1. 基础数据类型：

|  |  |  |
| --- | --- | --- |
| 名称 | 类型 | 描述 |
| peerid | STRING | 标识一个PEER区别于另一个PEER的字符串 |
| EP（endpoint） | STRING | 按照“4@IP@port@protocol”格式组织的表示一个PEER上可连接地址的字符串;  Protocol = ‘u’表示UDP  Protocol = ‘t’表示TCP |
| eplist | ARRAY[EP] | 一个peer可能有多个可连接的EP，比如：内网地址和公网地址 |
| PEER | {  id: peerid,  eplist: eplist  } | 一个PEER的准确描述信息 |

1. BDT

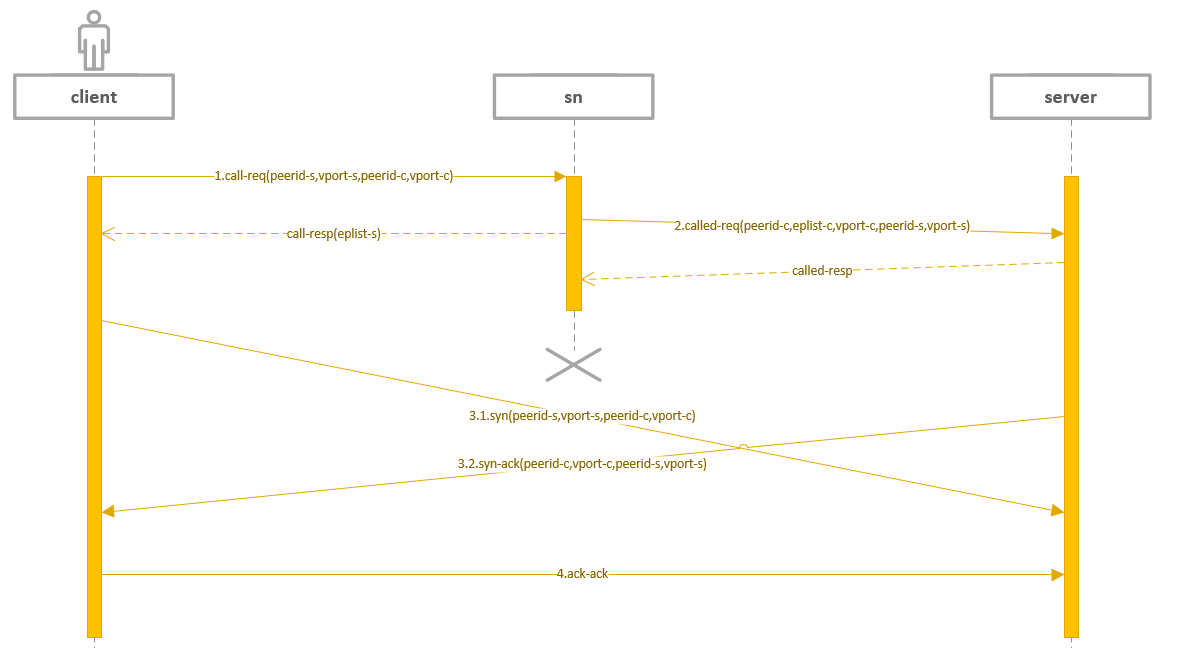
BDT负责在两个PEER之间建立连接，建立连接时客户端需要知道服务端的peerid，BDT模块负责将peerid转换为可用于网络通信的地址（即eplist）,保证双方通信数据的顺序性和完整性，并维护连接的状态。

1. 寻址和握手

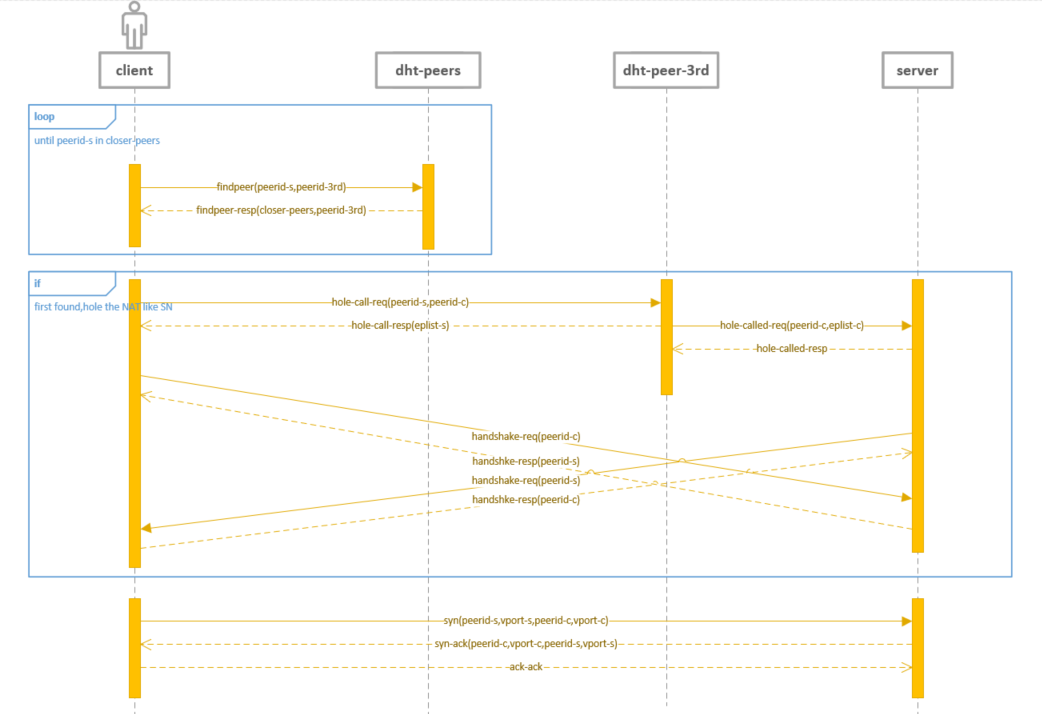
BDT在连接对方PEER时，要先通过peerid查到对方PEER的eplist，然后对eplist中所有的地址发起握手，第一个有响应的地址即是对方PEER对本地PEER可达的地址，当然本地PEER可能连接不到对方PEER，这时候需要通过双方都可达的第三方主机（一般是专门提供穿透功能的服务器，即后面提到的SN）通知对方主动向本地连接（即是所谓的穿透）。

寻址有SN和DHT两种途径，下面分别介绍详细过程：

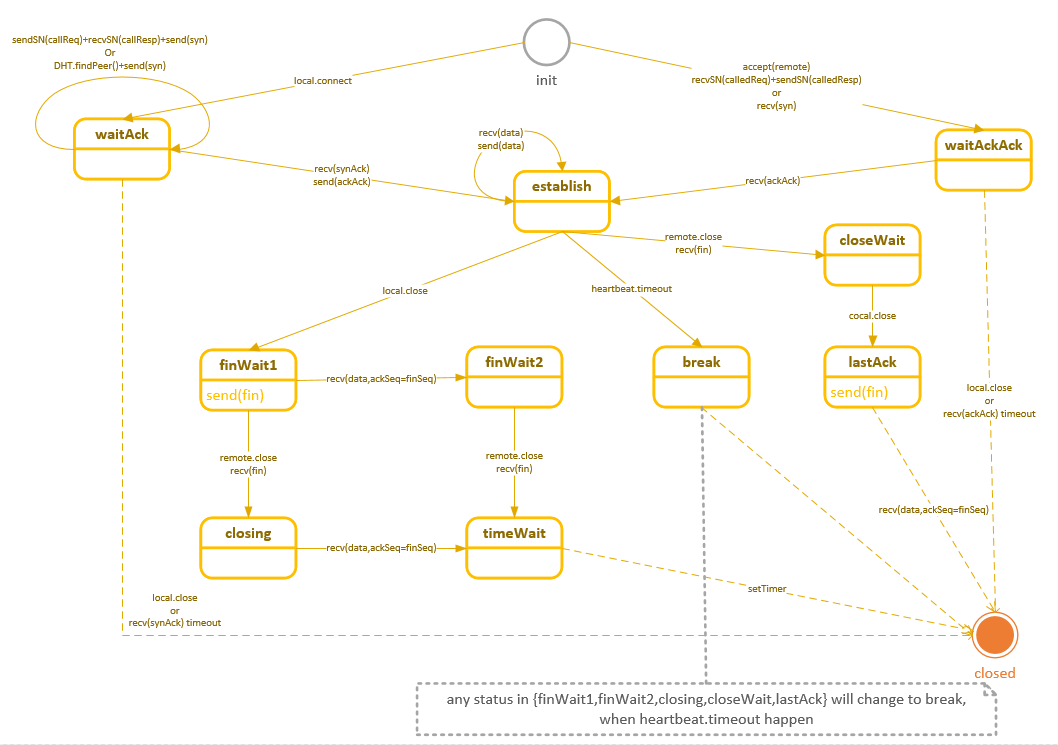
A.通过SN



B.通过DHT，穿透和握手逻辑大体和SN的差不多，区别在于通过DHT网络获取到server地址时一定是通过DHT网络中的一个第三方节点，而这个节点能够同时触达client和server，所以这个节点能够作为实现穿透的服务器；也就是说通过DHT建立连接时固定的穿透服务器不是必须的，DHT网络中的任意节点都可能提供穿透服务。



1. 连接状态机



1. 协议格式

BDT协议主要包括三个部分：协议头，控制命令body，负载数据data。协议头是个固定格式的二进制缓冲区；body是个用“msgpack-lite”编码的object；data是用户要传输的二进制缓冲区。

协议头格式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 描述 | 默认值 |
| magic | uint16 | 校验协议类型的常量 | 0x8083  常量 |
| version | uint16 | 协议版本号，协议升级的时候要修改它 | 0x0102 常量 |
| cmdType | uint16 | 命令类型 |  |
| totalLength | uint16 | body.length + data.length + headerLength |  |
| headerLength | BYTE | 用于协议版本兼容 | 35 |
| flags | uint16 |  | 协议类型相关 |
| bodyLength | uint16 | body经“msgpack-lite”编码后的字节数 |  |
| source\_vport | uint16 | 主动连接方的vport，类似IP协议的port |  |
| dest\_vport | uint16 | 连接目标方的vport，类似IP协议的port |  |
| src\_peerid\_hash | uint16 | 发包方peerid hash  1.hash算法：peerid所有字节累加和\*33，取低16位  2.和SN通信时，因为SN的peerid不是必须的，其hash值可以是0  3.如果body中有相应peerid，hash应该和它相匹配 | 0 |
| dest\_peerid\_hash | uint16 | 收包方peerid hash  用法同src\_peerid\_hash | 0 |
| seq | uint32 | 命令包唯一序号 | 0 |
| ackSeq | uint32 | 被确认包的seq |  |
| window\_size | uint16 | 本端接收窗口大小 |  |
| sessionid | Uint32 | 连接建立前（syn/callReq/ calledResp/synAck）用于通知对端自己的sessionid；  连接建立后（synAckAck/heartbeat/ heartbeatResp/data/fin）填充对端的sessionid，对端根据sessionid确定具体的连接；  SN包：  callResp/calledReq都填连接发起方的sessionid  pingReq/pingResp：由客户端决定，当客户端和多个SN建立ping时，客户端通过sessionid决定SN回包的分发 |  |

各命令详细格式：

1. pingReq = 0x10

方向：PEER -> SN

功能：在SN上登记，便于其他peer能从这个SN上找到

body = {

peerid, // string 自己的peerid

eplist, // [EP] 监听的地址信息列表，

}

1. pingResp = 0x11

方向：SN -> PEER

功能：响应pingReq

body = {

result; // int, 0-ok,1-peerid和hash不匹配

peerid; // string, 客户端的peerid，就是REQ.body.peerid

eplist; // [EP], merge(客户端的eplist, 客户端公网ep)

forward; // EP, 重定向到新的SN上线

offline; // boolean，SN准备下线，客户端需要重新找一个SN

nearSN; // string, 比当前SN距离客户端更近的SN-peerid，客户端需要考虑在这个SN上上线

}

1. callReq = 0x12

方向：PEER -> SN

功能：通过SN向目标peer发起连接

body = {

src; // string 发起call的peerid

dest; // string 被call的peerid

eplist; // [EP] 客户端本地发现的所有地址，客户端可能通过其他途径了解到自己有多个地址（局域网地址，和其他节点连接的地址），一个节点跟不同节点通信时可能采用不同的地址，不光和目标SN通信的公网地址

}

1. callResp = 0x13

方向：SN -> PEER

功能：响应callReq

body = {

src; // string 被call的peerid，req.body.dest

dest; // string 发起call的peerid，req.body.src

result; // int 查询peer信息的结果，0是成功，其他失败

eplist; // [EP] 被call peer的eplist，接收到对端eplist，定时向对方发起握手(CMD\_SYN)，直到成功/超时

nearSN; // PEER SN发现有距离目标更近的SN，连接发起方可以从这个SN上尝试搜索目标节点

}

1. calledReq = 0x14

方向：SN -> PEER

功能：通知对方有peer请求连接，让对方同时发起反向握手

body = {

src; // string 发起call的peerid，call\_req.body.src

dest; // string 被call的peerid，call\_req.body.dest

eplist; // [EP] 发起call的eplist，目标peer接收到对端eplist，定时向对方发起握手(CMD\_SYNACK)，直到成功/超时

}

1. calledResp = 0x15

方向：PEER -> SN

功能：响应calledReq

body = {

src; // string 被call的peerid，call\_req.body.dest

dest; // string 发起call的peerid，call\_req.body.src

sessionid; // number 发起call一端的sessionid

}

1. syn = 0x20

方向：client->server

功能：client向server端发起的握手包

body = {

src; // string 客户端peerid

dest; // string server端peerid

}

1. synAck = 0x21

方向：server->client

功能： server端收到client端发来的CMD\_SYN或者SN转发来的calledReq包后向client发出的握手确认包

body = {

src; // string server端peerid

dest; // string client peerid

sessionid; // client端sessionid，header.sessionid填server端sessionid，用于区分过去同vport连接的过时包

}

1. ackAck = 0x22

方向：client->server

功能：确认server发来的synAck包

Flag:

BIT0：是否使用TCP通信

body = {

src; // string client peerid

dest; // string server端peerid

}

1. data = 0x30

方向：client<->server

功能：传输连接双方通信的负载数据

Flag:

BIT0: 是否使用TCP通信

BIT1: 是否是超时重发ack包，长时间没收到对方数据时启用

BIT2: 是否是SACK包

BIT3: 该包是否重发包

Body：无

Data：

为空时，这个包仅仅是个ack包，确认当前顺序收到包的seq;

非空时有两种情况:

SACK标记未设定时，是负载数据;

SACK标记设定时，Data部分存放的是当前已接收到的非连续数据包seq区间，格式为：data = [scope], scope=[fromSeq,toSeq)

注意： header.seq 和命令包不同，表示该包的负载在这个连接上传输数据流中的“固定随机值+偏移量+前序分包个数 ”；“+前序分包个数”可防止0长度的data包导致几个包seq相同；

如：

Package1.header.seq = RANDOM,

Package2.header.seq = Package1.header.seq + Package1.data.length + 1

= RANDOM + Package1.data.length + 1(前序只有一个包)

Package3.header.seq = Package2.header.seq + Package2.data.length + 1

= RANDOM + (Package1.header.seq + Package1.data.length + 1)

+ Package2.data.length + 1(前序两个包，加了两个1)

header.ackSeq表示当前收到对方连续包的最大seq

1. heartbeatReq = 0x32

方向：client<->server

功能：长时间没收到包时，发一个心跳包侦测连通性

Flag:

BIT0: 是否使用TCP通信

Body：无

Data：无

1. heartbeatResp = 0x33

方向：client<->server

功能：响应heartbeatReq

Flag:

BIT0: 是否使用TCP通信

Body：无

Data：无

1. fin = 0x34

方向：client<->server

功能：正常关闭连接，可理解为一个长度为0的特殊data包，发送这个包后不能再次发送其他数据

包结构（包括各项字段的含义）和data包完全相同

1. DHT
2. 功能介绍

DHT网络对加入的所有节点进行组织，按照相互间的逻辑距离建立路由表，通过路由表能够实现对DHT网络中的PEER进行检索。

1. PEER逻辑距离算法：XOR(MD5(peerid1), MD5(peerid2))，暂时只取了前32位
2. 路由表组织策略：先按逻辑距离给peer分级，从最高位开始计数，第一个1出现在第几位，该peer就被分到第几级路由表中，如第1级路由表中的peer距离本地peer的距离为：b1xxx，第二级为：b01xxx，以此类推；随着级别升高，其中的peer距离本地peer越近，符合特诊的peer总数也越少。但是为控制路由表大小，我们为所有等级的路由表都提供相同大小的数量限制，这样距离越近的peer在路由表中命中的概率就越大（因为同级别距离的peer总数少）；换句话说，距离越近的peer在路由表中分布越密集。
3. 查询目标peer地址
4. 先从本地路由表中找到几个距离目标peer最近的几个peer
5. 逐个向这些peer发起findPeer操作
6. 对方收到findPeer命令后，如果目标peer是自己，就把自己返回；
7. 否则，在自己的路由表中找到距离目标peer比自己更近的一组peer返回；
8. 收到findPeer命令的返回后，如果在返回值中有目标peer，则成功找到目标peer，停止搜索，并返回
9. 否则，查看在各返回peer列表中是否有还没有发起搜索的peer，如果有则转b继续搜索；
10. 如果全部findPeer命令都已经返回，且没有新的检索目标，则在所有搜索到的peer中找到距离目标peer最近的几个peer返回，部分应用可能只需要模糊查到相近的peer。
11. 路由表建立/完善/维护
12. 初始路由表由用户在DHT启动时填入DHT网络入口PEER地址；
13. 之后定期对一个随机peerid发起findPeer，把这过程中新发现的peer全部写入路由表中，并定期对路由表中的节点发ping，把不在线的peer清理掉；
14. 收到任意从其他peer发来的数据包，都将对方peer写入路由表；
15. 协议格式

DHT协议包括三个部分：固定长度的协议头header，命令主体body，通用的common结构。协议头是个固定格式的二进制缓冲区；body是个用“msgpack-lite”编码的object；common用于描述该命令发起和接收方peer信息，还包括一些用于加速完善路由表的数据，它也是一个用“msgpack-lite”编码的object。

排列顺序为：header+body+common

协议头格式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 描述 | 默认值 |
| magic | uint16 | 校验协议类型的常量 | 0x8084  常量 |
| version | Uint16 | 协议版本号 | 0x101  常量 |
| cmdType | Uint16 | 包类型 |  |
| totalLength | uint16 | body.length + common.length + header.length(36) |  |
| appid | Uint32 | 应用标识，0标识跟所有能联系的节点建立连接，其他表示只跟appid相同的节点建立连接 | 0 |
| bodyLength | Uint16 | body经“msgpack-lite”编码后的字节数 |  |
| src\_peerid\_hash | Uint32 | MD5(发送方peerid)前4字节 |  |
| online\_duration | Uint32 | Src\_peer在线时长（S） |  |
| Nat\_type | Uint8 | Src\_peer网络环境  0:unknown，1:internet，2:NAT，3:restrictedNAT，4:symmetricNAT |  |
| dest\_peerid\_hash | Uint32 | MD5(接收方peerid)前4字节 |  |
| seq | uint32 | 命令包唯一序号 |  |
| ackSeq | uint32 | 被确认包的seq |  |
| ttl | Uint8 | 该命令被转发次数，有些操作需要逐层收敛递归进行，直接中转该命令可以让命令更有效地触达，也可以分散网络压力；但考虑到泛洪问题，ttl暂时要求最多为1 |  |

COMMON结构定义：

{

id; // number 包的标识ID，请求包设定唯一值，响应包设定对应请求包的ID，用于统计往返响应速度；和seq不同的是，网络上存在的任意DHT包ID都是唯一的；而seq在发生重发时可能会存在多个包seq相同

src: {

id; // STRING peerid

eplist; // ARRAY[EP] 地址列表

services; // ARRAY[SERVICE] 该PEER提供的服务列表，SERVICE定义在后面

info; // ARRAY[key-value] 给该PEER设定的附加自定义信息

};

dest: {

id; // STRING 接收端PEERID

ep; // EP该包的发送的目标地址（src看到dest的地址），目标PEER收到后更新自己的eplist

};

nodes; // ARRAY[PEER] 可选，我看到你附近有几个节点，介绍你认识一下

}

SERVICE：

{

id; // STRING 服务ID

services; // ARRAY[SERVICE] 可选，该服务提供的子服务列表

info; // ARRAY[<key, value>] 可选，服务相关属性列表

}

各命令详细格式：

1. FindPeer：检索peer

req: 0x51

{

taskid; // NUMBER 多条协议包完成同一个任务

target; // STRING 要找寻的目标peerid

servicePath; // ARRAY[STRING]，搜索的范围，如只搜索提供了某种服务的PEER

// 如：[s1][sub1]表示在提供了s1服务中sub1子服务的PEER

e\_nodes; // ARRAY[peerid:STRING] 排除哪些节点，这些节点可能已经被检索过

}

resp: 0x52

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

n\_nodes; // ARRAY[PEER]，距离req.target较近的节点，请求方收到响应，自行根据find状态决定是否从中选取部分节点继续查询

}

1. UpdateValue，向分布式HASH表中填充数据

value结构：

table[tableName][key] = value;

以BT为例：table[infohash][peerid]=ARRAY[ip:port]

req: 0x53

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

e\_nodes; // ARRAY[peerid:STRING] 排除哪些节点，这些节点可能已经被检索过

timeout; // NUMBER 超时毫秒数，要在超时前返回

tableName; // STRING 数据存入的表名

values; // ARRAY[(key:STRING, obj)] 数据存入指定的KEY

}

// 如果req.ttl不为0，继续向下转发，转发返回后再resp，但应该限定超时时间

resp: 0x54

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

r\_nodes; // ARRAY[peerid:STRING]，ttl>0时，转发请求过的节点

n\_nodes; // ARRAY[PEER]，距离req.tableName较近的节点，没有转发过请求，请求方收到响应，自行根据update状态决定是否从中选取部分节点继续推送

}

FindValue：在分布式HASH表中检索数据

// 在DHT网络中查找一个值，请求距离key最近的value

req: 0x55

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

e\_nodes; // ARRAY[peerid:STRING] 排除哪些节点，这些节点可能已经被检索过

timeout; // NUMBER 超时毫秒数，要在超时前返回

flags; // uint32 操作相关的一些标记；如：查询结果与(tableName，key)匹配的匹配程度

tableName; // STRING 要搜索的表名

key; // STRING 要搜索的键名

}

// 如果req.ttl不为0，继续向下转发，转发返回后再resp，但应该限定超时时间

resp: 0x56

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

values; // array[(key:STRING, obj)] 查到的键值对列表，这些值所属的tableName=req.tableName，

// 对应key是最接近req.key的一组值，不完全匹配key是为了适应相似查找，比如查找距离某peer最近的SN

r\_nodes; // ARRAY[peerid:STRING]，ttl>0时，转发请求过的节点

n\_nodes; // ARRAY[PEER]，距离req.tableName较近的节点，没有转发过请求，请求方收到响应，自行根据update状态决定是否从中选取部分节点继续查询

}

1. Ping

// 检测路由表中的节点是否在线

req: 0x57

{

}

resp: 0x58

{

}

1. 打洞

// 打洞；发现一个新节点，通过当前可连接的第三方给它打个招呼，提高连通几率

// 请求帮忙给第三方打个招呼

HoleCall.req: 0x5B

{

taskid; // NUMBER 由发起方PEERID和一个序列号标识

target; // PEER 我发现了它，但不一定能联系上它，它也许能联系上我，麻烦你帮我知会它

}

HoleCall.resp: 0x5C

{

taskid; // NUMBER HoleCall.req.taskid

target; // PEER 我这里存储的目标信息

}

HoleCalled.req: 0x5D

{

taskid; // NUMBER HoleCall.req.taskid

src; // PEER HoleCall.src；谁发现了你，你去跟他打个招呼吧

}

HoleCalled.resp: 0x5E

{

taskid; // NUMBER HoleCall.req.taskid

}

HandShake.req: 0x59

{

taskid; // NUMBER HoleCall.req.taskid

}

HandShake.resp: 0x5A

{

taskid; // NUMBER HoleCall.req.taskid

}

1. BroadCast，在DHT网络中广播一个事件

req: 0x5F

{

taskid; // NUMBER

servicePath; // 同findPeer. req

timeout; // NUMBER 超时毫秒数，要在超时前返回

event; // STRING 事件ID

params; // object 事件参数

source; // STRING 广播发起者的peerid

r\_hashbits; // number 返回HASH值跟本地节点前r\_hashbits位相同的已触达节点

}

resp: 0x60

{

taskid; // NUMBER req.taskid

servicePath; // 同findPeer. req

r\_nodes; // ARRAY[peerid:STRING] 我转发给了哪些节点，

// 这些节点在req.r\_hashbits限定范围内，

// 如果这些节点在待广播节点列表中，则不再对这些节点广播

}

1. PackagePiece，给超过UDP\_MTU(1450)的数据包切块传输

// req不附带common，因为产生的分片包都是相同的COMMON头，接收方收全后复原出原始包就能得到COMMON包

// 另外，切片是二进制流格式，无法在body中封装，只能紧跟在body后面

req: 0x61

{

taskid; // NUMBER

peerid; // STRING 本地peerid

max; // NUMBER 切片包最大no

no; // NUMBER 本切片的序号

sz; // NUMBER 本切片字节数

}

resp: 0x62

{

taskid; // NUMBER

no; // NUMBER req.no

}