Charles的技术博客

具有故障模拟功能的RPC实现分析

2016-09-07 | 分布式

引言

分布式系统学习中，需要理解如何对各种故障进行正确地处理，包括网络故障，机器故障等等。如何能方便的模拟故障，来验证自己的原型系统是否能够正确的应对这些故障就非常重要。本文将分析一个基于golang实现，具有故障模拟的RPC实现原理，源码来自MIT 6.824课程的labrpc。

RPC基本原理

在了解RPC实现之前，我们先来了解下RPC的基本原理。

在分布式系统中，RPC(remote procedure call)是当一个计算机程序触发在其他地址空间(通常是通过网络相连的其他计算机)执行一个计算过程，用户无需了解其实现细节，就像调用本机的函数一样，通常，一个RPC的调用类似如下

```
Client:
  z = fn(x, y)
Server:
  fn(x, y) {
    compute
    return z
  }
```

如上所示，Client执行函数 `fn(x,y)` ，通过RPC，最终在Server端执行此函数，并获取结果。如上所示的调用，其消息流如下

```
Client            Server
  request--->
     <---response
```

Client调用函数后，RPC库会给相应的Server发送请求，Server端执行完得到结果后，会通过RPC库发送响应给Client。

整个RPC的结构如下

```
client app         handlers
  stubs           dispatcher
 RPC lib           RPC lib
   net  ------------ net
```

Client调用远程过程的时候，会传入一些参数，Client stub(往往也在RPC库中实现)会对这些参数序列化，通过RPC库发送到网络；Server端网络收到包之后，RPC库处理后，通过dispatcher反序列化参数后，然后调用对应的handler。

根据以上结构，一次RPC中，有可能出现各种各样的故障，例如：

- 丢包
- 网络中断
- 机器故障
- 机器变慢

当发生故障时，对于Client来讲，一般会有如下反应：

- Client没有收到Server的响应
- Client不知道Server端是否收到了请求

RPC Client对故障的处理根据实现的不同而不同，一般有如下处理方式。

at least once

- RPC库等待库一段时间
- 如果没收到响应，重发请求
- 在收到请求前，会重试多次
- 如果最终还是无响应，则报错

at least once的处理方式实现比较简单，但是，对于不是幂等的操作会有问题。

相对而言，比at least once更好的处理方式为at most once。

at most once

- Server端检测重复请求，如果之前处理过，则返回之前处理的结果，否则，生成结果并返回

当采用at most once时，并且是把重复请求信息记录在内存中的，如果Server端宕机了，则记录的重复请求信息就不存在了，重启后，就无法正常工作了。

解决方案可能是把重复记录信息持久化，为了更安全，可能需要把这些信息复制到多个Server。

exactly once

- at most once + 具有多副本容灾，并且Client一直重试到成功

带有故障模拟的RPC实现分析

了解RPC基本原理后，我们来看一个基于golang channel的单机具有故障模拟的RPC库实现。

先通过一个简单的例子，了解此RPC的API使用方法

使用例子

```
type JunkArgs struct {
  X int
}
type JunkReply struct {
  X string
}

type JunkServer struct {
  mu   sync.Mutex
  log1 []string
  log2 []int
}

func (js *JunkServer) Handler1(args string, reply *int) {
  js.mu.Lock()
  defer js.mu.Unlock()
  js.log1 = append(js.log1, args)
  *reply, _ = strconv.Atoi(args)
```

```
19  }
20
21  func (js *JunkServer) Handler2(args int, reply *string) {
22    js.mu.Lock()
23    defer js.mu.Unlock()
24    js.log2 = append(js.log2, args)
25    *reply = "handler2-" + strconv.Itoa(args)
26  }
27
28  func (js *JunkServer) Handler3(args int, reply *int) {
29    js.mu.Lock()
30    defer js.mu.Unlock()
31    time.Sleep(20 * time.Second)
32    *reply = -args
33  }
34
35  // args is a pointer
36  func (js *JunkServer) Handler4(args *JunkArgs, reply *JunkReply) {
37    reply.X = "pointer"
38  }
39
40  func TestBasic(t *testing.T) {
41    runtime.GOMAXPROCS(4)
42
43    rn := MakeNetwork()
44
45    e := rn.MakeEnd("end1-99")
46
47    js := &JunkServer{}
48    svc := MakeService(js)
49
50    rs := MakeServer()
51    rs.AddService(svc)
52    rn.AddServer("server99", rs)
53
54    rn.Connect("end1-99", "server99")
55    rn.Enable("end1-99", true)
56
57    {
58      reply := ""
59      e.Call("JunkServer.Handler2", 111, &reply)
60      if reply != "handler2-111" {
61        t.Fatalf("wrong reply from Handler2")
62      }
63    }
64
65    {
66      reply := 0
67      e.Call("JunkServer.Handler1", "9099", &reply)
```

```
68      if reply != 9099 {
69        t.Fatalf("wrong reply from Handler1")
70      }
71    }
72  }
```

一次通用的RPC用到API如下：

- MakeNetwork，创建网络，其中网络里面有Client和Server组成
- MakeServer，创建Server，Server里面有不同的Service
- MakeService，创建Service，Service里面定义了不同的handle
- MakeEnd，创建Client
- Connect，Client调用Server前先要执行Connect
- Call，Client调用Server端的过程，通过参数执行要调用的handle，handle需要的参数

RPC的基本结构

从上面使用例子看出，整个RPC过程中包含以下基本结构：

- NetWork
- Server
- Client

Network

Network中包含一个或多个的Server和Client，其定义如下

```
1   type Network struct {
2     mu             sync.Mutex
3     reliable       bool
4     longDelays     bool                        // pause a long time on send o
5     longReordering bool                        // sometimes delay replies a l
6     ends           map[interface{}]*ClientEnd  // ends, by name
7     enabled        map[interface{}]bool        // by end name
8     servers        map[interface{}]*Server     // servers, by name
9     connections    map[interface{}]interface{} // endname -> servername
10    endCh          chan reqMsg
11  }
```

主要包括以下内容：

- servers：该网络中的所有Server
- ends：该网络中的所有Client
- connections：Client到Server的所有链接
- endCh：golang的channel，用来模拟传送数据的网络
- enabled：模拟Server是否宕机
- reliable：用来模拟网络是否可靠
- longDelays：用来模拟慢Server
- longReording：用来模拟网络的乱序

Server

Server中包含一系列的Service，其定义如下

```
type Server struct {
  mu       sync.Mutex
  services map[string]*Service
  count    int // incoming RPCs
}
```

- Service：Server所包含的Service
- count：达到Server的总的RPC总数

Client

Client为客户端，其定义如下

```
type ClientEnd struct {
  endname interface{} // this end-point's name
  ch      chan reqMsg // copy of Network.endCh
}
```

- endname：客户端名称
- reqMsg：发送消息的模拟网络，和Network的endCh是同一个channel

RPC实现分析

本部分主要分析API的实现，主要包括如下：

- MakeNetwork
- MakeEnd
- MakeServer
- AddServer
- MakeService
- AddService
- Connect
- Enable
- Call

MakeNetwork

其实现如下

```
1  func MakeNetwork() *Network {
2    rn := &Network{}
3    rn.reliable = true
4    rn.ends = map[interface{}]*ClientEnd{}
5    rn.enabled = map[interface{}]bool{}
6    rn.servers = map[interface{}]*Server{}
7    rn.connections = map[interface{}](interface{}){}
8    rn.endCh = make(chan reqMsg)
9
10   // single goroutine to handle all ClientEnd.Call()s
11   go func() {
12     for xreq := range rn.endCh {
13       go rn.ProcessReq(xreq)
14     }
15   }()
16
17   return rn
18 }
```

主要是初始化Network数据结构，然后，启动一个goroutine来处理Client的Call调用请求。

MakeEnd

创建Client，实现如下

```
1  func (rn *Network) MakeEnd(endname interface{}) *ClientEnd {
2    rn.mu.Lock()
3    defer rn.mu.Unlock()
4
5    if _, ok := rn.ends[endname]; ok {
6      log.Fatalf("MakeEnd: %v already exists\n", endname)
7    }
8
9    e := &ClientEnd{}
10   e.endname = endname
11   e.ch = rn.endCh
12   rn.ends[endname] = e
13   rn.enabled[endname] = false
14   rn.connections[endname] = nil
15
16   return e
17 }
```

主要是在Network结构中，添加客户端，并把其enabled和connections设置成空。

MakeServer

创建Server，其实现如下

```
1  func MakeServer() *Server {
2    rs := &Server{}
3    rs.services = map[string]*Service{}
4    return rs
5  }
```

初始化Server结构体的service为空的hashmap。

AddServer

往Network中添加Server，其实现如下

```
func (rn *Network) AddServer(servername interface{}, rs *Server) {
  rn.mu.Lock()
  defer rn.mu.Unlock()

  rn.servers[servername] = rs
}
```

在Network的servers中添加server。

MakeService

创建一个Service，其实现如下

```
func MakeService(rcvr interface{}) *Service {
  svc := &Service{}
  svc.typ = reflect.TypeOf(rcvr)
  svc.rcvr = reflect.ValueOf(rcvr)
  svc.name = reflect.Indirect(svc.rcvr).Type().Name()
  svc.methods = map[string]reflect.Method{}

  for m := 0; m < svc.typ.NumMethod(); m++ {
    method := svc.typ.Method(m)
    mtype := method.Type
    mname := method.Name

    //fmt.Printf("%v pp %v ni %v 1k %v 2k %v no %v\n",
    //  mname, method.PkgPath, mtype.NumIn(), mtype.In(1).Kind(), mtype.In(
    if method.PkgPath != "" || // capitalized?
      mtype.NumIn() != 3 ||
      //mtype.In(1).Kind() != reflect.Ptr ||
      mtype.In(2).Kind() != reflect.Ptr ||
      mtype.NumOut() != 0 {
      // the method is not suitable for a handler
      //fmt.Printf("bad method: %v\n", mname)
    } else {
      // the method looks like a handler
      svc.methods[mname] = method
    }
  }

  return svc
}
```

rcvr是一个golang的结构体，其上定义了一系列的方法，每个方法对应RPC的一个调用函数。整个处理方式流程如下：

- 创建Service结构体
- 通过golang的reflection方式，获取结构体的所有方法，通过reflect.TypeOf(rcvr).NumMethod()来获取
- 检测结构体中所有的method的参数是否符合RPC的标准。
- 把符合的方法添加到Service中，作为handle

Connect

Client连接Server,其实现如下

```
1  func (rn *Network) Connect(endname interface{}, servername interface{}) {
2    rn.mu.Lock()
3    defer rn.mu.Unlock()
4
5    rn.connections[endname] = servername
6  }
```

在Network结构体中的connections中设置endname的连接为servername。

Enable

设置此Client对应的Server是否宕机，其实现如下

```
1  func (rn *Network) Enable(endname interface{}, enabled bool) {
2    rn.mu.Lock()
3    defer rn.mu.Unlock()
4
5    rn.enabled[endname] = enabled
6  }
```

在Network结构体中的enables中设置endname的为enabled。

Call

Client调用RPC过程，其流程如下

```
1   qb := new(bytes.Buffer)
2   qe := gob.NewEncoder(qb)
3   qe.Encode(args)
4   req.args = qb.Bytes()
```

首先，把Client要发送的数据进行encode，即序列化

```
1   e.ch <- req
```

发送请求的数据到channel上，即模拟的网络上

```
1   rep := <-req.replyCh
```

Client等待Server端返回数据

在Client端发送数据后，Server端的处理流程如下

```
1   go func() {
2     for xreq := range rn.endCh {
3       go rn.ProcessReq(xreq)
4     }
5   }()
```

Server端检测到endCh中有数据，然后调用ProcessReq处理请求。

```
1   if enabled && servername != nil && server != nil {
2     if reliable == false {
3       // short delay
4       ms := (rand.Int() % 27)
5       time.Sleep(time.Duration(ms) * time.Millisecond)
```

```
    }

    if reliable == false && (rand.Int()%1000) < 100 {
      // drop the request, return as if timeout
      req.replyCh <- replyMsg{false, nil}
      return
    }
```

如果要模拟网络不是可靠的请求下，会按照如下流程处理

- 随机等待一小段时间
- 等待完后，以一定地概率不处理结果，直接返回Client失败

接着需要把请求分发到相应的handle处理

```
ech := make(chan replyMsg)
go func() {
  r := server.dispatch(req)
  ech <- r
}()
```

具体地dispatch实现如下

```
func (rs *Server) dispatch(req reqMsg) replyMsg {
  rs.mu.Lock()

  rs.count += 1

  // split Raft.AppendEntries into service and method
  dot := strings.LastIndex(req.svcMeth, ".")
  serviceName := req.svcMeth[:dot]
  methodName := req.svcMeth[dot+1:]

  service, ok := rs.services[serviceName]

  rs.mu.Unlock()

  if ok {
    return service.dispatch(methodName, req)
  } else {
    choices := []string{}
    for k, _ := range rs.services {
```

```
20          choices = append(choices, k)
21        }
22        log.Fatalf("labrpc.Server.dispatch(): unknown service %v in %v.%v; expe
23          serviceName, serviceName, methodName, choices)
24        return replyMsg{false, nil}
25      }
26    }
```

通过调用的结构体和函数名，定位到需要具体处理的函数，调用它，流程如下

```
1   // decode the argument.
2   ab := bytes.NewBuffer(req.args)
3   ad := gob.NewDecoder(ab)
4   ad.Decode(args.Interface())
5
6   // allocate space for the reply.
7   replyType := method.Type.In(2)
8   replyType = replyType.Elem()
9   replyv := reflect.New(replyType)
10
11  // call the method.
12  function := method.Func
13  function.Call([]reflect.Value{svc.rcvr, args.Elem(), replyv})
14
15  // encode the reply.
16  rb := new(bytes.Buffer)
17  re := gob.NewEncoder(rb)
18  re.EncodeValue(replyv)
```

首先，对RPC请求进行反序列化，调用对应的函数处理，最后把生成的结果进行序列化。

```
1     serverDead = rn.IsServerDead(req.endname, servername, server)
2
3     if replyOK == false || serverDead == true {
4       // server was killed while we were waiting; return error.
5       req.replyCh <- replyMsg{false, nil}
6     } else if reliable == false && (rand.Int()%1000) < 100 {
7       // drop the reply, return as if timeout
8       req.replyCh <- replyMsg{false, nil}
9     } else if longreordering == true && rand.Intn(900) < 600 {
10      // delay the response for a while
11      ms := 200 + rand.Intn(1+rand.Intn(2000))
```

```
      time.Sleep(time.Duration(ms) * time.Millisecond)
      req.replyCh <- reply
    } else {
      req.replyCh <- reply
    }
  } else {
    // simulate no reply and eventual timeout.
    ms := 0
    if rn.longDelays {
      // let Raft tests check that leader doesn't send
      // RPCs synchronously.
      ms = (rand.Int() % 7000)
    } else {
      // many kv tests require the client to try each
      // server in fairly rapid succession.
      ms = (rand.Int() % 100)
    }
    time.Sleep(time.Duration(ms) * time.Millisecond)
    req.replyCh <- replyMsg{false, nil}
  }
```

根据一系列的配置，决定是否返回结果以及何时返回结果，用来模拟故障情况

- 如果enabled为false，则模拟Server挂掉的情况，则直接返回失败。
- 如果reliable为false，则模拟网络不可靠情况，有概率返回失败
- 如果longreording为true，则以一定概率等待一定时间返回结果，以模拟网络包乱序地情况
- 如果是longDelays为true，则会等待一段事件再返回结果，模拟高时延的情况

最后，Client收到数据后，会按照以下流程处理

```
  if rep.ok {
    rb := bytes.NewBuffer(rep.reply)
    rd := gob.NewDecoder(rb)
    if err := rd.Decode(reply); err != nil {
      log.Fatalf("ClientEnd.Call(): decode reply: %v\n", err)
    }
    return true
  } else {
    return false
  }
```

即反序列化Server端的响应，最终返回结果给应用端。

PS:

本博客更新会在第一时间推送到微信公众号，欢迎大家关注。

参考文献

- MIT 6.824 labrpc
- remote procedure call
- labrpc code

#golang #RPC

golang reflection学习笔记 gfs原理

1条评论

Hello_Code

Mock RPC 来写单测应该不错

2016年9月13日 回复 顶 转发