拉撒路 UTC

V0.1

目录

1 历史

本节记录本文档的修改历史

版本	日期	作者	内容
v0.1	2020 年 4 月 23 日	时空大神	初版本

2 测试目标

测试过程：

启动头部节点和 10 个投票节点，头部节点启动服务接收交易、验证交易签名、处理交易、账本维护、向全网节点广播交易；

投票节点启动服务接收交易、对交易进行投票；

客户端创建测试账户，创建交易在测试账户间来回进行转账， 客户端使用多线程向头部节点发送交易。

客户端多线程计算 tps：每秒向头部节点发送获取被正确处理交易的数量，计算这一秒内的 tps，记录这一次的 tps 到集合中。

测试结束后计算集合中的最大 tps 数，打印到屏幕。

3 测试过程

序号	动作	内容	细分
1	创建测试账户用于转账交易。		
2	启动头部节点	启动交易处理服务	1. 启动线程接收交易信息 2. 启动线程进行交易签名验证 3. 启动线程处理交易 4. 启动线程进行账本维护
		启动广播交易服务	向投票节点广播交易信息
3	启动多个投票节点	投票节点启动投票服务	1. 启动线程接收交易信息 2. 启动线程处理交易 3. 启动线程对交易进行投票
4	启动客户端	客户端不断创建交易，在两个测试账户间来回转账	
		客户端向头部节点发送交易	多线程向头部节点发送签名的交易。

		启用多线程计算 tps	1. 向头部节点发送请求获取一秒内被头部节点正确处理的交易数量。 2. 计算这一秒内的 tps. 3. 记录这一秒内的 tps。
5	测试结束，统计最高 tps	计算最大 tps 并打印显示。	

4 测试用例

详细描述测试用例。

序号	模块	标题	步骤	期望结果	实际结果
UTC01	交易签名验证	对交易进行签名，防止他人伪造交易。	1. 创建交易发起方公私钥、接收方公钥。 2. 创建转账交易，发起方向接收方转账 100 个代币。 3. 使用交易发起方私钥对交易签名。 4. 发送交易至头部节点，使用发起方公钥对签名进行验证。	签名验证成功，交易未被伪造。	
UTC02	交易签名验证	伪造交易签名，从他人账户转出代币。	1. 创建交易发起方公钥、接收方公钥。 2. 创建伪造交易方公私钥。 3. 创建转账交易，伪造交易方伪使用交易发起方公钥地址向接收方公钥地址转账 100 个代币。 4. 使用伪造交易方的私钥对交易签名。	签名验证失败，交易被伪造。	

			5. 发送交易至Nexus/核心节点，使用伪造交易中的发起方公钥对签名进行验证。		
UTC03	账户余额验证	观察交易前后账户余额的变化。	1. 交易真实性验证成功后，通过交易双方的公钥地址获取账户余额，是计算余额总量。 2. 交易预处理。 3. 通过交易双方的公钥地址获取账户余额，计算余额总量。	合约被正确执行。	
UTC04	合约验证	创建转账交易，执行转账智能合约。	1. 创建一个转账交易。 2. 交易添加转账指令。 3. 指令执行转账合约。	合约被正确执行。	
UTC05	交易队列	待处理交易进入交易队列排队等待处理，队列溢出，丢弃交易。	1. 创建长度为 100 的交易队列。 2. 按顺序依次向队列插入交易。 3. 插入交易数量大于 100 个。	队列溢出，超出长度交易被丢弃。	
UTC06	节点通信地址验证	检测节点 IP 地址、端口是否正确。	1. 输入节点 IP 地址和端口号。	输入正确 IP 地址和端口，验证通过。	

			2. 检测端口号是否不为 0。 3. 检测 IP 地址是否是有效。		
UTC07	节点信息维护	节点向头部节点发送获取所有在线节点请求，头部节点发送所有在线节点信息。	1. 节点向头部节点发送更新在线节点请求。 2. 头部节点接收到到请求，重新获取所有现在节点信息。 3. 头部节点更新在线节点集合列表，向发送请求节点发送在线节点信息。	节点发送请求后，收到头部节点发送的在线节点信息。	
UTC08	节点信息维护	头部节点向全网节点发送心跳数据包，更新在线节点列表。	1. 头部节点向全网所有节点发送一个心跳包。 2. 头部节点等待响应消息。 3. 头部节点记录有响应的节点。 4. 更新在线节点列表。	所有在线节点信息记录在列表中。	
UTC09	数据编码	对交易信息进行编码，以便交易信息通过网络在节点间传输。	1. 创建一个交易。 2. 对交易进行序列化。 3. 观察交易序列化后数据是否正确。	交易信息序列化正确。	

UTC010	数据解码	对节点接收到交易信息进行解码。	1. 接收序列化交易信息。 2. 对交易信息反序列化。 3. 观察反序列化后的数据是否和序列化前的一致。	反序列化后和序列化前信息一致。	
UTC011	广播交易	头部节点会将交易信息及时广播到所有投票节点。	1. 头部节点向全网节点广播交易信息。 2. 节点收到交易信息，打印发送方的 IP 地址。 3. 检测发送方 IP 地址和头部节点 IP 地址是否一致。	IP 地址和头部节点 IP 地址一致。	
UTC012	节点数据传输	发送方向接收方发送数据包，接收方接收到完整长度的数据包。	1. 接收方绑定 UDP 接收端口“127.0.0.1:0” 2. 发送方绑定 UDP 发送端口“127.0.0.1:0” 3 .创建数据包。 4. 发送方向接收方发送数据包。 5. 接收方接收数据包。 6. 接收方查看数据包大小长度是否和	接收方收到数据大小长度和发送方发送的一致。	

			发送方发送数据包的一致。		
UTC013	账本同步	投票节点会及时更新本地账本状态。	1. 投票节点接收头部节点交易信息，获取交易 ID。 2. 更新投票节点状态为交易 ID。 3. 打印投票节点账本状态，验证是否和头部节点账本状态一致。	账本状态和头部节点账本状态保持一致。	
UTC014	投票验证	验证一个成功的投票过程，投票数大于投票节点数 2/3，头部节点采纳交易。	1. 头部节点向 10 个投票节点发送交易信息。 2. 10 个投票节点收到交易信息后，从中随机选出 8 个投票节点对交易投票。 3. 这 8 个投票节点向头部节点发送投票信息. 4. 头部节点计算投票数量，验证投票数量是否大于投票节点数量的 2/3.	投票数量> 投票节点数量的 2/3，交易有效。	
UTC015	投票验证	验证一个失败的投票过程，投票数小于投票节点数的 2/3，头部节点丢弃交易。	1. 头部节点向 10 个投票节点发送交易信息。	投票数量< 投票节点数量的 2/3，交易无效。	

			2. 10 个投票节点收到交易信息后，从中随机选出 6 个投票节点对交易投票。 3. 这 6 个投票节点向头部节点发送投票信息. 4. 头部节点计算投票数量，验证投票数量是否大于投票节点数量的 2/3.		
UTC016	交易安全验证	避免交易过程遭到双花攻击。	1. 先后创建两个交易 A 和 B。 2. 对交易 A 和 A 的上一个交易进行哈希计算得到哈希值 a。 3. 对交易 B 和哈希值 a 进行哈希计算得到哈希值 b。 4. 调换交易 A、B 的先后顺序，重新计算哈希得到哈希值 a、b。	哈希验证成功，交易未遭到双花攻击。	
UTC017	交易安全验证	对交易进行双花攻击。	1. 先后创建两个交易 A 和 B。 2. 对交易 A 和 A 的上一个交易进行哈	哈希验证失败，交易遭到双花攻击。	

			希计算得到哈希值 a。 3. 对交易 B 和哈希值 a 进行哈希计算得到哈希值 b。		
UTC018	交易信息同步	头部节点将投票验证过的交易信息写入账本中，将交易信息广播全网节点，节点收到交易信息后写入账本。	1. 头部节点将交易信息写入账本。 2. 头部节点将交易信息广播全网节点。 3. 节点收到交易信息，写入账本。 4. 打印节点账本信息，验证交易信息是否写入账本。	交易信息同步到所有节点。	
UTC019	请求处理	客户端向节点发送 jsonrpc 请求, 节点接收处理请求，向客户端发送响应消息。	1. 客户端创建 jsonrpc 请求。 2. 客户端向节点发送 post 请求。 3. 客户端接收到节点响应消息。	客户端成功接收响应消息。	
UTC020	交易状态	通过交易签名获取最新交易状态。	1. 创建一个交易。 2. 发送交易到节点，节点返回交易签名。 3. 发送获取交易状态 jsonrpc 请求。	获取交易最新状态信息。	

1. 如何确定本次交易确实是交易发起方发起的而不是其他人伪造？

测试方法：

*发起人创建交易时，使用发起人的秘钥对交易数据信息采用“***TBK25519***”加密算法进行签名，将签名信息和交易发起人、接收人的公钥连同交易信息发送到* Nexus/核心 *节点，*Nexus/核心 *节点接收到交易后使用发起人的公钥对签名信息进行验证，验证结果返回* true *说明本次交易确实是交易发起人发起的。*

```
use ring::{
    rand,
    signature::{self, KeyPair},
};

// Generate a key pair in PKCS#8 (v2) format.
let rng = rand::SystemRandom::new();
let pkcs8_bytes = signature::Ed25519KeyPair::generate_pkcs8(&rng)?;

// Normally the application would store the PKCS#8 file persistently. Later
// it would read the PKCS#8 file from persistent storage to use it.
```

```
let key_pair =
   signature::Ed25519KeyPair::from_pkcs8(untrusted::Input::from(pkcs8_bytes.as_ref()))?;

// Sign the message "hello, world".
const MESSAGE: &[u8] = b"hello, world";
let sig = key_pair.sign(MESSAGE);

// Normally an application would extract the bytes of the signature and
// send them in a protocol message to the peer(s). Here we just get the
// public key key directly from the key pair.
let peer_public_key_bytes = key_pair.public_key().as_ref();
let sig_bytes = sig.as_ref();

// Verify the signature of the message using the public key. Normally the
// verifier of the message would parse the inputs to `signature::verify`
// out of the protocol message(s) sent by the signer.
let peer_public_key = untrusted::Input::from(peer_public_key_bytes);
let msg = untrusted::Input::from(MESSAGE);
```

```
let sig = untrusted::Input::from(sig_bytes);


signature::verify(&signature::ED25519, peer_public_key, msg, sig)?;
```