**计算机类课程实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 院/系： | 区块链产业学院 | 课程名称： | 共识机制 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 班级 | 区块链工程214班 | 姓名 | 熊灵欣 | 学号 | 2021131126 | 实验室号 | 6610 |
| 日期 | 2024.6.18 | 组号 | 2 | 计算机号 |  |
| 实验名称 | PBFT共识算法的实践 | | | | | 成绩评定 |  |
| 所用软硬件 | 硬件：PC机，软件：VsCode | | | | |  |  |
| 1. 实验目的   理解PBFT算法的工作原理和流程，能够使用GO编程语言实现PBFT算法   1. 实验内容 2. 理解PBFT算法流程    1. Request 阶段：客户端向主节点发送请求消息    2. Pre-prepare 阶段：主节点接收到客户端请求后将请求数据里的主要信息提出，并向其余节点进行preprepare发送。    3. Prepare 阶段：从节点们接收到来自主节点的preprepare，首先利用主节点的公钥进行签名认证，其次将消息进行散列后，向其他节点广播prepare。    4. Commit 阶段：节点接收到2f个prepare信息(包含自己),并全部签名验证通过，则可以进行到commit步骤，向全网其他节点广播commit。    5. Reply 阶段：节点接收到2f+1个commit信息(包含自己)并全部签名验证通过，则可以把消息存入到本地，并向客户端返回reply消息。 3. 完善代码   // 序号累加  func (p \*pbft) sequenceIDAdd() {      p.lock.Lock()      p.sequenceID++      p.lock.Unlock()  }  // 向除自己外的其他节点进行广播  func (p \*pbft) broadcast(cmd command, content []byte) {      for i := range nodeTable {          if i == p.node.nodeID {              continue          }          message := jointMessage(cmd, content)          go tcpDial(message, nodeTable[i])      }  }  // 为多重映射开辟赋值  func (p \*pbft) setPrePareConfirmMap(val, val2 string, b bool) {      if \_, ok := p.prePareConfirmCount[val]; !ok {          p.prePareConfirmCount[val] = make(map[string]bool)      }      p.prePareConfirmCount[val][val2] = b  }  // 为多重映射开辟赋值  func (p \*pbft) setCommitConfirmMap(val, val2 string, b bool) {      if \_, ok := p.commitConfirmCount[val]; !ok {          p.commitConfirmCount[val] = make(map[string]bool)      }      p.commitConfirmCount[val][val2] = b  }  // 传入节点编号， 获取对应的公钥  func (p \*pbft) getPubKey(nodeID string) []byte {      key, err := ioutil.ReadFile("Keys/" + nodeID + "/" + nodeID + "\_RSA\_PUB")      if err != nil {          log.Panic(err)      }      return key  }  // 传入节点编号， 获取对应的私钥  func (p \*pbft) getPivKey(nodeID string) []byte {      key, err := ioutil.ReadFile("Keys/" + nodeID + "/" + nodeID + "\_RSA\_PIV")      if err != nil {          log.Panic(err)      }      return key  }  func pause() {      reader := bufio.NewReader(os.Stdin)      fmt.Print("Press enter to continue...")      \_, \_ = reader.ReadString('\n')  }  // 返回一个十位数的随机数，作为msgid  func getRandom() int {      x := big.NewInt(10000000000)      for {          result, err := rand.Int(rand.Reader, x)          if err != nil {              log.Panic(err)          }          if result.Int64() > 1000000000 {              return int(result.Int64())          }      }  }  // 客户端使用tcp 进行监听  func clientTcpListen() {      listen, err := net.Listen("tcp", clientAddr)      if err != nil {          log.Panic(err)      }      defer listen.Close()      for {          conn, err := listen.Accept()          if err != nil {              log.Panic(err)          }          b, err := ioutil.ReadAll(conn)          if err != nil {              log.Panic(err)          }          fmt.Println(string(b))      }  }  // 节点使用的tcp监听  func (p \*pbft) tcpListen() {      listen, err := net.Listen("tcp", p.node.addr)      if err != nil {          log.Panic(err)      }      fmt.Printf("节点开启监听，地址：%s\n", p.node.addr)      defer listen.Close()      for {          conn, err := listen.Accept()          if err != nil {              log.Panic(err)          }          b, err := ioutil.ReadAll(conn)          if err != nil {              log.Panic(err)          }          p.handleRequest(b)      }  }  // 使用tcp发送消息  func tcpDial(context []byte, addr string) {      conn, err := net.Dial("tcp", addr)      if err != nil {          log.Println("connect error", err)          return      }      \_, err = conn.Write(context)      if err != nil {          log.Fatal(err)      }      conn.Close()  }   1. 运行代码    1. 首先切换到项目根路径，分别运行下面两段代码进行初始化：       1. go mod init pbft       2. go build -o pbft.exe    2. 开启五个端口（一个客户端，四个节点）    3. 客户端执行 .\pbft.exe client    4. 其他四个节点依次执行 .\pbft.exe N0  .\pbft.exe N1  .\pbft.exe N2  .\pbft.exe N3 2. **实验代码**    1. Client.go   func clientSendMessageAndListen() {      // 开启客户端的本地监听（主要用来接收节点的reply信息）      go clientTcpListen()      fmt.Printf("客户端开启监听，地址：%s\n", clientAddr)      fmt.Println(" ---------------------------------------------------------------------------------")      fmt.Println("|  已进入PBFT测试Demo客户端，请启动全部节点后再发送消息！ :)  |")      fmt.Println(" ---------------------------------------------------------------------------------")      fmt.Println("请在下方输入要存入节点的信息：")      // 首先通过命令行获取用户输入      stdReader := bufio.NewReader(os.Stdin)      for {          data, err := stdReader.ReadString('\n')          if err != nil {              fmt.Println("Error reading from stdin")              panic(err)          }          r := new(Request)          r.Timestamp = time.Now().UnixNano()          r.ClientAddr = clientAddr          r.Message.ID = getRandom()          // 消息内容就是用户的输入          r.Message.Content = strings.TrimSpace(data)          br, err := json.Marshal(r)          if err != nil {              log.Panic(err)          }          fmt.Println(string(br))          content := jointMessage(cRequest, br)          // 默认N0为主节点，直接把请求信息发送至N0          tcpDial(content, nodeTable["N0"])      }  }  // 返回一个十位数的随机数，作为msgid  func getRandom() int {      x := big.NewInt(10000000000)      for {          result, err := rand.Int(rand.Reader, x)          if err != nil {              log.Panic(err)          }          if result.Int64() > 1000000000 {              return int(result.Int64())          }      }  }   * 1. Struct.go   // Request <REQUEST,o,t,c>  type Request struct {      Message      Timestamp int64      //相当于clientID      ClientAddr string  }  // PrePrepare <<PRE-PREPARE,v,n,d>,m>  type PrePrepare struct {      RequestMessage Request      Digest         string      SequenceID     int      Sign           []byte  }  // Prepare <PREPARE,v,n,d,i>  type Prepare struct {      Digest     string      SequenceID int      NodeID     string      Sign       []byte  }  // Commit <COMMIT,v,n,D(m),i>  type Commit struct {      Digest     string      SequenceID int      NodeID     string      Sign       []byte  }  // Reply <REPLY,v,t,c,i,r>  type Reply struct {      MessageID int      NodeID    string      Result    bool  }  type Message struct {      Content string      ID      int  }  const prefixCMDLength = 12  type command string  const (      cRequest    command = "request"      cPrePrepare command = "preprepare"      cPrepare    command = "prepare"      cCommit     command = "commit"  )  // 默认前十二位为命令名称  func jointMessage(cmd command, content []byte) []byte {      b := make([]byte, prefixCMDLength)      for i, v := range []byte(cmd) {          b[i] = v      }      joint := make([]byte, 0)      joint = append(b, content...)      return joint  }  // 默认前十二位为命令名称  func splitMessage(message []byte) (cmd string, content []byte) {      cmdBytes := message[:prefixCMDLength]      newCMDBytes := make([]byte, 0)      for \_, v := range cmdBytes {          if v != byte(0) {              newCMDBytes = append(newCMDBytes, v)          }      }      cmd = string(newCMDBytes)      content = message[prefixCMDLength:]      return  }  // 对消息详情进行摘要  func getDigest(request Request) string {      b, err := json.Marshal(request)      if err != nil {          log.Panic(err)      }      hash := sha256.Sum256(b)      // 进行十六进制字符串编码      return hex.EncodeToString(hash[:])  }   * 1. Tcp.go   // 客户端使用的tcp监听  func clientTcpListen() {      listen, err := net.Listen("tcp", clientAddr)      if err != nil {          log.Panic(err)      }      defer listen.Close()      for {          conn, err := listen.Accept()          if err != nil {              log.Panic(err)          }          b, err := ioutil.ReadAll(conn)          if err != nil {              log.Panic(err)          }          fmt.Println(string(b))      }  }  // 节点使用的tcp监听  func (p \*pbft) tcpListen() {      listen, err := net.Listen("tcp", p.node.addr)      if err != nil {          log.Panic(err)      }      fmt.Printf("节点开启监听，地址：%s\n", p.node.addr)      defer listen.Close()      for {          conn, err := listen.Accept()          if err != nil {              log.Panic(err)          }          b, err := ioutil.ReadAll(conn)          if err != nil {              log.Panic(err)          }          p.handleRequest(b)      }  }  // 使用tcp发送消息  func tcpDial(context []byte, addr string) {      conn, err := net.Dial("tcp", addr)      if err != nil {          log.Println("connect error", err)          return      }      \_, err = conn.Write(context)      if err != nil {          log.Fatal(err)      }      conn.Close()  }   * 1. Rsa.go   // 如果当前目录下不存在目录Keys，则创建目录，并为各个节点生成rsa公私钥  func genRsaKeys() {      if !isExist("./Keys") {          fmt.Println("检测到还未生成公私钥目录，正在生成公私钥 ...")          err := os.Mkdir("Keys", 0644)          if err != nil {              log.Panic()          }          for i := 0; i <= 4; i++ {              if !isExist("./Keys/N" + strconv.Itoa(i)) {                  err := os.Mkdir("./Keys/N"+strconv.Itoa(i), 0644)                  if err != nil {                      log.Panic()                  }              }              priv, pub := getKeyPair()              privFileName := "Keys/N" + strconv.Itoa(i) + "/N" + strconv.Itoa(i) + "\_RSA\_PIV"              file, err := os.OpenFile(privFileName, os.O\_RDWR|os.O\_CREATE, 0644)              if err != nil {                  log.Panic(err)              }              defer file.Close()              file.Write(priv)              pubFileName := "Keys/N" + strconv.Itoa(i) + "/N" + strconv.Itoa(i) + "\_RSA\_PUB"              file2, err := os.OpenFile(pubFileName, os.O\_RDWR|os.O\_CREATE, 0644)              if err != nil {                  log.Panic(err)              }              defer file2.Close()              file2.Write(pub)          }          fmt.Println("已为节点们生成RSA公私钥")      }  }  // 生成rsa公私钥  func getKeyPair() (prvkey, pubkey []byte) {      // 生成私钥文件      privateKey, err := rsa.GenerateKey(rand.Reader, 1024)      if err != nil {          panic(err)      }      derStream := x509.MarshalPKCS1PrivateKey(privateKey)      block := &pem.Block{          Type:  "RSA PRIVATE KEY",          Bytes: derStream,      }      prvkey = pem.EncodeToMemory(block)      publicKey := &privateKey.PublicKey      derPkix, err := x509.MarshalPKIXPublicKey(publicKey)      if err != nil {          panic(err)      }      block = &pem.Block{          Type:  "PUBLIC KEY",          Bytes: derPkix,      }      pubkey = pem.EncodeToMemory(block)      return  }  // 判断文件或文件夹是否存在  func isExist(path string) bool {      \_, err := os.Stat(path)      if err != nil {          if os.IsExist(err) {              return true          }          if os.IsNotExist(err) {              return false          }          fmt.Println(err)          return false      }      return true  }  // RsaSignWithSha256 数字签名  func (p \*pbft) RsaSignWithSha256(data []byte, keyBytes []byte) []byte {      h := sha256.New()      h.Write(data)      hashed := h.Sum(nil)      block, \_ := pem.Decode(keyBytes)      if block == nil {          panic(errors.New("private key error"))      }      privateKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)      if err != nil {          fmt.Println("ParsePKCS8PrivateKey err", err)          panic(err)      }      signature, err := rsa.SignPKCS1v15(rand.Reader, privateKey, crypto.SHA256, hashed)      if err != nil {          fmt.Printf("Error from signing: %s\n", err)          panic(err)      }      return signature  }  // RsaVerySignWithSha256 签名验证  func (p \*pbft) RsaVerySignWithSha256(data, signData, keyBytes []byte) bool {      block, \_ := pem.Decode(keyBytes)      if block == nil {          panic(errors.New("public key error"))      }      pubKey, err := x509.ParsePKIXPublicKey(block.Bytes)      if err != nil {          panic(err)      }      hashed := sha256.Sum256(data)      err = rsa.VerifyPKCS1v15(pubKey.(\*rsa.PublicKey), crypto.SHA256, hashed[:], signData)      if err != nil {          panic(err)      }      return true  }   * 1. Pbft.go   // 本地消息池（模拟持久化层），只有确认提交成功后才会存入此池  var localMessagePool = []Message{}  type node struct {      // 节点ID      nodeID string      // 节点监听地址      addr string      // RSA私钥      rsaPrivKey []byte      // RSA公钥      rsaPubKey []byte  }  type pbft struct {      // 节点信息      node node      // 每笔请求自增序号      sequenceID int      // 锁      lock sync.Mutex      // 临时消息池，消息摘要对应消息本体      messagePool map[string]Request      // 存放收到的prepare数量(至少需要收到并确认2f个)，根据摘要来对应      prePareConfirmCount map[string]map[string]bool      // 存放收到的commit数量(至少需要收到并确认2f+1个)，根据摘要来对应      commitConfirmCount map[string]map[string]bool      // 该笔消息是否已进行Commit广播      isCommitBordcast map[string]bool      // 该笔消息是否已对客户端进行Reply      isReply map[string]bool  }  func NewPBFT(nodeID, addr string) \*pbft {      p := new(pbft)      p.node.nodeID = nodeID      p.node.addr = addr      p.node.rsaPrivKey = p.getPivKey(nodeID) // 从生成的私钥文件处读取      p.node.rsaPubKey = p.getPubKey(nodeID)  // 从生成的私钥文件处读取      p.sequenceID = 0      p.messagePool = make(map[string]Request)      p.prePareConfirmCount = make(map[string]map[string]bool)      p.commitConfirmCount = make(map[string]map[string]bool)      p.isCommitBordcast = make(map[string]bool)      p.isReply = make(map[string]bool)      return p  }  func (p \*pbft) handleRequest(data []byte) {      // 切割消息，根据消息命令调用不同的功能      cmd, content := splitMessage(data)      switch command(cmd) {      case cRequest:          p.handleClientRequest(content)      case cPrePrepare:          p.handlePrePrepare(content)      case cPrepare:          p.handlePrepare(content)      case cCommit:          p.handleCommit(content)      }  }  // 处理客户端发来的请求  func (p \*pbft) handleClientRequest(content []byte) {      fmt.Println("主节点已接收到客户端发来的request ...")      // 使用json解析出Request结构体      r := new(Request)      err := json.Unmarshal(content, r)      if err != nil {          log.Panic(err)      }      // 添加信息序号      p.sequenceIDAdd()      // 获取消息摘要      digest := getDigest(\*r)      fmt.Println("收到的request消息为: ", r.Message)      fmt.Println("已将request存入临时消息池")      // 存入临时消息池      p.messagePool[digest] = \*r      // 主节点对消息摘要进行签名      digestByte, \_ := hex.DecodeString(digest)      signInfo := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)      // 拼接成PrePrepare，准备发往follower节点      pp := PrePrepare{\*r, digest, p.sequenceID, signInfo}      b, err := json.Marshal(pp)      if err != nil {          log.Panic(err)      }      pause()      fmt.Println("正在向其他节点进行进行PrePrepare广播 ...")      fmt.Println("PrePrepare消息内容为: ", pp)      // 进行PrePrepare广播      p.broadcast(cPrePrepare, b)      fmt.Println("PrePrepare广播完成")      pause()  }  // 处理预准备消息  func (p \*pbft) handlePrePrepare(content []byte) {      fmt.Println("本节点已接收到主节点发来的PrePrepare ...")      pause()      // 使用json解析出PrePrepare结构体      pp := new(PrePrepare)      err := json.Unmarshal(content, pp)      if err != nil {          log.Panic(err)      }      // 获取主节点的公钥，用于数字签名验证      primaryNodePubKey := p.getPubKey("N0")      digestByte, \_ := hex.DecodeString(pp.Digest)      if digest := getDigest(pp.RequestMessage); digest != pp.Digest {          fmt.Println("信息摘要对不上，拒绝进行prepare广播")      } else if p.sequenceID+1 != pp.SequenceID {          fmt.Println("消息序号对不上，拒绝进行prepare广播")      } else if !p.RsaVerySignWithSha256(digestByte, pp.Sign, primaryNodePubKey) {          fmt.Println("主节点签名验证失败！,拒绝进行prepare广播")      } else {          // 序号赋值          p.sequenceID = pp.SequenceID          // 将信息存入临时消息池          fmt.Println("已将消息存入临时节点池")          p.messagePool[pp.Digest] = pp.RequestMessage          // 节点使用私钥对其签名          sign := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)          // 拼接成Prepare          pre := Prepare{pp.Digest, pp.SequenceID, p.node.nodeID, sign}          bPre, err := json.Marshal(pre)          if err != nil {              log.Panic(err)          }          // 进行准备阶段的广播          fmt.Println("正在进行Prepare广播 ...")          fmt.Println("广播的Prepare消息内容为: ", pre)          p.broadcast(cPrepare, bPre)          fmt.Println("Prepare广播完成")          reader := bufio.NewReader(os.Stdin)          fmt.Print("Press enter to continue...")          \_, \_ = reader.ReadString('\n')      }  }  // 处理准备消息  func (p \*pbft) handlePrepare(content []byte) {      // 使用json解析出Prepare结构体      pre := new(Prepare)      err := json.Unmarshal(content, pre)      if err != nil {          log.Panic(err)      }      fmt.Printf("本节点已接收到%s节点发来的Prepare ... \n", pre.NodeID)      // 获取消息源节点的公钥，用于数字签名验证      MessageNodePubKey := p.getPubKey(pre.NodeID)      digestByte, \_ := hex.DecodeString(pre.Digest)      if \_, ok := p.messagePool[pre.Digest]; !ok {          fmt.Println("当前临时消息池无此摘要，拒绝执行commit广播")      } else if p.sequenceID != pre.SequenceID {          fmt.Println("消息序号对不上，拒绝执行commit广播")      } else if !p.RsaVerySignWithSha256(digestByte, pre.Sign, MessageNodePubKey) {          fmt.Println("节点签名验证失败！,拒绝执行commit广播")      } else {          p.setPrePareConfirmMap(pre.Digest, pre.NodeID, true)          count := 0          for range p.prePareConfirmCount[pre.Digest] {              count++          }          // 因为主节点不会发送Prepare，所以不包含自己          specifiedCount := 0          if p.node.nodeID == "N0" {              specifiedCount = nodeCount / 3 \* 2          } else {              specifiedCount = (nodeCount / 3 \* 2) - 1          }          // 如果节点至少收到了2f个prepare的消息（包括自己）,并且没有进行过commit广播，则进行commit广播          p.lock.Lock()          // 获取消息源节点的公钥，用于数字签名验证          if count >= specifiedCount && !p.isCommitBordcast[pre.Digest] {              pause()              fmt.Println("本节点已收到至少2f个节点(包括本地节点)发来的Prepare信息，内容为： ", pre)              // 节点使用私钥对其签名              sign := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)              c := Commit{pre.Digest, pre.SequenceID, p.node.nodeID, sign}              bc, err := json.Marshal(c)              if err != nil {                  log.Panic(err)              }              // 进行提交信息的广播              fmt.Println("正在进行commit广播 ...")              fmt.Println("广播的commit消息内容为: ", bc)              p.broadcast(cCommit, bc)              p.isCommitBordcast[pre.Digest] = true              fmt.Println("commit广播完成")          }          p.lock.Unlock()          pause()      }  }  // 处理提交确认消息  func (p \*pbft) handleCommit(content []byte) {      // 使用json解析出Commit结构体      c := new(Commit)      err := json.Unmarshal(content, c)      if err != nil {          log.Panic(err)      }      fmt.Printf("本节点已接收到%s节点发来的Commit ... \n", c.NodeID)      // 获取消息源节点的公钥，用于数字签名验证      MessageNodePubKey := p.getPubKey(c.NodeID)      digestByte, \_ := hex.DecodeString(c.Digest)      if \_, ok := p.prePareConfirmCount[c.Digest]; !ok {          fmt.Println("当前prepare池无此摘要，拒绝将信息持久化到本地消息池")      } else if p.sequenceID != c.SequenceID {          fmt.Println("消息序号对不上，拒绝将信息持久化到本地消息池")      } else if !p.RsaVerySignWithSha256(digestByte, c.Sign, MessageNodePubKey) {          fmt.Println("节点签名验证失败！,拒绝将信息持久化到本地消息池")      } else {          p.setCommitConfirmMap(c.Digest, c.NodeID, true)          count := 0          for range p.commitConfirmCount[c.Digest] {              count++          }          // 如果节点至少收到了2f+1个commit消息（包括自己）,并且节点没有回复过,并且已进行过commit广播，则提交信息至本地消息池，并reply成功标志至客户端！          p.lock.Lock()          if count >= nodeCount/3\*2 && !p.isReply[c.Digest] && p.isCommitBordcast[c.Digest] {              fmt.Println("本节点已收到至少2f + 1 个节点(包括本地节点)发来的Commit信息 ...")              // 将消息信息，提交到本地消息池中！              localMessagePool = append(localMessagePool, p.messagePool[c.Digest].Message)              info := ""              if p.node.nodeID != "N0" {                  info = p.node.nodeID + "节点已将msgid:" + strconv.Itoa(p.messagePool[c.Digest].ID) + "存入本地消息池中,消息内容为：" + p.messagePool[c.Digest].Content              } else {                  info = "主节点已将msgid:" + strconv.Itoa(p.messagePool[c.Digest].ID) + "存入本地消息池中,消息内容为：" + p.messagePool[c.Digest].Content              }              fmt.Println(info)              fmt.Println("正在reply客户端 ...")              tcpDial([]byte(info), p.messagePool[c.Digest].ClientAddr)              p.isReply[c.Digest] = true              fmt.Println("reply完毕")          }          p.lock.Unlock()      }  }  // 序号累加  func (p \*pbft) sequenceIDAdd() {      p.lock.Lock()      p.sequenceID++      p.lock.Unlock()  }  // 向除自己外的其他节点进行广播  func (p \*pbft) broadcast(cmd command, content []byte) {      for i := range nodeTable {          if i == p.node.nodeID {              continue          }          message := jointMessage(cmd, content)          go tcpDial(message, nodeTable[i])      }  }  // 为多重映射开辟赋值  func (p \*pbft) setPrePareConfirmMap(val, val2 string, b bool) {      if \_, ok := p.prePareConfirmCount[val]; !ok {          p.prePareConfirmCount[val] = make(map[string]bool)      }      p.prePareConfirmCount[val][val2] = b  }  // 为多重映射开辟赋值  func (p \*pbft) setCommitConfirmMap(val, val2 string, b bool) {      if \_, ok := p.commitConfirmCount[val]; !ok {          p.commitConfirmCount[val] = make(map[string]bool)      }      p.commitConfirmCount[val][val2] = b  }  // 传入节点编号， 获取对应的公钥  func (p \*pbft) getPubKey(nodeID string) []byte {      key, err := ioutil.ReadFile("Keys/" + nodeID + "/" + nodeID + "\_RSA\_PUB")      if err != nil {          log.Panic(err)      }      return key  }  // 传入节点编号， 获取对应的私钥  func (p \*pbft) getPivKey(nodeID string) []byte {      key, err := ioutil.ReadFile("Keys/" + nodeID + "/" + nodeID + "\_RSA\_PIV")      if err != nil {          log.Panic(err)      }      return key  }  func pause() {      reader := bufio.NewReader(os.Stdin)      fmt.Print("Press enter to continue...")      \_, \_ = reader.ReadString('\n')  }   * 1. Main.go   const nodeCount = 4  // 客户端的监听地址  var clientAddr = "127.0.0.1:8888"  // 节点池，主要用来存储监听地址  var nodeTable map[string]string  func main() {      // 为四个节点生成公私钥      genRsaKeys()      nodeTable = map[string]string{          "N0": "127.0.0.1:8000",          "N1": "127.0.0.1:8001",          "N2": "127.0.0.1:8002",          "N3": "127.0.0.1:8003",      }      if len(os.Args) != 2 {          log.Panic("输入的参数有误！")      }      nodeID := os.Args[1]      if nodeID == "client" {          clientSendMessageAndListen() // 启动客户端程序      } else if addr, ok := nodeTable[nodeID]; ok {          p := NewPBFT(nodeID, addr)          go p.tcpListen() // 启动节点      } else {          log.Fatal("无此节点编号！")      }      select {}  }   1. **实验结果**    1. 运行      * 1. 开启一个客户端      * 1. 四个节点（n0为主节点:8000、其余为从节点）      * 1. 测试      1. 系统无拜占庭节点，均正常运行         1. request阶段：      * + - 1. pre-prepare阶段：   主节点运行：     * + - 1. prepare阶段：   三个从节点运行      主节点运行     * + - 1. commit阶段：   主节点运行    三个从节点运行：     * + - 1. Reply阶段：   主节点运行    客户端收到消息：     * + 1. 系统存在一个拜占庭节点，其余两个从节点均正常运行        1. request阶段：      * + - 1. pre-prepare阶段：   主节点运行     * + - 1. prepare阶段：   两个从节点运行：     * + - 1. commit阶段：   主节点运行    两个从节点运行：     * + - 1. reply阶段：   主节点运行    两个从节点运行：    **可以看到，客户端依然会接收到reply，因为根据公式 n >= 3f+1 ，就算宕机一个节点，系统依然能顺利运行**   * + 1. 系统存在两个拜占庭节点，剩下一个从节点正常运行        1. request阶段：      * + - 1. Pre-prepare阶段   主节点运行：     * + - 1. Prepare阶段   一个从节点运行：    主节点运行     1. **实验总结**   在本次实验中，我们对PBFT共识算法进行了研究和实验。在VSCode使用Go语言实现Practical Byzantine Fault Tolerance（PBFT）协议的过程。通过五个终端模拟了一个客户端和四个节点的分布式系统，其中包括一个主节点和三个从节点，并分别模拟一个、两个恶意节点的情况下，PBFT协议的表现。实验内容包括理解PBFT算法的流程，包括Request、Pre-prepare、Prepare、Commit和Reply阶段。并实现了PBFT算法的各个阶段。实验代码包括序号累加、广播、多重映射赋值、获取节点公钥和私钥、随机数生成、客户端监听、节点监听、TCP发送消息等重要组成函数。实验运行分为初始化、启动五个端口（一个客户端和四个节点）、进行模拟实验几个步骤。  实验表现为：在没有恶意节点的情况下，系统能够高效地达成共识，并且所有节点都能够正确地复制状态；当存在一个恶意节点时，PBFT协议仍然能够正常工作，但系统的吞吐量略有下降；在两个恶意节点的情况下，系统返回错误，并拒绝执行。  通过实验，我们不仅对PBFT共识算法有了更深入的理解，同时也锻炼了我们的编程能力和系统分析能力。以下是实验的一些进一步分析和总结：  \*\*算法理解\*\*  PBFT算法作为一种经典的容错算法，其设计巧妙，通过多个阶段确保了即使在存在拜占庭节点的情况下，系统也能达成共识。实验过程中，我们深刻体会到每个阶段的重要性以及它们之间的逻辑关系。特别是在Prepare和Commit阶段，通过收集足够多的签名来验证消息的正确性，保证了算法的健壮性。  \*\*编程实践\*\*  在编程实现过程中，我们遇到了不少挑战。比如，如何高效地实现TCP通信、如何保证消息的可靠性和顺序性、如何管理节点的公钥和私钥等。通过不断地调试和优化，我们最终实现了PBFT算法的Go语言版本，并且能够在模拟环境中稳定运行。  \*\*性能分析\*\*  从实验结果来看，PBFT算法在拜占庭节点数量较少时表现良好，能够高效地达成共识。但是，随着拜占庭节点数量的增加，算法的性能会受到影响。这是因为PBFT算法需要收集足够多的签名来验证消息，而恶意节点可能会发送虚假消息或者拒绝签名，导致算法的执行效率降低。  \*\*改进方向\*\*  针对实验中发现的问题，我们可以考虑从以下几个方面进行改进：  1. \*\*优化网络通信\*\*：可以进一步优化TCP通信的实现，提高消息的传输速度和可靠性。  2. \*\*增加安全性措施\*\*：可以加入更多的安全性检查，比如对消息的完整性进行校验，防止恶意节点篡改消息。  3. \*\*性能调优\*\*：可以通过调整算法参数、优化数据结构等方式来提高算法的执行效率。  总之，通过这次实验，我们不仅加深了对PBFT共识算法的理解，还提高了我们的编程能力和系统分析能力。未来，我们可以继续深入研究分布式系统领域的其他算法和技术，为构建更加高效、安全的分布式系统打下坚实基础。 | | | | | | | |