# 实验 12 PBFT 共识算法简单实现

## 【实验介绍】

本次实验涉及到实用拜占庭容错算法（PBFT, Practical Byzantine Fault Tolerance）的基础知识及共识流程，并需要读者填充共识阶段中的代码以实现 PBFT 算法。通过此次实验，读者将对拜占庭容错下最经典的算法有较全面的认识和理解，这也为未来深入学习区块链内核打下了基础。为了后续实验的顺利进行，接下来将介绍 PBFT 的相关知识及共识流程。

（1）实用拜占庭容错算法（PBFT）

实用拜占庭容错算法于 1999 年被提出，它最杰出的贡献之一就是将拜占庭容错算法的时间复杂度降为了 $O(n^2)$，这使得其成为了第一个在实际系统中应用的 BFT 算法。PBFT 应用于一个互不信任的多方网络中，可容忍 1/3 的恶意结点。假设有 f 个恶意结点，则总结点数须大于等于 3f+1 个才可使系统正常工作。总体上讲，每个结点在投票阶段确定收到 2f + 1 个响应就可以进行下一阶段；客户端的请求发到 leader 结点，由其组织进 pbft 共识。下图为 PBFT 算法的时序图，其有三个阶段。其中 0 是 leader 结点，3 是恶意结点。

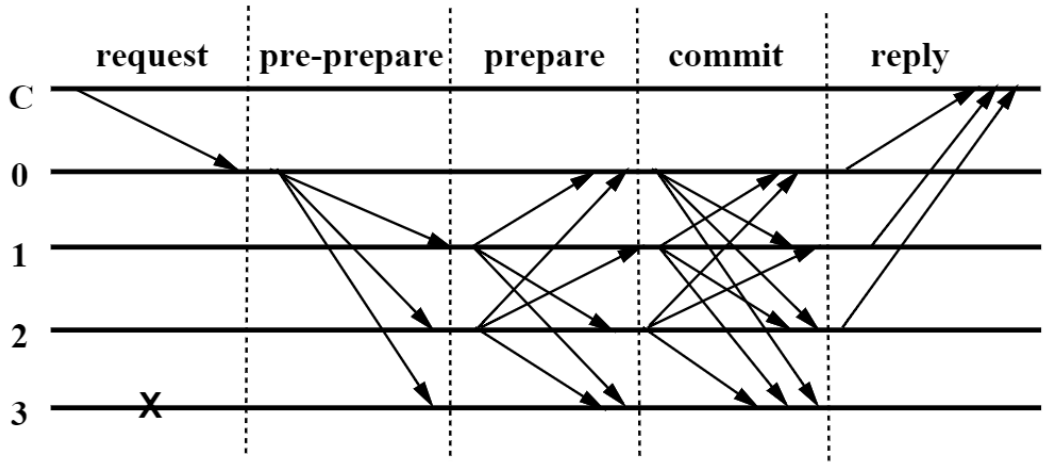



图 12-1 PBFT 算法的共识时序

PBFT 在某种程度上可以理解为是一种 SMR 算法。在该机制下，视图（view）是一个重要的概念。在一个 view 下，仅有一个主节点（leader），其余均为从节点。Leader 负责将客户端的请求发送给从节点。而在 BFT 下，leader 有可能是拜占庭的。这便要求从节点有义务检查 leader 的合法性，才出现异常时通过视图切换（view change）来确立新的 leader。Leader 的计算方法为：当前的视图号模节点的总数。接下来将介绍 PBFT 算法具体的共识流程：

1) REQUEST：

客户端向 leader 发送请求<request, o, t, c>。其中 request 包括请求的内容、内容的摘要及内容的签名，o 表示进行的操作，t 表示客户端赋予的时间戳，c 表示客户端的标识。

2) PRE-PREPARE：

leader 收到客户端的请求，根据签名即可判断请求是否合法。若合法则将其广播给所有从节点，广播的信息为<<pre-prepare, v, n, d>, m>。其中 v 是当前视图的编号，n 为请求的编号，d 是客户端请求内容的摘要，m 为具体内容。

1) PREPARE：

从节点收到 leader 的消息后，需要进行如下三项校验：leader 广播的消息的签名的正确性；从节点有没有收到一条在同一视图下且编号也是 n，但是签名却不同的 pre-prepare 信息；m 的摘要与 d 是否相同。

之后若请求合法，从节点则向其他节点发送<prepare, v, n, d, i>。其中 v、n、d 与先前表示的含义相同，而 i 是从节点的编号。

2) COMMIT：

leader 和从节点收到 prepare 消息后，须完成如下的检验：从节点 prepare 消息签名的正确性；此时从节点是否收到同一视图下的 n；已收到 pre-prepare 中的 d 和 prepare 消息中的 d 是否一致。

之后若从节点收到了 2f+1 个合法的 prepare 消息，则向所有节点发送形如<commit, v, n, d, i>的确认消息。其中 v、n、d、i 均与先前表示的含义相同。

3) REPLY：

当节点收到 commit 消息，须完成如下的检验：从节点 commit 消息签名的正确性；此时从节点是否收到同一视图下的 n；计算 d 的摘要与 m 的摘要是否相同。

之后若从节点收到了 2f+1 个合法的 commit 消息，表示现已完成确认，故向客户端发送形如<reply, v, t, c, i, r>的消息，其中 r 表示请求操作的返回结果。若客户端收到了 f+1 个一致的 reply 消息，表明此请求已完成了共识。

# 【实验要求】

了解 PBFT 算法的原理及共识流程，并能对给出的不完整代码进行补充。

# 【实验准备】

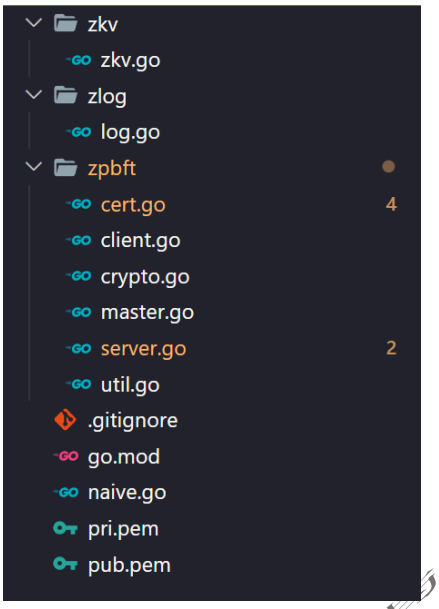
1. 熟悉 PBFT 算法的相关知识及基本流程。
2. 熟悉本次实验所给代码。

# 【实验过程】

### 1. 实验代码介绍

本实验是一个简单的分布式系统，节点有三种角色：master、server、client。master 负责登记 server 的注册信息（地址和公钥），并将所有 server 信息发给每个节点，然后 server 间互相建立 P2P 连接，等待 client 的请求到达。Client 也需要连接 master 获取 server 集群信息，当用户输入命令后，构建请求发给 server 中的 leader，由 leader 发起共识，在集群中达成一致后，执行该命令，这样所有的 server 将保持一致的副本状态（每个 server 拥有独自的 kv 数据库，但只要按同样顺序执行同样的命令序列，数据库的状态将保持一致）。

实验代码框架如下图所示，zkv 是简单的 kv 数据库，zlog 是分级别打印日志，zpbft 是共识协议实现，其中有 master、server、client 的实现代码框架。




主函数代码在 naive.go 中，其解析命令行参数（下图红框注释部分），通过指定不同的 role 参数选择不同身份运行。

```go
func main() {

    // 解析命令行参数
    var role, maddr, saddr, caddr, pri, pub string
    var f, logLevel int
    flag.StringVar(&role, "role", "", "role of process")
    flag.StringVar(&maddr, "maddr", "", "address of master")
    flag.StringVar(&saddr, "saddr", "", "address of server")
    flag.StringVar(&caddr, "caddr", "", "address of client")
    flag.StringVar(&pri, "pri", "", "private key file")
    flag.StringVar(&pub, "pub", "", "public key file")
    flag.IntVar(&f, "f", 1, "fault tolerance num")
    flag.IntVar(&logLevel, "log", 1, "log level(0:Debug,1:Info,2:Warn,3:Error)")

    flag.Parse()

    // 设置日志级别，默认为 Info(1)
    zlog.SetLevel(logLevel)

    // 根据不同身份启动节点
    switch role {
    case "master":
        // 执行命令: ./naive -role master -maddr localhost:8000 [-f 1]
        zpbft.RunMaster(maddr, f)
    case "server":
        // 执行命令: ./naive -role server -maddr localhost:8000 -pri ... -pub ... -saddr localhost:800x
        zpbft.RunServer(maddr, saddr, pri, pub)
    case "client":
        // 执行命令: ./naive -role client -maddr localhost:8000 -caddr localhost:9000
        zpbft.RunClient(maddr, caddr)
    default:
        flag.Usage()
    }
}
```

master.go 提供了 master 功能的全部实现，其具有两个 rpc，一个提供给 server 进行注册，一个提供给 client 获取 server 信息，所以 master 要先启动，这两个 rpc 会阻塞直至所有 server 都注册成功后再返回。master 需要 f 参数（即 pbft 中的容错数）来计算集群需要的 server 数（3f+1），本实验 f 默认为 1，需要启动四个 server 去注册，如果 f=2，则需要 7 个。

```
10  type Master struct {
11      peerNum int         // 节点数目 = 3f + 1
12      addrs   []string    // server注册地址表
13      pubkeys [][]byte    // server注册公钥表
14      mu      sync.Mutex // 互斥器
15      cond    sync.Cond  // 条件变量锁
16  }
17
18  func RunMaster(maddr string, f int) {
19      zlog.Info("Master, maddr:%s, f=%d, peerNum=%d", maddr, f, 3*f+1)
20      m := &Master{
21          peerNum: 3*f + 1,
22          addrs:   make([]string, 0),
23          pubkeys: make([][]byte, 0),
24          cond:    sync.Cond{L: &sync.Mutex{}},
25      }
26      // 开启 rpc server 监听
27      zlog.Info("waiting for node registration ...")
28      rpc.Register(m)
29      rpc.HandleHTTP()
30      err := http.ListenAndServe(maddr, nil) // 此处阻塞
31      if err != nil {
32          zlog.Error("http.ListenAndServe failed, %v", err)
33      }
34  }
```

client.go 提供了 client 功能的全部实现，在连接 master 获取 server 信息后，就可以接受用户的命令，构造请求发送给 leader，等 server 进行共识后，server 会发送 reply 给 client，client 确定接受到 f+1 个 reply 消息后就确定该命令执行成功，打印输出结果。下图为 client 接受命令构造请求通过 RPC 发送给 leader，其他逻辑见代码。

```go
func (c *Client) start() {

    time.Sleep(500 * time.Millisecond)
    zlog.Info("client start ...")
    for {
        // 构造请求，发送给 leader，通过共识获得结果
        fmt.Print(">>> ")
        // 读取一行输入
        fmt.Scan()
        cmd, err := bufio.NewReader(os.Stdin).ReadString('\n')
        if err != nil {
            zlog.Warn("read input error")
            continue
        }
        // 去掉最后面的换行符
        cmd = cmd[:len(cmd)-1]

        start := time.Now()
        // 构建请求参数
        args := &RequestArgs{
            Req: &RequestMsg{
                ClientAddr: c.addr,
                Timestamp:  time.Now().UnixNano(),
                Command:    cmd,
            },
        }
        reply := &RequestReply{}

        // rpc，失败则重试
        if err := c.peers[c.leader].rpcCli.Call("Server.RequestRpc", args, reply); err != nil {
            zlog.Warn("Call(\"Server.RequestRpc\", args, reply) failed, %v", err)
            continue
        }
        zlog.Info("L=%d seq= %d| 0:request => %d | cmd: [%s]", c.leader, reply.Seq, c.leader, cmd)
```

server.go 提供了 server 端的大部分代码，在向 master 注册信息返回并与其他节点建立连接后，leader 等待 client 请求到达，然后发起一轮 pbft 共识，经过三阶段的消息交互，执行该命令，改变本节点的 zkv 数据库状态，并将结果异步返回给 client。server 的具体逻辑见代码，下图是你需要实现的三个函数，注释给出了相关提示。

```
200 > func (s *Server) pbft() {···
206   }
207
208 > func (s *Server) prePrepare(seq int64) {···
235   }
236
237 > func (s *Server) PrePrepareRpc(args *PrePrepareArgs, reply *PrePrepareReply) error {···
270   }
271
272 > func (s *Server) prepare(seq int64) {···
293   }
294
295 > func (s *Server) PrepareRpc(args *PrepareArgs, reply *PrepareReply) error {···
308   }
309
310   func (s *Server) commit(seq int64) {
311       // TODO 请给出你的实现 (可参考prepare代码)
312   }
313
314   func (s *Server) CommitRpc(args *CommitArgs, reply *CommitReply) error {
315       // TODO 请给出你的实现 (可参考PrepareRpc代码)
316       return nil
317   }
318
319 > func (s *Server) verifyBallot(cert *LogCert) {···
331   }
332
333 > func (s *Server) verifyBallotPrepare(cert *LogCert) {···
372   }
373
374   func (s *Server) verifyBallotCommit(cert *LogCert) {
375       // TODO 请给出你的实现 (可参考verifyBallotPrepare代码)
376   }
377
378   // apply 日志，执行完后返回
379 > func (s *Server) apply(seq int64) {···
392   }
```

**2. 执行实验代码**

实现完上面提及的三个函数后，在项目目录下执行 go build naive.go 进行编译，生成可执行文件。(本实验请在 linux 环境下进行，其他系统因为换行符格式存在命令解析问题)

先启动 master，f 参数默认为 1 可不写。

./naive -role master -maddr :8000

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:43:08]
$ ./naive -role master -maddr :8000
master.go:19: INFO| Master, maddr::8000, f=1, peerNum=4
master.go:27: INFO| waiting for node registration ...
```

再启动四个 server，下图可以看到 id=0，addr=localhost:8001 的 server 成为 leader。(默认 id=0 为 leader，但 id 的分配与谁先注册到 master 有关)

./naive -role server -pri pri.pem -pub pub.pem -maddr :8000 -saddr :8001

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:44:18]
$ ./naive -role server -pri pri.pem -pub pub.pem -maddr :8000 -sadd
r :8001
server.go:85: INFO| connect master ...
server.go:94: INFO| register peer info ...
server.go:118: INFO| register success, id=0, leader=0
server.go:122: INFO| build connect with other peers ...
server.go:146: INFO| ==== connect all peers success ====
server.go:201: INFO| == start work loop ==
```

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:44:24]
$ ./naive -role server -pri pri.pem -pub pub.pem -maddr :8000 -saddr
:8002
server.go:85: INFO| connect master ...
server.go:94: INFO| register peer info ...
server.go:118: INFO| register success, id=1, leader=0
server.go:122: INFO| build connect with other peers ...
server.go:146: INFO| ==== connect all peers success ====
server.go:201: INFO| == start work loop ==
```

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:44:29]
$ ./naive -role server -pri pri.pem -pub pub.pem -maddr :8000 -sadd
r :8003
server.go:85: INFO| connect master ...
server.go:94: INFO| register peer info ...
server.go:118: INFO| register success, id=2, leader=0
server.go:122: INFO| build connect with other peers ...
server.go:146: INFO| ==== connect all peers success ====
server.go:201: INFO| == start work loop ==
```

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:44:34]
$ ./naive -role server -pri pri.pem -pub pub.pem -maddr :8000 -saddr
:8004
server.go:85: INFO| connect master ...
server.go:94: INFO| register peer info ...
server.go:118: INFO| register success, id=3, leader=0
server.go:122: INFO| build connect with other peers ...
server.go:146: INFO| ==== connect all peers success ====
server.go:201: INFO| == start work loop ==
```

最后启动 client。

./naive -role client -maddr :8000 -caddr :8005

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:46:03]
$ ./naive -role client -maddr :8000 -caddr :8005
client.go:69: INFO| connect master ...
client.go:96: INFO| build connect with other peers ...
client.go:119: INFO| ==== connect all peers success ====
client.go:125: INFO| client start ...
>>>
```

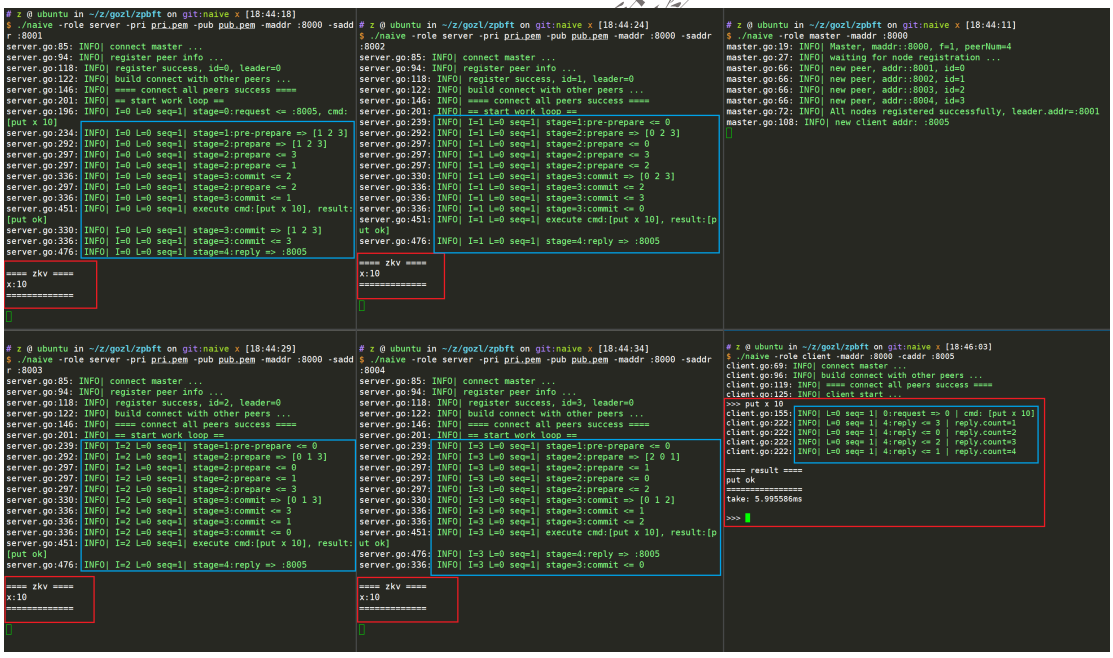
此时 master 端输出了相关注册信息：

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:44:11]
$ ./naive -role master -maddr :8000
master.go:19: INFO| Master, maddr::8000, f=1, peerNum=4
master.go:27: INFO| waiting for node registration ...
master.go:66: INFO| new peer, addr::8001, id=0
master.go:66: INFO| new peer, addr::8002, id=1
master.go:66: INFO| new peer, addr::8003, id=2
master.go:66: INFO| new peer, addr::8004, id=3
master.go:72: INFO| All nodes registered successfully, leader.addr=:8001
master.go:108: INFO| new client addr: :8005
```

本实验的 zkv 数据库支持四种操作。

```
usage: get <key>, put <key> <val>, del <key>, all
```

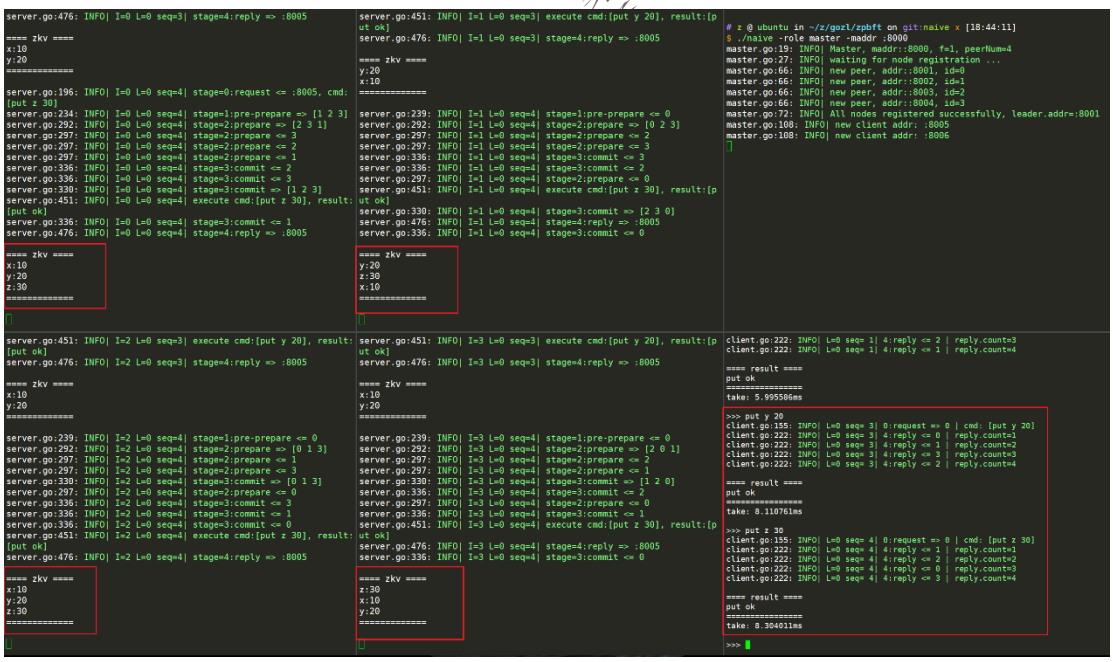
Client 输入命令 put x 10，如下图所示，右上角为 master，右下角为 client，左四个框为 server，蓝色框内容为 pbft 共识协议消息交互过程，=》和《=为发送方向，红色框为执行结果，四个 server 都执行了该命令，对本命令达成共识。



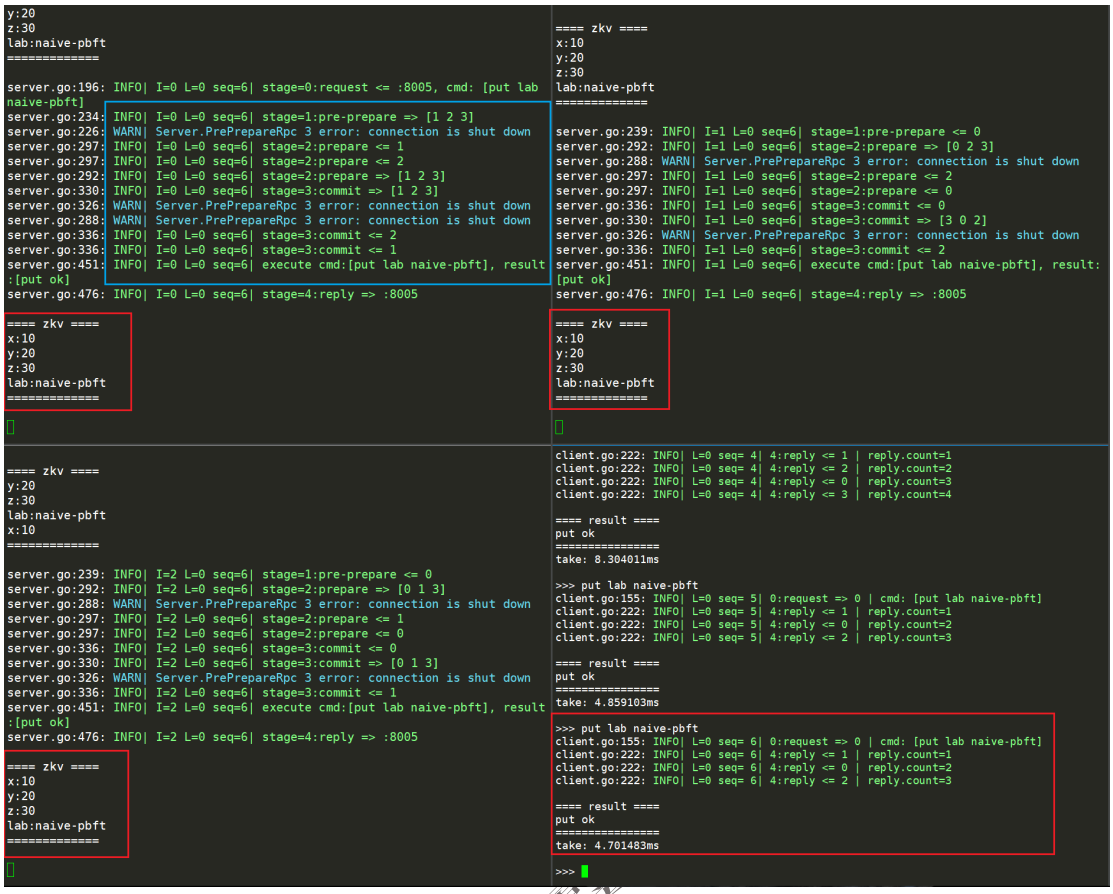
然后通过 get x 命令可以获取结果，这里我们启动另一个客户端来查看。

```
# z @ ubuntu in ~/z/gozl/zpbft on git:naive x [18:42:52] C:130
$ ./naive -role client -maddr :8000 -caddr :8006
client.go:69: INFO| connect master ...
client.go:96: INFO| build connect with other peers ...
client.go:119: INFO| ==== connect all peers success ====
client.go:125: INFO| client start ...
>>> get x
client.go:155: INFO| L=0 seq= 2| 0:request => 0 | cmd: [get x]
client.go:222: INFO| L=0 seq= 2| 4:reply <= 2 | reply.count=1
client.go:222: INFO| L=0 seq= 2| 4:reply <= 1 | reply.count=2
client.go:222: INFO| L=0 seq= 2| 4:reply <= 0 | reply.count=3
client.go:222: INFO| L=0 seq= 2| 4:reply <= 3 | reply.count=4

==== result ====
result: 10
================
take: 4.556776ms

>>>
```

此时回去查看 server，会发现 server 又根据这个客户端的命令达成共识。



此时关闭 id=3 的 server（右下角的 server），在故障节点为 1 的情况下仍然能达成共识，见下图，如果故障节点为 2，则无法达成共识。




其他的操作可见演示视频。

## 【实验小结】

本次实验介绍了实用拜占庭容错（PBFT）算法的背景知识、基本概念及共识流程，其中对共识流程的 5 个子阶段(REQUEST、PRE-PREPARE、PREPARE、COMMIT、REPLY)展开详细讲解，并描述了主从节点在不同子阶段的行为以及它们如何实现对客户端请求的成功共识。除此之外，实验实现了简单的 PBFT 算法并要求读者对代码中缺失的部分进行补全。这加深了读者对拜占庭环境下容错机制的理解，也为进一步学习区块链系统内核打下了坚实的基础。

## 【习题】

1. 请问在节点数量很大时 PBFT 算法的性能如何？

2. PBFT 算法解决了什么问题？请描述其五个阶段。

3.以太坊、Fabric 等区块链平台均开始使用 Golang，经过使用你觉得 Golang 的优势在哪里？

# 【参考文献】


[1] Castro M, Liskov B. Practical Byzantine fault tolerance and proactive recovery[J]. ACM Transactions on Computer Systems(TOCS), 2002, 20(4): 398-461.

[2] Lampotr L. The Part-Time Parliament[J]. ACM Transactions on Computer Systems, 1998, 16(2): 133-169.

[3] Lamport L, Shostak R, Pease M. The Byzantine Generals Problem[J]. ACM Transactions on Programming Languages and Systems, 1982, 4(3): 382-401.