经“受到感染”，即接下来要传播信息的源头，连线表示这个初始化感染的节点能正常连接的节点（其不能连接的节点只能靠接下来感染的节点向其传播消息）。并且N等于4，我们假设4根较粗的线路，就是它第一次传播消息的线路：



第一次消息完成传播后，新增了4个节点会被“感染”，即这4个节点也收到了消息。这时候，总计有5个节点变成红色：



那么在下一次传播周期时，总计有5个节点，且这5个节点每个节点都会向4个节点传播消息。最后，经过3次循环，20个节点全部被感染（都变成红色节点），即说明需要传播的消息已经传播给了所有节点：



需要说明的是，20个节点且设置fanout=4，公式结果是2.16，这只是个近似值。真实传递时，可能需要3次甚至4次循环才能让所有节点收到消息。这是因为每个节点在传播消息的时候，是随机选择N个节点的，这样的话，就有可能某个节点会被选中2次甚至更多次。

发送消息

- 节点传播消息是周期性的，并且**每个节点有它自己的周期**。另外，节点发送消息时的**目标节点数**由参数fanout决定。至于往哪些目标节点发送，则是**随机**的。
- 一旦消息被发送到目标节点，那么目标节点也会被感染。一旦某个节点被感染，那么它也会向其他节点传播消息，试图感染更多的节点。最终，每一个节点都会被感染，即消息被同步给了所有节点

优点

异步消息传递

Gossip 过程是异步的，也就是说发消息的节点不会关注对方是否收到，即不等待响应；不管对方有没有收到，它都会每隔一定时间向周围节点发消息。

可扩展性

Gossip协议是可扩展的，因为它只需要O(logN) 个周期就能把消息传播给所有节点。某个节点在往固定数量节点传播消息过程中，并不需要等待确认（ack），并且，即使某条消息传播过程中丢失，它也不需要做任何补偿措施。比方说，某个节点本来需要将消息传播给4个节点，但是由于网络或者其他原因，只有3个消息接收到消息，即使这样，这对最终所有节点接收到消息是没有任何影响的。

如下表格所示，假定fanout=4，那么在节点数分别是20、40、80、160时，消息传播到所有节点需要的循环次数对比，在节点成倍扩大的情况下，循环次数并没有增加很多。所以，Gossip协议具备可扩展性：

节点数	20	40	80	160	320
循环次数	2.16	2.66	3.16	3.44	4.16

容错能力

Gossip也具备失败容错的能力，即使网络故障等一些问题，Gossip协议依然能很好的运行。因为一个节点会**多次**分享某个需要传播的信息，即使不能连通某个节点，其他被感染的节点也会尝试向这个节点传播信息。

健壮性

Gossip协议下，没有任何扮演特殊角色的节点（比如leader等）。任何一个节点无论什么时候下线或者加入，并不会破坏整个系统的服务质量。

去中心化

Gossip 协议不要求任何中心节点，所有节点都可以是对等的，任何一个节点无需知道整个网络状况，只要网络是连通的，任意一个节点就可以把消息散播到全网。

一致性收敛

Gossip 协议中的消息会以一传十、十传百一样的指数级速度在网络中快速传播，因此系统状态的不一致可以在很快的时间内收敛到一致。消息传播速度达到了 logN。

简单

Gossip 协议的过程极其简单，实现起来几乎没有太多复杂性。

```
do forever
  wait(T time units)
  p ← selectPeer()
  if push then
    // 0 is the initial hop count
    myDescriptor ← (myAddress, 0)
    buffer ← merge(view,{myDescriptor})
    send buffer to p
  else
    // empty view to trigger response
    send {} to p
  if pull then
    receive view_p from p
    view_p ← increaseHopCount(view_p)
    buffer ← merge(view_p,view)
    view ← selectView(buffer)
```

(a) active thread

```
do forever
  (p, view_p) ← waitMessage()
  view_p ← increaseHopCount(view_p)
  if pull then
    // 0 is the initial hop count
    myDescriptor ← (myAddress, 0)
    buffer ← merge(view,{myDescriptor})
    send buffer to p
  buffer ← merge(view_p,view)
  view ← selectView(buffer)
```

(b) passive thread

缺陷

拜占庭问题（Byzantine）

如果有一个恶意传播消息的节点，Gossip协议的分布式系统就会出问题。

消息延迟

由于 Gossip 协议中，节点只会随机向少数几个节点发送消息，消息最终是通过多个轮次的散播而到达全网的，因此使用 Gossip 协议会造成不可避免的消息延迟。不适合用在对实时性要求较高的场景下。

消息冗余

Gossip 协议规定，节点会定期随机选择周围节点发送消息，而收到消息的节点也会重复该步骤，因此就不可避免的存在消息重复发送给同一节点的情况，造成了消息的冗余，同时也增加了收到消息的节点的处理压力。而且，由于是定期发送，因此，即使收到了消息的节点还会反复收到重复消息，加重了消息的冗余。

Kademlia

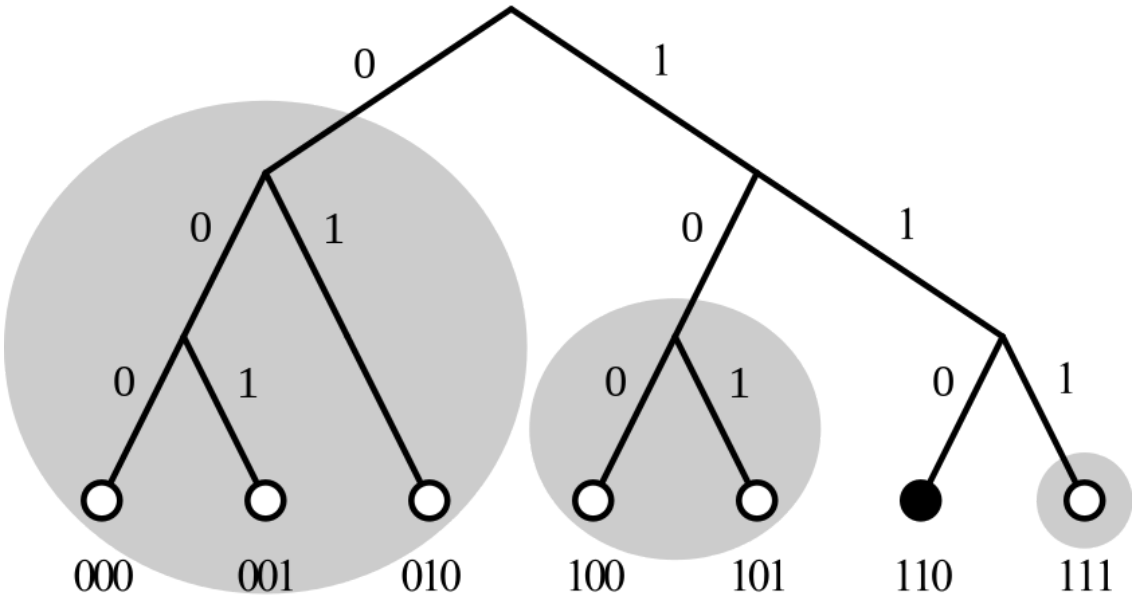
- 以太坊的 P2P 网络主要采用了 **Kademlia(简称 Kad)** 算法实现。
    - Kad 是一种**分布式哈希表(DHT)**技术，使用该技术，可以实现在分布式环境下快速而又准确地路由、定位数据的问题。
    - Kademlia规定了网络的结构，也规定了通过节点查询进行信息交换的方式。Kademlia网络节点之间使用UDP进行通讯。参与通讯的所有节点形成一张虚拟网（或者叫做覆盖网）。这些节点通过一组数字（或称为节点ID）来进行身份标识。节点ID不仅可以用来做身份标识，还可以用来进行值定位（值通常是文件的散列或者关键词）。其实，节点ID与文件散列直接对应，它所表示的那个节点存储着哪儿能够获取文件和资源的相关信息。当我们在网络中搜索某些值（即通常搜索存储文件散列或关键词的节点）的时候，Kademlia算法需要知道与这些值

相关的键，然后分步在网络中开始搜索。每一步都会找到一些节点，这些节点的ID与键更为接近，如果有节点直接返回搜索的值或者再也无法找到与键更为接近的节点ID的时候搜索便会停止。这种搜索值的方法是非常高效的：与其他的分散式杂凑表的实现类似，在一个包含n个节点的系统的值的搜索中，Kademlia仅访问O(log(n))个节点。非集中式网络结构还有更大的优势，那就是它能够显著增强抵御拒绝服务攻击的能力。即使网络中的一整批节点遭受泛洪攻击，也不会对网络的可用性造成很大的影响，通过绕过这些漏洞（被攻击的节点）来重新编织一张网络，网络的可用性就可以得到恢复。

- Kademlia基于两个节点之间的距离计算，该距离是两个网络节点ID号的异或（ XOR distance ），计算的结果最终作为整型数值返回。关键字和节点ID有同样的格式和长度，因此，可以使用同样的方法计算关键字和节点ID之间的距离。节点ID一般是一个大的随机数，选择该数的时候所追求的一个目标就是它的唯一性（希望在整个网络中该节点ID是唯一的）。异或距离跟实际上的地理位置没有任何关系，只与ID相关。因此很可能来自德国和澳大利亚的节点由于选择了相似的随机ID而成为邻居。选择异或是因为通过它计算的距离享有几何距离公式的一些特征，尤其体现在以下几点：节点和它本身之间的异或距离是0；异或距离是对称的：即从A到B的异或距离与从B到A的异或距离是等同的；异或距离符合三角不等式：三个顶点A B C，AC异或距离小于或等于AB异或距离和BC异或距离之和。由于以上的这些属性，在实际的节点距离的度量过程中计算量将大大降低。Kademlia搜索的每一次迭代将距目标至少更近1 bit。一个基本的具有2的n次方个节点的Kademlia网络在最坏的情况下只需花n步就可找到被搜索的节点或值。
    - 异或有一个重要的性质：假设 a、b、c 为任意三个数，如果 a^b = a^c 成立，那就一定 b = c。因此，如果给定一个结点 a 和距离 L，那就有且仅有一个结点 b, 会使得 D(a,b) = L。通过这种方式，就能有效度量 Kad 网络中不同节点之间的逻辑距离。

路由表

- Kademlia路由表由多个列表组成，每个列表对应节点ID的一位（例如：假如节点ID共有128位，则节点的路由表将包含128个列表），包含多个条目，条目中包含定位其他节点所必要的一些数据。列表条目中的这些数据通常是由其他节点的IP地址，端口和节点ID组成。每个列表对应于与节点相距特定范围距离的一些节点，节点的第n个列表中所找到的节点的第n位与该节点的第n位肯定不同，而前n-1位相同，这就意味着很容易使用网络中远离该节点的一半节点来填充第一个列表（第一位不同的节点最多有一半），而用网络中四分之一的节点来填充第二个列表（比第一个列表中的那些节点离该节点更近一位），依次类推。如果ID有128个二进制位，则网络中的每个节点按照不同的异或距离把其他所有的节点分成了128类，ID的每一位对应于其中的一类。随着网络中的节点被某节点发现，它们被逐步加入到该节点的相应的列表中，这个过程中包括向节点列表中存信息和从节点列表中取信息的操作，甚至还包括当时协助其他节点寻找相应键对应值的操作。这个过程中发现的所有节点都将被加入到节点的列表之中，因此节点对整个网络的感知是动态的，这使得网络一直保持着频繁地更新，增强了抵御错误和攻击的能力。
- 在**Kademlia**相关的论文中，列表也称为K桶，其中K是一个系统变量，如20，每一个K桶是一个最多包含K个条目的列表，也就是说，网络中所有节点的一个列表（对应于某一位，与该节点相距一个特定的距离）最多包含20个节点。随着对应的bit位变低（即对应的异或距离越来越短），K桶包含的可能节点数迅速下降（这是由于K桶对应的异或距离越近，节点数越少），因此，对应于更低bit位的K桶显然包含网络中所有相关部分的节点。由于网络中节点的实际数量远远小于可能ID号的数量，所以对应那些短距离的某些K桶可能一直是空的（如果异或距离只有1，可能的数量就最大只能为1，这个异或距离为1的节点如果没有发现，则对应于异或距离为1的K桶则是空的）。
- 在异或距离度量的基础上，Kad 还可以将整个网络拓扑组织成如下图所示的一个**二叉前缀树**，每个 NodeID 会映射到二叉树上的某个叶子



映射规则主要是：

1. 将 NodeID 以二进制形式表示，然后从高到低对每一位的 0 或 1 依次处理；
2. 二进制的第 n 位就对应了二叉树的第 n 层；
3. 如果该位是 0，进入左子树，是 1 则进入右子树（反过来也可以）；
4. 全部位都处理完后，这个 NodeID 就对应了二叉树上的某个叶子。

- 在这种二叉树结构下，对每个节点来说，离它越近的节点异或距离也是越近的。接着，可以按照离自己异或距离的远近，对整颗二叉树进行拆分。拆分规则是：从根节点开始，将不包括自己的那颗子树拆分出来，然后在包含自己的子树中，把不包括自己的下一层子树再拆分出来，以此类推，直到只剩下自己。
    - 以上图的 110 节点为例，从根节点开始，由于 110 节点在右子树，所以将左边的整颗子树拆分出来，即包含 000、001、010 这三个节点的这颗子树；接着，到第二层子树，将不包含 110 节点的左子树再拆分出来，即包含 100 和 101 这两个节点的子树；最后，再将 111 拆分出来。这样，就将 110 节点之外的整个二叉树拆分出了三颗子树。
    - 完成子树拆分后，只要知道每个子树里面的其中一个节点，就可以进行递归路由实现整颗二叉树所有节点的遍历。但在实际场景下，由于节点是动态变化的，所以一般不会只知道每个子树的一个节点，而是需要知道多个节点。每个 **K-桶(K-bucket)**会记录每颗子树里所知道的多个节点。其实，一个**K-桶**就是一张**路由表**，如果拆分出来有 m 颗子树，那对应节点就需要维护 m 个路由表。每个节点都会各自维护自己的 m 个 K-桶，每个 K-桶里记录的节点信息一般会包括 NodeID、IP、Endpoint、与 Target 节点（即维护该 K-桶的节点）的异或距离等信息。以太坊中，每个节点维护的 K-桶数量为 256 个，这 256 个 K-桶会根据与 Target 节点的异或距离进行排序，每个 K-桶保存的节点数量上限是 16。
    - 该网络最大可有2^3，即8个关键字和节点，目前共有7个节点加入，每个节点用一个小圈表示（在树的底部）。我们考虑那个用黑圈标注的节点6，它共有3个K桶，节点0，1和2（二进制表示为000，001和010）是第一个K桶的候选节点，节点3目前（二进制表示为011）还没有加入网络，节点4和节点5（二进制表示分别为100和101）是第二个K桶的候选节点，只有节点7（二进制表示为111）是第3个K桶的候选节点。图中，3个K桶都用灰色圈表示，假如K桶的大小（即K值）是2，那么第一个K桶只能包含3个节点中的2个。
    - 众所周知，那些长时间在线连接的节点未来长时间在线的可能性更大，基于这种静态统计分布的规律，**Kademlia**选择把那些长时间在线的节点存入K桶，这一方法增长了未来某一时刻有效节点的数量，同时也提供了更为稳定的网络。当某个K桶已满，而又发现了相应于该桶的新节点的时候，那么，就首先检查K桶中最早访问的节点，假如该节点仍然存活，那么新节点就被安排到一个附属列表中（作为一个替代缓存）。只有当K桶中的某个节点停止响应的时候，替代cache才被使用。换句话说，新发现的节点只有在老的节点消失后才被使用。
- 在以太坊的 Kad 网络中，节点之间的通信是基于 UDP 的，另外设置了 4 个主要的通信协议：
    1. PING消息—用来测试节点是否仍然在线。

2. STORE消息—在某个节点中存储一个键值对
3. FIND_NODE消息—消息请求的接收者将返回自己桶中离请求键值最近的K个节点。
4. FIND_VALUE消息，与FIND_NODE一样，不过当请求的接收者存有请求者所请求的键的时候，它将返回相应键的值。每一个RPC消息中都包含一个发起者加入的随机值，这一点确保响应消息在收到的时候能够与前面发送的请求消息匹配。

定位节点

节点查询可以异步进行，也可以同时进行，同时查询的数量由α表示，一般是3。在节点查询的时候，它先得到它K桶中离所查询的键值最近的K个节点，然后向这K个节点发起FIND_NODE消息请求，消息接收者收到这些请求消息后将在他们的K桶中进行查询，如果他们知道离被查键更近的节点，他们就返回这些节点（最多K个）。消息的请求者在收到响应后将使用它所收到的响应结果来更新它的结果列表，这个结果列表总是保持K个响应FIND_NODE消息请求的最优节点（即离被搜索键更近的K个节点）。然后消息发起者将向这K个最优节点发起查询，不断地迭代执行上述查询过程。因为每一个节点比其他节点对它周边的节点有更好的感知能力，因此响应结果将是一次一次离被搜索键值越来越近的某节点。如果本次响应结果中的节点没有比前次响应结果中的节点离被搜索键值更近了，这个查询迭代也就终止了。当这个迭代终止的时候，响应结果集中的K个最优节点就是整个网络中离被搜索键值最近的K个节点（从以上过程看，这显然是局部的，而非整个网络）。

节点信息中可以增加一个往返时间，或者叫做RTT的参数，这个参数可以被用来定义一个针对每个被查询节点的超时设置，即当向某个节点发起的查询超时的时候，另一个查询才会发起，当然，针对某个节点的查询在同一时刻从来不超过α个。

定位资源

通过把资源信息与键进行映射，资源即可进行定位，杂凑表是典型的用来映射的手段。由于以前的STORE消息，存储节点将会有对应STORE所存储的相关资源的信息。定位资源时，如果一个节点存有相应的资源的值的时候，它就返回该资源，搜索便结束了，除了该点以外，定位资源与定位离键最近的节点的过程相似。

考虑到节点未必都在线的情况，资源的值被存在多个节点上（节点中的K个），并且，为了提供冗余，还有可能在更多的节点上储存值。储存值的节点将定期搜索网络中与储存值所对应的键接近的K个节点并且把值复制到这些节点上，这些节点可作为那些下线的节点的补充。另外，对于那些普遍流行的内容，可能有更多的请求需求，通过让那些访问值的节点把值存储在附件的一些节点上（不在K个最近节点的范围之类）来减少存储值的那些节点的负载，这种新的存储技术就是缓存技术。通过这种技术，依赖于请求的数量，资源的值被存储在离键越来越远的那些节点上，这使得那些流行的搜索可以更快地找到资源的储存者。由于返回值的节点的NODE_ID远离值所对应的关键字，网络中的“热点”区域存在的可能性也降低了。依据与键的距离，缓存的那些节点在一段时间以后将会删除所存储的缓存值。分散式杂凑表的某些实现（如Kad）即不提供冗余（复制）节点也不提供缓存，这主要是为了能够快速减少系统中的陈旧信息。在这种网络中，提供文件的那些节点将会周期性地更新网络上的信息（通过FIND_NODE消息和STORE消息）。当存有某个文件的所有节点都下线了，关于该文件的相关的值（源和关键字）的更新也就停止了，该文件的相关信息也就从网络上完全消失了。

加入网络

想要加入网络的节点首先要经历一个引导过程。在引导过程中，节点需要知道其他已加入该网络的某个节点的IP地址和端口号（可从用户或者存储的列表中获得）。假如正在引导的那个节点还未加入网络，它会计算一个目前为止还未分配给其他节点的随机ID号，直到离开网络，该节点会一直使用该ID号。

正在加入**Kademlia**网络的节点在它的某个K桶中插入引导节点（负责加入节点的初始化工作），然后向它的唯一邻居（引导节点）发起FIND_NODE操作请求来定位自己，这种“自我定位”将使得**Kademlia**的其他节点（收到请求的节点）能够使用新加入节点的Node Id填充他们的K桶，同时也能够使用那些查询过程的中间节点（位于新加入节点和引导节点的查询路径上的其他节点）来填充新加入节点的K桶。这一自查询过程使得新加入节点自引导节点所在的那个K桶开始，由远及近，逐个得到刷新，这种刷新只需通过位于K桶范围内的一个随机键的定位便可达到。

最初的时候，节点仅有一个K桶（覆盖所有的ID范围），当有新节点需要插入该K桶时，如果K桶已满，K桶就开始分裂，（参见APeer-to-peer Information System 2.4）分裂发生在节点的K桶的覆盖范围（表现为二叉树某部分从左至右的所有值）包含了该节点本身的ID的时候。对于节点内距离节点最近的那个K桶，**Kademlia**可以放松限制（即可以到达K时不发生分裂），因为桶内的所有节点离该节点距离最近，这些节点个数很可能超过K个，而且节点希望知道所有的这些最近的节点。因此，在路由树中，该节点附近很可能出现高度不平衡的二叉子树。假如K是20，新加入网络的节点ID为“xxx000011001”，则前缀为“xxx0011……”的节点可能有21个，甚至更多，新的节点可能包含多个含有21个以上节点的K桶。（位于节点附近的k桶）。这点保证使得该节点能够感知网络中附近区域的所有节点。（参见A Peer-to-peer Information System 2.4）

查询加速

Kademlia使用异或来定义距离。两个节点ID的异或（或者节点ID和关键字的异或）的结果就是两者之间的距离。对于每一个二进制位来说，如果相同，异或返回0，否则，异或返回1。异或距离满足三角形不等式：任何一边的距离小于（或等于）其它两边距离之和。

异或距离使得Kademlia的路由表可以建在单个bit之上，即可使用位组（多个位联合）来构建路由表。位组可以用来表示相应的K桶，它有个专业术语叫做前缀，对一个m位的前缀来说，可对应2^{m-1}个K桶。（m位的前缀本来可以对应2^m个K桶）另外的那个K桶可以进一步扩展为包含该节点本身ID的路由树。一个b位的前缀可以把查询的最大次数从logn减少到logn/b.这只是查询次数的最大值，因为自己K桶可能比前缀有更多的位与目标键相同，（这会增加在自己K桶中找到节点的机会，假设前缀有m位，很可能查询一个节点就能匹配2m甚至更多的位组），所以其实平均的查询次数要少的多。（参考Improving Lookup Performance over a Widely-DeployedDHT第三部分）

节点可以在他们的路由表中使用混合前缀，就像eMule中的Kad网络。如果以增加查询的复杂性为代价，Kademlia网络在路由表的具体实现上甚至可以是有异构的。

比较

- 二者都具有区中心化的特征
- 二者应用的场景不同，Gossip 的收敛性保证最终所有节点都能收到消息，而 Kademlia 可以实现在分布式环境下快速而又准确地路由
- Kademlia 更加灵活，Kad 的“K-bucket”是可以根据使用场景来调整 K 值，而且对 K 值的调整完全不影响代码实现
- Kademlia 更加稳定且性能好，Kad 的路由算法天生就支持并发。由于公网上的线路具有很大的不确定性（极不稳定），哪怕是同样两个节点，之间的传输速率也可能时快时慢。由于 Kad 路由请求支持并发，发出请求的节点总是可以获得最快的那个 peer 的响应。而 Gossip 的转发是随机的，没有 Kademlia 稳定

4、简述PBFT共识的主要工作流程。

概述

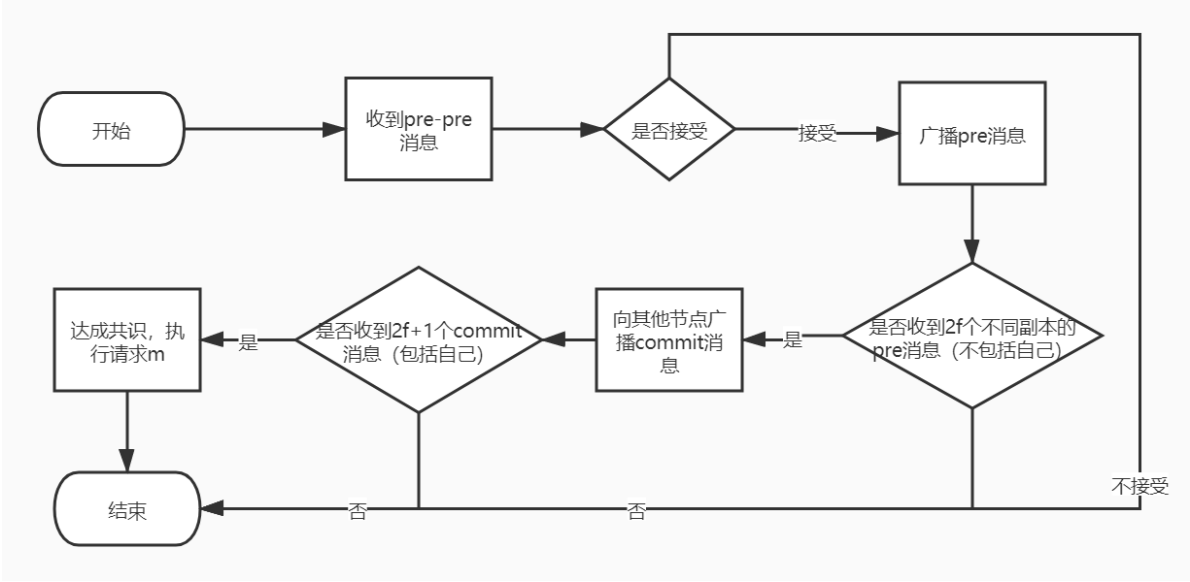
- PBFT 算法假设共识过程的运行环境是一个异步分布式网络（ 例如互联网环境），该网络中可能发生消息传输失败 、延迟、重复发送或乱序等， 并且可能存在蓄意发送错误消息的恶意节点。 PBFT 算法假定这些节点故障都是独立发生的， 并采用加密技术来防止欺骗和重播以及检测损坏的消息。消息包括公钥签名 、 消息认证编码以及由无碰撞散列函数产生的消息摘要等。
- PBFT 算法将消息表示为 m ,$< m > \sigma_i$ 表示由节点 i 签名的消息， D(m) 表示消息 m 的摘要。 按照惯例， PBFT 算法只对消息摘要而非完整消息进行签名， 并附在消息文本后。 所有节点都知道其他节点的公钥。

- PBFT 算法模型允许系统中存在一个强大的敌对节点， 它可以协调故障节点， 延迟通信或延迟正常节点，以造成最大的损害。 但是， 模型中假设该敌对节点不能无限期延迟正确的节点， 并且这个节点（以及它控制的错误节点） 在计算上受到限制， 因此（非常高概率） 它无法破坏上面提到的加密技术。 例如， 敌对节点不能产生正常节点的有效签名， 由摘要反向计算出相应的消息， 或找到两个具有相同摘要的消息等
- PBFT 也是一种基于状态机复制的实用共识算法： 服务被建模为状态机，在分布式系统中的不同节点上进行复制； 每个状态机副本（RepUca）都维护了服务状态， 并实施服务的相应操作。 以 R 表示状态机副本集合， 每个副本以一个{ 0， 1， … ， R - 1 }之间的整数作为编号。 假设 |R| =3f+1， 其中 f 为可能发生故障的副本的最大数量。 尽管 PBFT 系统可以容纳超过 3f+1 个副本，但这将降低系统整体性能而且没有提供更好的弹性（ 因为系统必须交换更多的消息） 。 换言之，PBFT 共识算法可在不高于约 33% 的拜占庭错误节点的系统中保证活性和安全性。
- 所有副本的状态变迁通过称为视图（ View） 的配置更换而进行。 在每个视图中， 只有一个副本为主节点（ Primary)， 其余副本作为备份（ Backups)。 主节点可以简单地由 p =v mode|R|产生， 其中 v为整数，是当前视图的编号。 若主节点失效， 则执行视图更换过程。 这一点与 VR 和 Paxos 共识算法相似， 区别在于后者不能容忍拜占庭故障。 副本状态包括服务状态 、 消息日志（记录副本收到的消息） 和当前视图编号
- 与其他状态机复制技术一样， PBFT 对每个副本节点提出了两个限定条件：
    - 所有节点必须是确定性的， 也就是说， 在给定状态和参数相同的情况下， 操作执行的结果必须相同；
    - 所有节点必须从相同的状态开始执行。 在这两个限定条件下， 即使失效的副本节点存在，PBFT 算法对所有非失效副本节点的请求执行总顺序达成一致， 从而保证安全性。

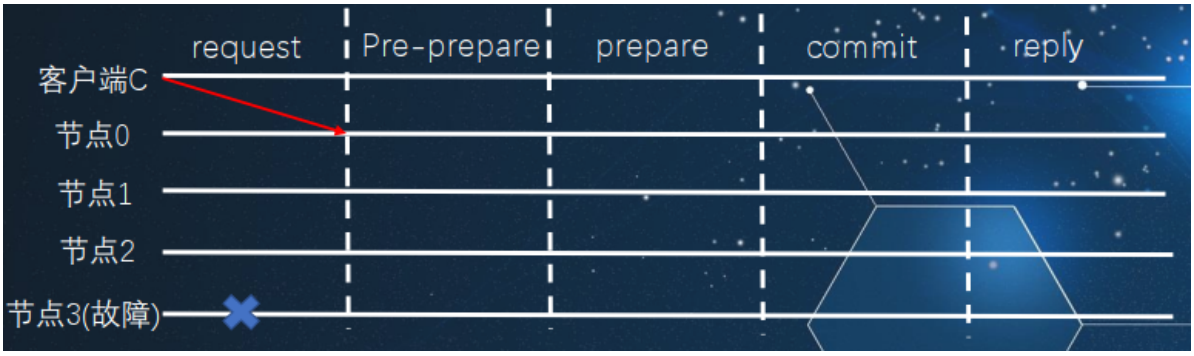
算法流程

下面描述在没有主节点故障的情况下，PBFT算法的正常运行流程。一次完整的共识需要经历请求 (Request) 、 预准备 （ Pre-prepare) 、 准备 （ Prepare) 、 确认 （ Commit) 和回复(Reply)五个阶段。

- 预准备和准备两个阶段用来确保同一个视图中请求发送的时序性(使对请求进行排序的主节点失效)
- 准备和确认两个阶段用来确保在不同的视图之间的确认请求是严格排序的

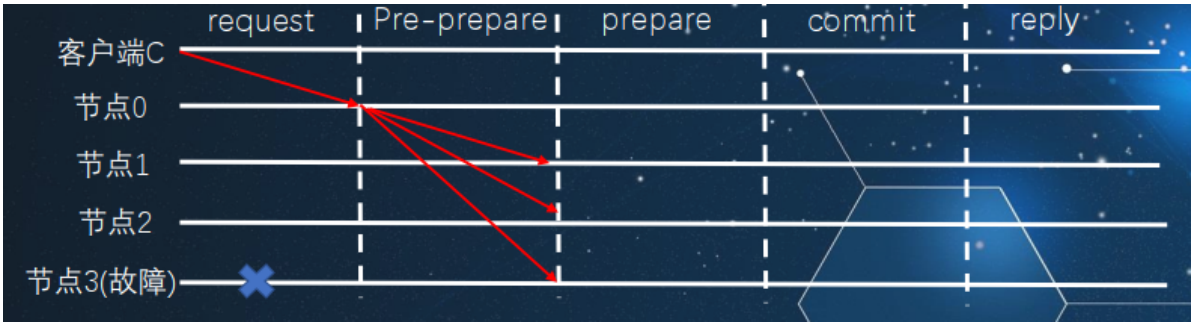


第一阶段：请求



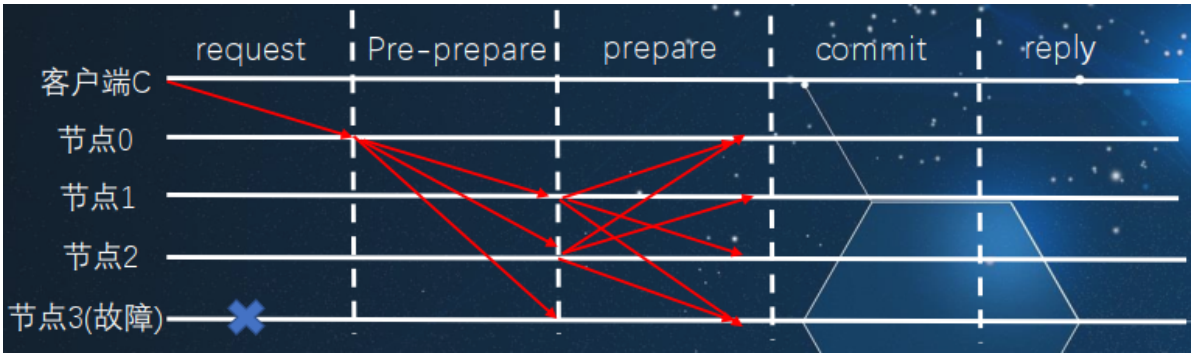
客户端签名发送消息$< REQUEST, o, t, c > \sigma_c$给主节点0， o为操作； t为时间戳，用于保证请求只被执行一次，也可以用于比较操作执行顺序，例如t可以设置为客户端发送请求时的本地时钟； c为客户端编号。

第二阶段：预准备



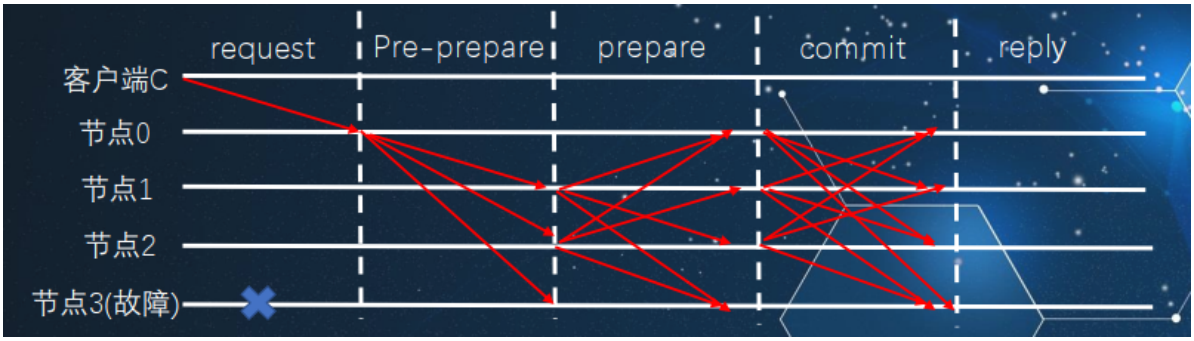
- 主节点构造消息$<< PRE-PREPARE, v, n, d > \sigma_0, m >$广播到集群中的其它节点，同时消息追加到消息日志中。
    - PRE-PREPARE标识当前消息所处的协议阶段。
    - v标识当前视图编号， n为主节点分配给所广播消息的一个唯一递增序号， m为客户端发来的消息， d为m的数字摘要

第三阶段：准备



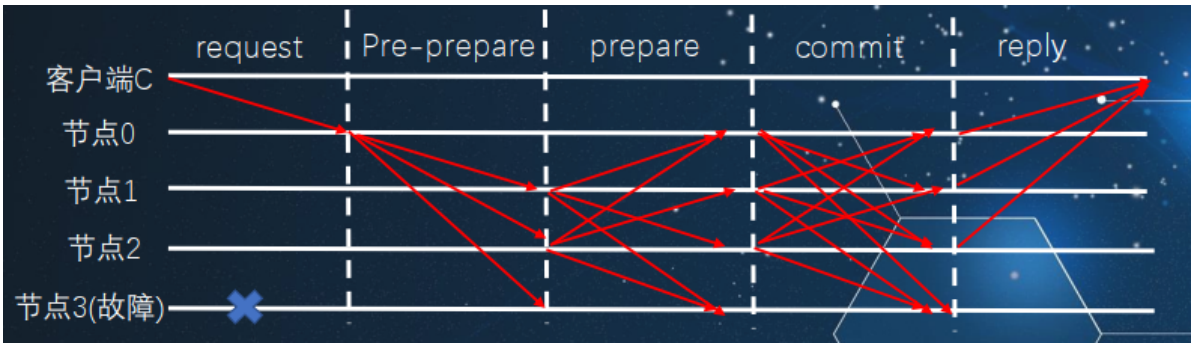
- 副本收到主节点请求后， 会对消息有效性进行检查， 检查通过会追加在消息日志中， 并广播消息$< PREPARE, v, n, d, i > \sigma_i$， 其中i是本节点的编号。
- 对消息的有效性有如下检查：
    - 检查收到的消息体中摘要d， 是否和自己对m生成的摘要一致， 确保消息的完整性。
    - 检查v是否和当前视图v一致。
    - 检查之前是否接收过相同序号n和v， 但是不同摘要d的消息。
    - 检查序号n是否在水线h和H之间， 避免快速消耗可用序号。 （防止作恶节点消耗序号空间）

第四阶段：确认



- 副本收到2f（不包括自己）个一致的PREPARE消息后，会进入COMMIT阶段，并且广播消息$<COMMIT, v, n, D(m), i>\sigma_i$给集群中的其它节点。
- 在收到PREPARE消息后，副本同样也会对消息进行有效性检查，包含上一阶段介绍的1、 2、 4三个检查步骤。

第五阶段：回复



- 副本收到2f+1（包括自己） 个一致的 COMMIT 消息后且已经没有序号小于n的请求， 则执行m中包含的操作（保证多个m按照序号n从小到大执行） ， 执行完毕后发送消息$<REPLY, v, t, c, i, r>\sigma_i$给客户端。 v为视图编号， t为时间戳， c为客户端编号, i为节点编号， r为操作结果。
- 客户端在收到回复后要进行签名验证、 时间戳比较和操作结果r比较。 当搜集到 f+1 个一致结果的回复之后才能确定执行结果。 如果在客户端给主节点发送请求之后的一定时间内没有响应， 客户端会广播请求， 所有副本会进行响应。 如果响应结果发现是主节点失效， 则会通过视图更换（ ViewChange ） 来切换主节点。

参考文献

https://github.com/liuchengxu/blockchain-tutorial/blob/master/content/part-5/address.md

https://xiaozhuanlan.com/topic/5017829364

https://zh.wikipedia.org/wiki/Kademlia

https://www.jianshu.com/p/54eab117e6ae

https://zhuanlan.zhihu.com/p/41228196

PPT 区块链共识层2-PBFT