# 基本问题

## 数据报的设计

在UDP层实现GBN协议,就是将GBN协议的报文封装在UDP数据报中.要想从UDP-GBN协议中获取真实的数据,还需要在GBN头部中添加一些信息

## 问题涉及的实体和过程

GBNSender{

N //limit the maximum of unconfirmed number

base //confirmed base

nextsendseq //next available

}

GBNReceiver{

nextexpectedseq

}

//实际上双方都具有收发GBNPacket的功能.

//但是有所限制, client具有发送数据的功能,而server只有确认数据的功能

//为了像TCP那样可以在确认的同时发送数据,增加标志位, 标明这是一个包的类型

//类型包括:

// 载有数据,ACK为数据编号

// 不载数据,ACK为确认编号,规定datalen=0的时候表示确认,datalen>0的时候表示发送数据

// 载有数据,ACK为数据编号,但是也包含一个确认编号 == 这种数据报除非必须在收到数据之后给予回复,否则确认ACK总是立即回复的, 所以没有必要在同一个报中传输确认和数据.

~~// 为了支持这几种类型的数据报, 可以设置两个ACK头部,ACK\_CONFIRM,ACK\_DATA~~

~~// 为了实现第三种数据报的传输, 需要试着GBNSender和GBNReceiver协同工作. 规定无论是谁需要发数据, 首先查看缓冲区~~

数据包的结构

UDP()/{

ACK\_number(16bit),datalength(16bit)

data… (length <= 65535-4-8-20)

}

交互过程:

GBNSender发送数据到GBNReceiver,

如果数据序号为nextexpectedseq,则为GBNReceiver所期望的,则返回一个确认的序号的ACK,同时将自己的nextexpectedseq加1

否则,直接丢弃,并且回复一个永远使发送端不会混淆的ACK(即已经确认的ACK-1)(这一点已经确认是必须的了)

GBNReceiver如果收到ACK<base 或者ACK>nextsendseq,直接丢弃

示例:

初始 s{N=10,base=0,nextsendseq=0}

r{nextexpectedseq=0}

发送方接连发送3个数据报,没有丢失

send 0 : s.nextsendseq=1; r replies {ACK=0,len=0} ; s.base=1

…

send 2: s.nextsendseq=3; r replies{ACK=2,len=0} ;s.base=2

发送方这时再发送两个数据报,第一个丢失,第二个ACK丢失

send 3: s.nextsendseq=4, r no replie

send 4: s.nextsendseq=5, r discard this, replies {ACK=2;len=0}; s updates base fakely; then start the timer

send的具体实现:

当上层需要发送数据时, 首先检查有没有空闲的窗口位置;如果有位置,直接发送这个数据.

同时将数据存入缓存,因为是GBN,可以使用一个带有maxlen的队列,只缓存那些base(base表示下一个待确认的数据)的数据

## 如何模拟随机的数据丢失

模拟链路上的丢包,丢包率可以在发送方单独设置.如果发送方的丢失率为p(0~1),p为小数,则在发送一个数据报的时候

(确认数据报 和 数据数据报可以使用不同的丢包率, 但是原则上同一发送方的丢包率都是相同的)

在第n次发送数据报的时候,

如果 lost/all < p

如果 (lost+1)/all < p, 则选择发送这个数据报;

否则不发送这个数据报,lost++

否则不发送这个数据报,lost++

all++

注:丢失的含义就是不发送

## 差错控制

直接使用UDP本身的差错检测, 每当数据报到来的时候,首先计算UDP校验和,如果不为0就丢弃数据报.发送ACK最后一个需要确认的.

## 定时器

设置定时器的方法, 首先必须确认一点: 定时器总是用来为下一个待接收的包计时,也即是base,所以一旦确认了某个数据报并且还有其他的数据报没确认,就需要启动下一个定时器.

所以,定时器只有一个.

简单地说,定时器用来计时当前最小的待确认的数据报.

计时器的python实现:

一个定时器可能处于未定时状态, 正在定时, 已经超时

Timer结构{

time

state//NOT\_STARTED, TIMING, TIMEOUTED

isTimeout()

isTiming()

isReady()

reset()

start()

stop()

getState()

}

调用start方法的时候, 会开启一个 线程,这个线程记录自己的开始时间, 直到结束时间才返回结果

timerThread=

startTime=datetime.datetime() //now

while True:

try:

usleep(T)

except:

pass

if datetime.datetime() – startTime != T contine

else break

由于使用某些库(concurrent.futures)无法完成cancel功能,导致Timer不可取消,所以无法完成.

现在,推荐使用threading.Timer类

## 多线程模型

计时线程

发送线程

接收线程

主线程(负责同步)

主线程提供以下方法:

安全地关闭套接字

安全地关闭一个子线程

(任何能影响其他线程生命周期的事情都由主线程来做)

* 如何在线程内部结束自己的生命周期

对于定时器来说,这是错误的;因为既然已经到了超时阶段, 就没有取消线程的逻辑正确性.

此时需要做的是让线程先把当前工作做完, 做完之后才开始新的计时.

* 线程同步

设置event

如果同时有A1,A2同时进入了某个临界趋于V

让其中一个进入的同时把另一个保持在外面;

进入前等待锁, 进入时置锁, 离开时放锁

临界区域是指那些同时只能有一个线程在操作的代码区域.

A1:

myLock.acquire(True)

…

myLock.release()

A2:

myLock.acquire(True)

…

myLock.release()

* threading.Event

wait(timeout=None) 等待事件发生

* threading.Lock

acquire(block=True,timeout=None)

release()

## 向SR协议升级所要做的兼容工作

# 实验要求

1. 基于UDP设计一个简单的GBN协议，实现单向可靠数据传输（服务器到客户的数据传输）。

2) 模拟引入数据包的丢失，验证所设计协议的有效性。

3)改进所设计的GBN协议，支持双向数据传输；（选作内容，加分项目，可以当堂完成或课下完成）

4）将所设计的GBN协议改进为SR协议。（选作内容，加分项目，可以当堂完成或课下完成）

# Python多线程模型和同步

线程不能自己结束自己

# 附录:Python的随机数发生器

如果一件事需要以p的概率发生,则每次产生一个数n[0,1.0)

如果n<p(p=0时不发生) 就让其发生, 否则不发生

random.**random**()

Return the next random floating point number in the range [0.0, 1.0).

# 附录:Python的队列

**>>> from** **collections** **import** deque

**>>>** queue = deque(["Eric", "John", "Michael"])

**>>>** queue.append("Terry") *# Terry arrives*

**>>>** queue.append("Graham") *# Graham arrives*

**>>>** queue.popleft() *# The first to arrive now leaves*

'Eric'

**>>>** queue.popleft() *# The second to arrive now leaves*

'John'

**>>>** queue *# Remaining queue in order of arrival*

deque(['Michael', 'Terry', 'Graham'])

# 附录:Python的struct和ctypes

两个方法: struct.pack(fmt,\*data)

struct.unpack(fmt,packeddata)

fmt的形式:

开始符号 <(小端尾) >(大端尾) !(网络字节序) = 与 sys.byteorder有关

规定大小的符号 x 填充

c char(1)(byte)

b signed char integer

B unsigned char integer

h short(2)

i int(4)

s bytes

使用nS 其中n是个数,S表示符号类型

用法:

**>>> from** **struct** **import** \*

**>>>** pack('hhl', 1, 2, 3)

b'\x00\x01\x00\x02\x00\x00\x00\x03'

**>>>** unpack('hhl', b'**\x00\x01\x00\x02\x00\x00\x00\x03**')

(1, 2, 3)

**>>>** calcsize('hhl')

8

和namedtuple一起使用:

**>>> from** **collections** **import** namedtuple

**>>>** Student = namedtuple('Student', 'name serialnum school gradelevel')

**>>>** Student.\_make(unpack('<10sHHb', record))

Student(name=b'raymond ', serialnum=4658, school=264, gradelevel=8)

>>> import ctypes

>>> ctypes.c\_ulong(-1) # stuff Python's -1 into a C unsigned long

c\_ulong(4294967295L)

>>> \_.value

4294967295L

# 附录:Python3的enum

import enum

enum.Enum(‘Name’,”va1 val2 … “)

# 附录:socket实用工具集

## 如何从UDP socket中读取所有数据?

|  |  |
| --- | --- |
|  | Each read from UDP socket de-queues *one whole datagram* off kernel socket receive buffer no matter what's your userland buffer size. That is:   * If your buffer is bigger than the next pending datagram, you'll read less than your buffer size. * If your buffer is smaller, you'll read your buffer size worth and the rest of the data is discarded. * You can set MSG\_TRUNC option in the flags, so [recv(2)](http://linux.die.net/man/2/recv) will return the whole datagram length, not just the part you read into your userland buffer.   Hope this help |

关于MSG\_TRUNC:用于按时是否数据被丢失

|  |  |
| --- | --- |
| 3down voteaccepted | MSG\_TRUNC indicates that the buffer space provided for receiving was insufficient, so that some of the packet data were lost. This flag is used when you want to discover how much data was truncated. The recvmsg function will set this flag (msg\_flags) for the structure msghdr (used for sending UDP packets) |

最一般的解决方法:设置BUFFER为65535,因为UDP不可能发送 65535还多的数据.

UDP的好处是, 一次recv/send即可发送所有数据.

经过测试,发送 65535-28个数据刚刚好, 如果超过一个都会报错:

OSError: [Errno 122] Message too long

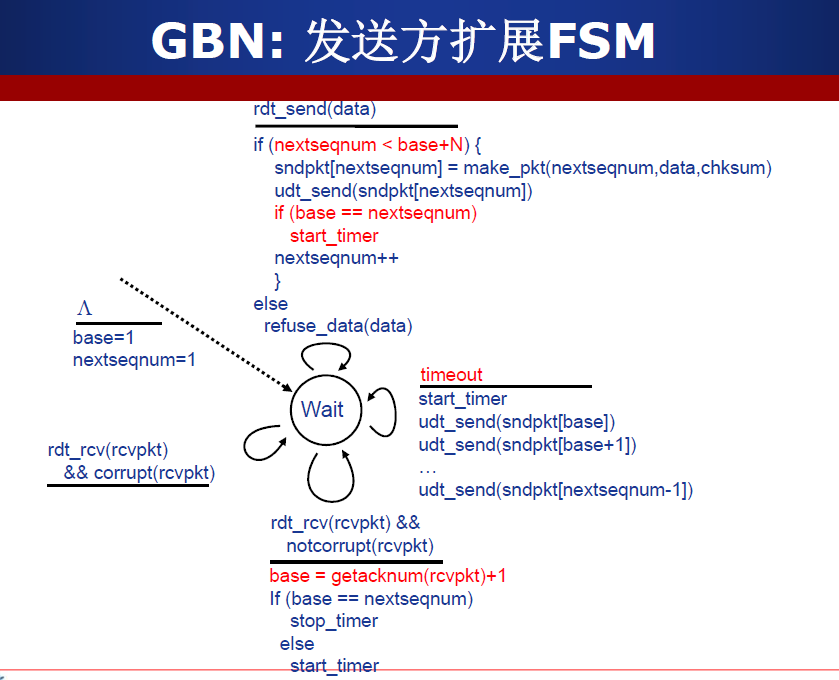
而原本的UDP的头部只有8字节,为什么多出20个不能用的字节?

这是因为UDP是包装在IP中的,IP中的有一个数据报的总长度,由于IP首部长度至少是20字节,所以在填写UDP总长度的时候,必须保证将来在填写IP数据报的时候不会溢出.

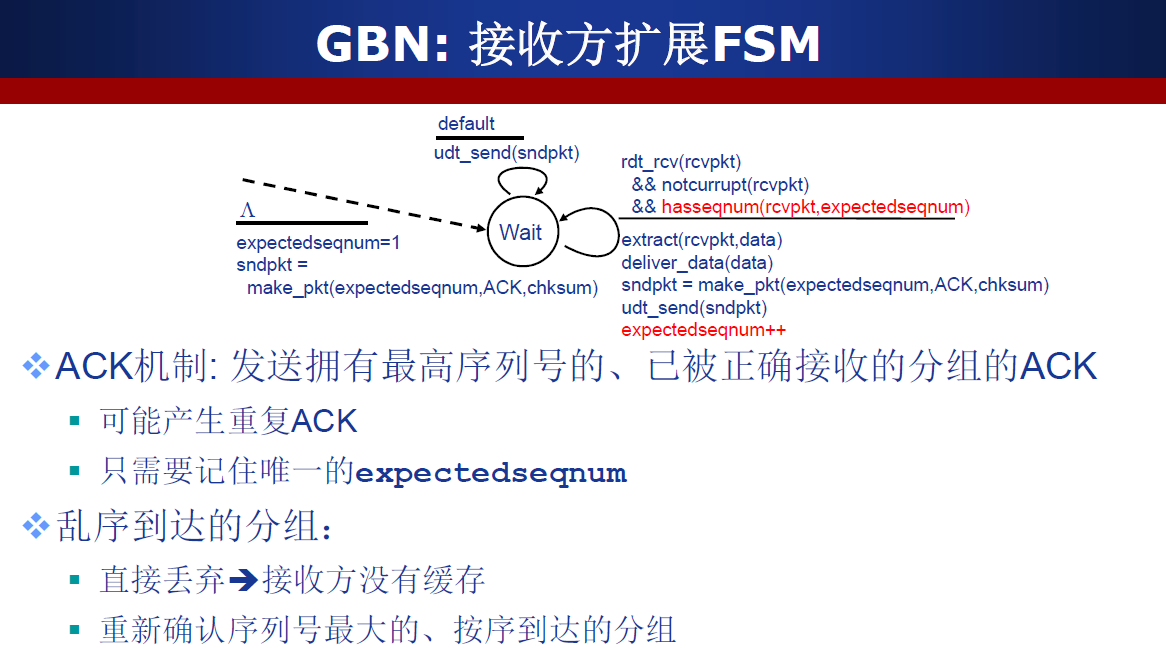
# 附录:GBN协议的基本内容

## 工作流程

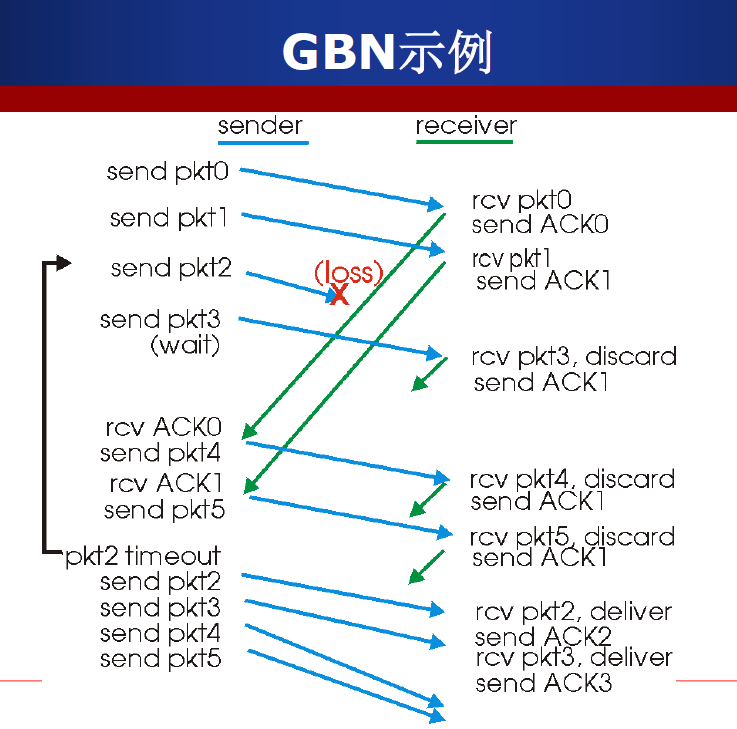
发送方



接收方

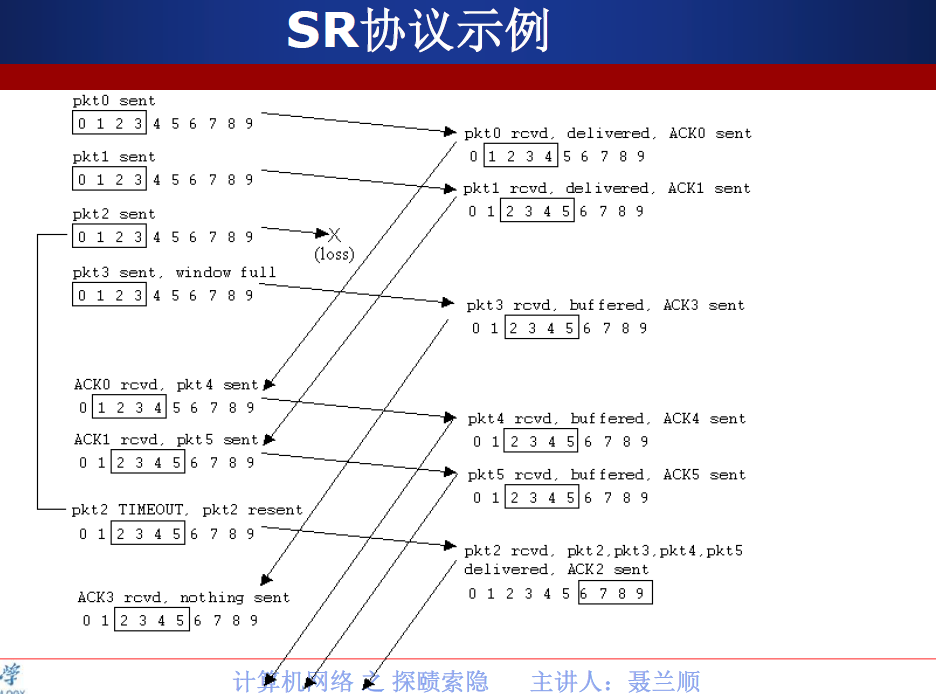


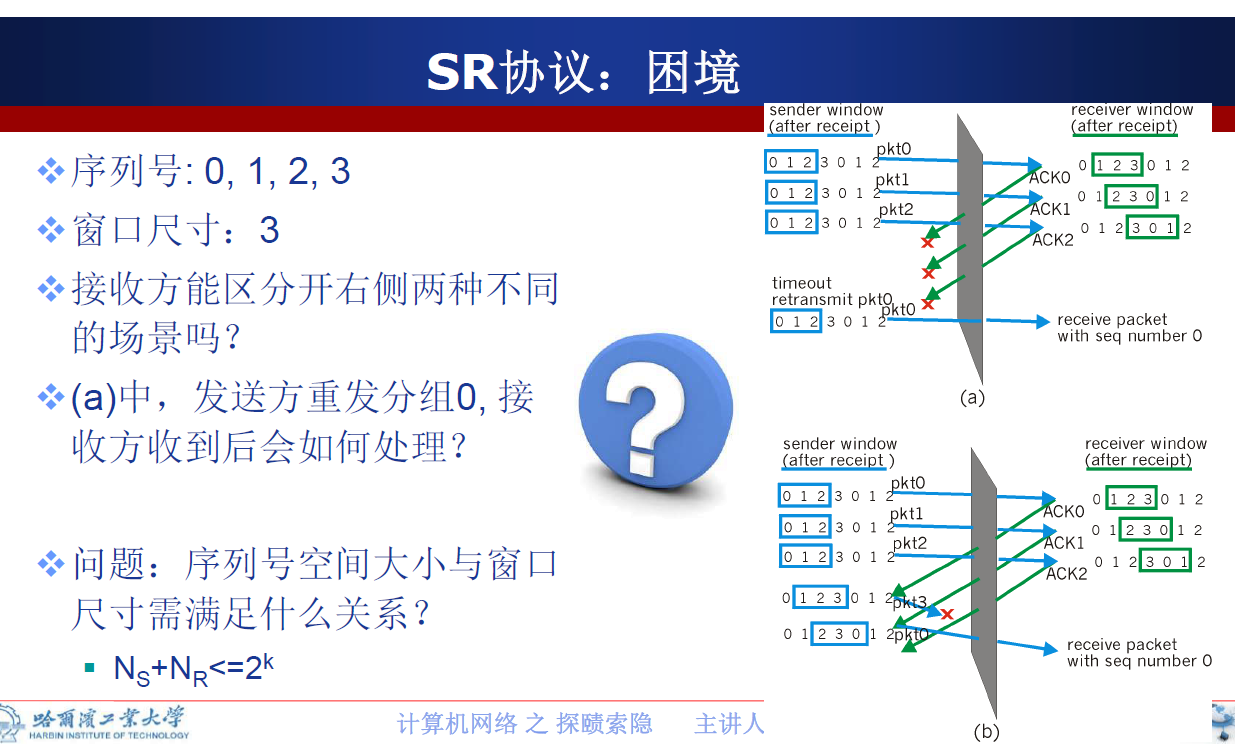
## 一个示例



# 附录:SR协议的基本内容







(请模拟这个困境的发生,或者给出发生条件并证明)

# 附录:UDP socket的使用步骤

客户端:

client=socket(socket.AF\_INET,socket.SOCK\_DGRAM)

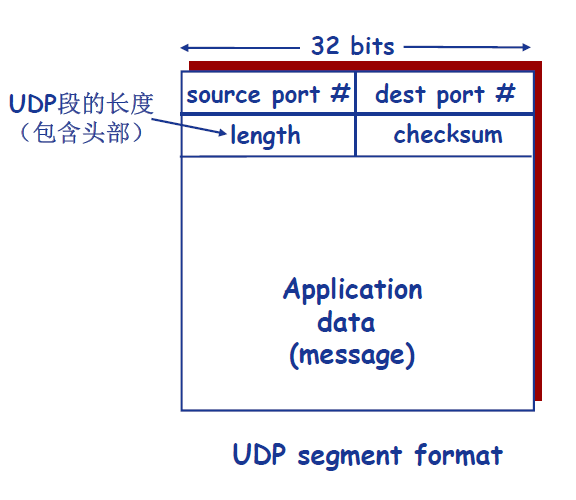
client.sendto(data,receiver)

服务器端:

server=socket(socket.AF\_INET,socket.SOCK\_DGRAM)

server.bind( (addr,port))

# 附录:UDP报文格式



source port是可选的, 用于接收方进行回复.

长度是8bit的倍数(字节作为单位),并且必须包括头部的数据(所以长度最小值是8)

## UDP的差错检测机制

参考: <https://tools.ietf.org/html/rfc768>

UDP校验和计算包括: 头部+IP头部+数据

UDP伪头部,作用:检测被错误路由的数据报(TCP也采用伪头部的方式)

0 7 8 15 16 23 24 31

+--------+--------+--------+--------+

| source address |

+--------+--------+--------+--------+

| destination address |

+--------+--------+--------+--------+

| zero |protocol| UDP length |

+--------+--------+--------+--------+

如果checksum=0,表示不需要检错.

如果需要对0进行检测,就将0发送为0xffff

术语:Complement:补码

one’s complement反码

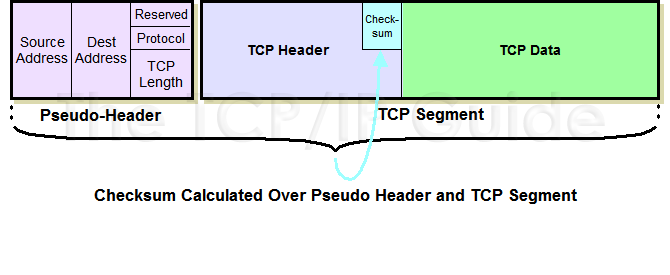
one’s complement sum就是指所有数的求和,每次求和如果产生进位,在下一次运算时进位被加入. 对于多余2组的数据,也等于一次加完之后,截断进位再加,直到进位为0

伪头部的段

* source address: 32 bits/4 bytes, taken from IP header
* destination address: 32 bits/4 bytes, taken from IP header
* reserved: 8 bits/1 byte, set to all 0s.
* protocol: 8 bits/1 byte, taken from IP header
* length: Because UDP header has a length field that indicates the length of the entire datagram, including UDP header and data, the value from UDP header is used. Note that this is different from TCP pseudo header, which is computed on the fly(Because TCP does not have such field). But they both indicates the header+payload length.

如果数据部分的最后一个字节不足以构成两个字节,就需要在尾部而不是首部补0

## TCP校验



## Python验证UDP计算过程

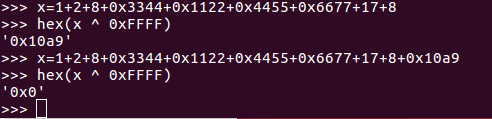


IP伪头部: 协议=17(udp) src=0x11223344 dst=0x44556677 长度=8

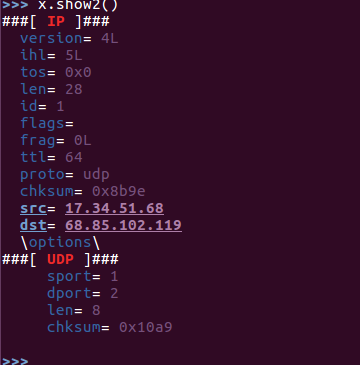
UDP头部 sport=1 dport=2 len=8

数据 NULL

计算过程:



实际结果



计算正确.