transport services and protocols

在不同的host的**app processes** 之间提供logical communication

transport protocol在end system 运行

sender: segment切割 app message， 传递给network layer

receiver:把segments片段重新装配成message，传递给application layer

一个app可以使用多个transport protocol:例如internet就使用了TCP与UDP

Network layer: **hosts**之间的logical communication

transport layer: **process**之间的logical communication

依赖，增强network layer的服务

Internet transport-layer protocol

TCP: reliable, in-order delivery

congestion control拥挤控制

flow control

connection setup

UDP: unreliable, unordered delivery

不能 delay guarantee 延迟确保

bandwidth guarantee 带宽确保

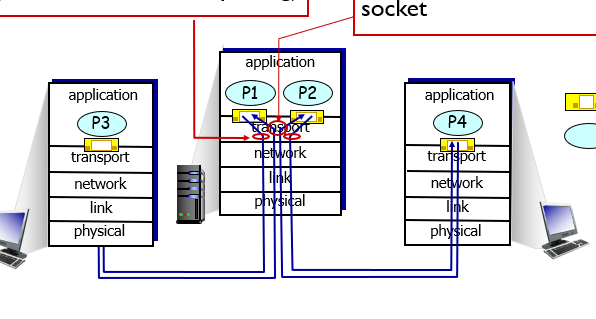
Multiplexing/demultiplexing

multiplexing at sender: sender的多路技术

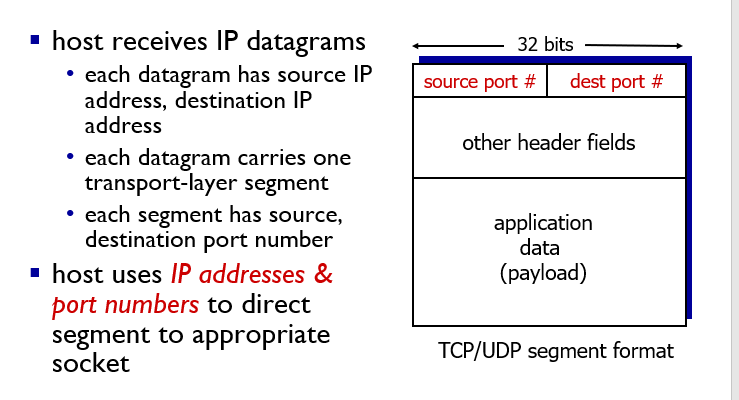
掌管多个socket的数据，增加对应的transport header(在demultiplexing中使用)

demultiplexing at receiver: receiver的多路分解技术

用header info 来把收到的segments传递给正确的socket



demultiplexing 工作原理



host收到 IP datagrams 数据包

每个datagram都有source ip address, destination ip address

每个datagram 都带有一个transport-layer的segment

每个segment都有source destination port number

host使用ip address与port number来把segment 引导到合适的socket

Connectionless demultiplexing

创造socket的时候会给他附一个port #(host-local而并不是目的地)



然后在创造发给UDP socket的datagram时，指定

目的地的IP ADDRESS， Port number

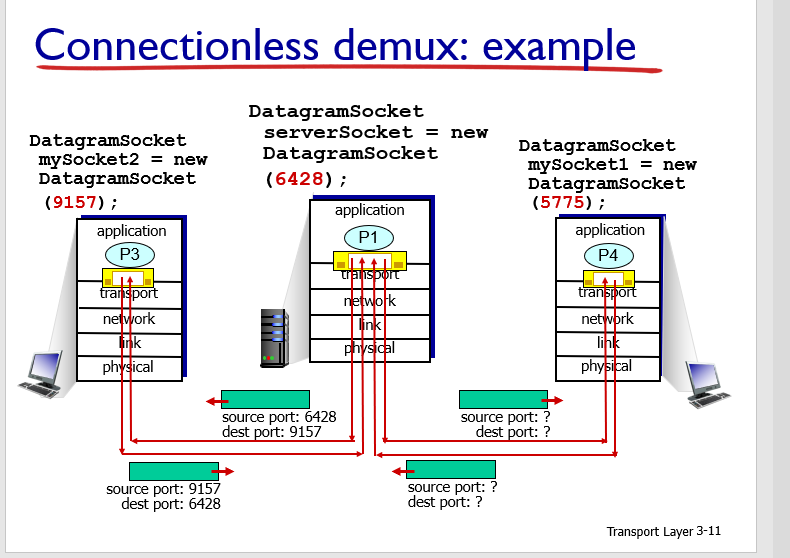
当host收到了UDP segment

检查destination port number

把UDP segment发到对应的有这个port number的socket

拥有不同source Ip address, 但有相同destination port number的 ip datagram，将被送往destination 的同一个socket

connectionless demux :无连接多路分配器



创造的socket时是自己的port

所以依次port是9157， 6428(server), 5775

然后发送信息指定dest port

发送的信息发给destination port，x

好多没看

UDP：User datagram protocol 用户数据包协议

没有装饰，最简单的· internet transport protocol

ConnectionLess:

不需要Handshaking因此没有handshaking delay

每个UDP segment自己管自己的

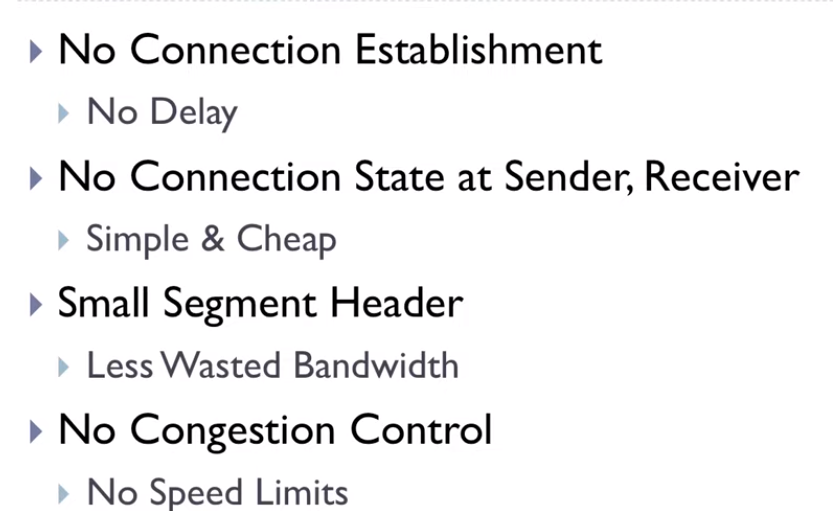
为什么使用UDP

第一，没有connection，没有handshaking delay

第二，simple， sender与receiver不需要connection state

第三, smaller header size相较于TCP

第四： 没有congestion control //拥塞控制，可以随时爆发传送速度根据需求



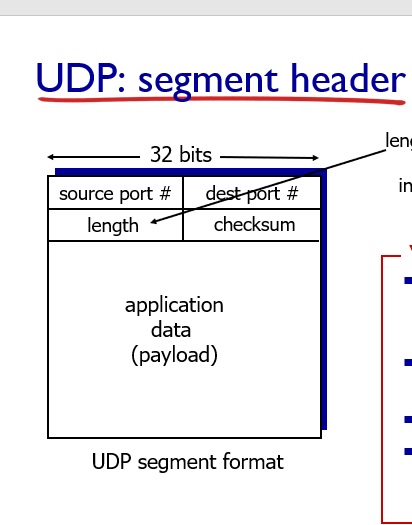
但是UDP并不是特别稳定的transfer方式

我们可以

1.在application layer增加稳定性

2.application-specific error recovery application layer的错误恢复

意思就是你选用UDP作为transport layer的时候， 你的application layer会进行对UDP的error recovery，例如你丢失了一个packet,application layer会重新再request一次



UDP一个segment(packet)的格式，

16bit source por number

16bit dest port number

16bit length: 就是整个包的长度，包括header与application data

但是大小是byte， 要除以8，所以整个header的length就是8byte, 64 bit/8

CheckSum: 根据segment data所计算出来的一组数据，这样收到的人可以根据segment data再算一次checksum,两个checksum一比较，知道有没有Lost

UDP checksum

目的：检测到传输的segment的error

Sender:

把setgment把的内容以及header fields看做一串16位二进制数

checksum: 把所有二进制数加起来，取后16位

sender把checksum值放入UDP checksum field

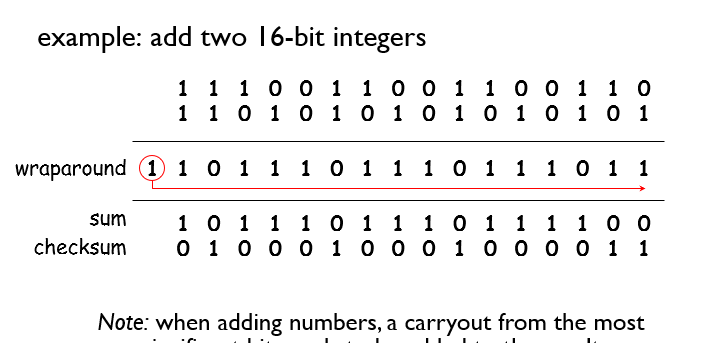
receiver:

计算收到的segment的checksum

对比

如果如果不相等，代表检测到error

如果想等，代表没检测到error，但是还是有可能有error



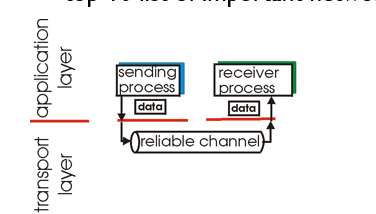
例如这个，两个加起来，取后16位，自己加出来的结果是sum，与checksum不同

.

3.4

Principles of reliable data transfer

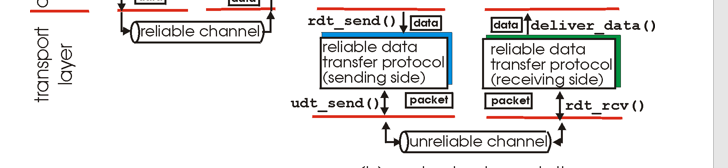
我们怎么把data从sending process通过一个reliable channel 传输给receiver process

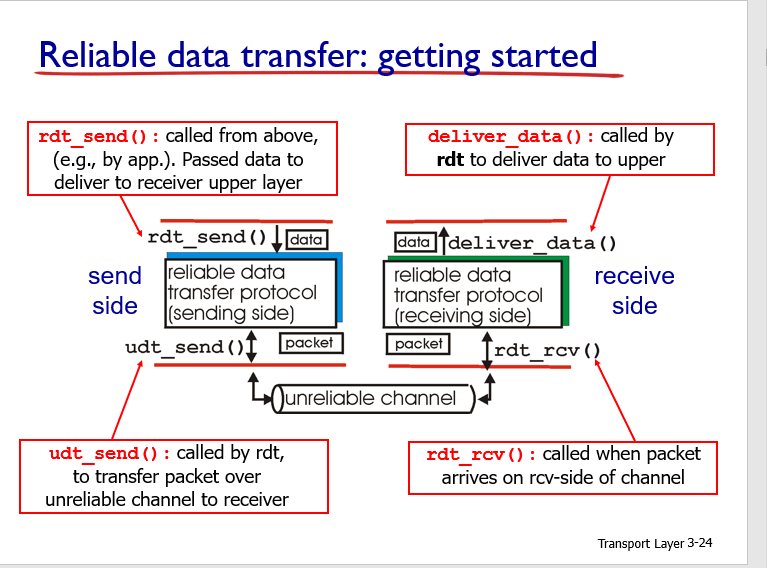


sending process 与receiver process是application layer的，他们负责把data push到 transport layer

然后 transport layer负责构建一个reliable channel

然而实际上，我们很多时候要把data push到unreliable channel，我们要怎样保证接收到正确的DATA





transport Layer装的四个function:

rdt\_send()，由application layer所call， 把信息传送给transfer layer的上层

然后经过reliable data transfer protocol,转移到下层

udt\_send:由rdt所call，把packet从unreliable channel传递给receiver

rdt\_receiver():当packet 到了另外一端，rdt\_rcv()就会被call,

deliver\_data()被RDT所call，把信息传送给更上层的application layer

简单地说上面两个负责application layer与transport layer的传递

他们通过transfer protocol切割成packet，

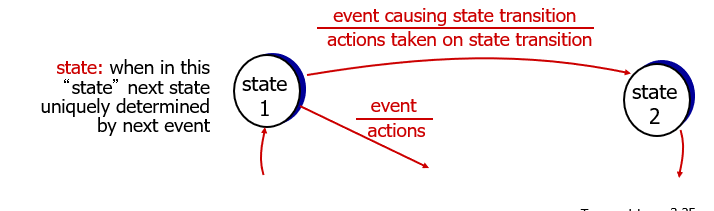
然后udt\_send()与rdt\_rcv()会接受传递信息

我们关注两个蓝绿盒子，（sending/receiving side transfer protocol）,看看这一系列function calls是怎样通过蓝绿盒子implemented的

我们会一步步的模拟构建sender/receiver rdt protocol的过程·

只考虑单向data transfer(但是control info是双向的)

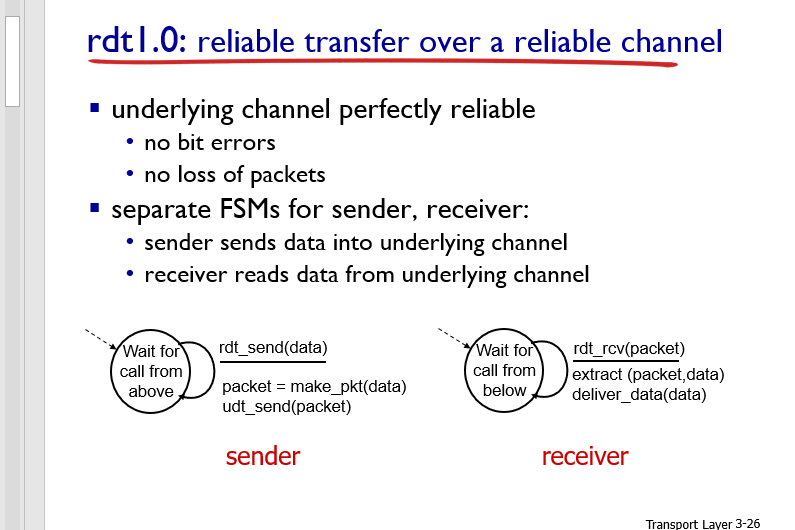
用一个FSM 有穷自动机来表示各个状态



横线上面是导致state改变的event

下面是改变state时需要做的操作

rdt 1.0



假设channel是完全reliable的

没有bit error,没有packet loss

这样我们两个FSM并没啥关联性,sender send data, receiver receive data

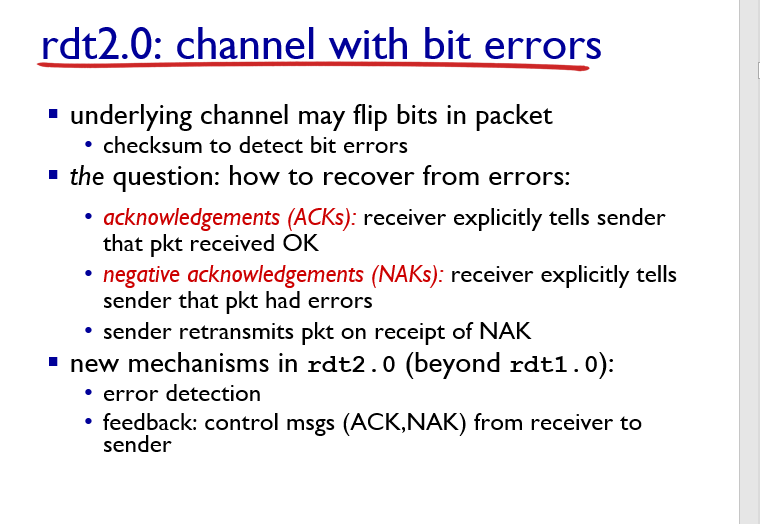
rdt\_send(data)会导致进入下一个阶段（还是他自己）

然后改变阶段时会进行

把data转为packet

使用udt\_send(packet)

rdt 2.0



channel只有bit error，没有packet loss

怎么发现error: checkSum

怎么recover from error

receiver根据收到的packet发送ACK/NAK给sender

acknowledgements(ACKs), receiver显式的告诉sender收到的packet没问题

negative acknowlegements NAKs, receiver显式告诉sender packet有error

如果收到NAKs，sender重新传输packet

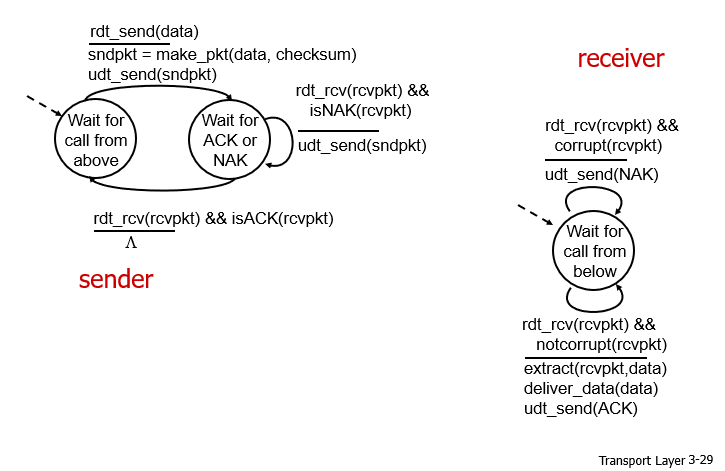
2。0在1.0的基础上新添加了

error detection

feedback: 发送control messages给sender

概念  
corrupt:收到的包CHECKSUM对不上

notcorrupt:收到的包checksum对上了



Sender在wait for call from above状态：

如果遇到rdt\_send(data)，开始进行state转换

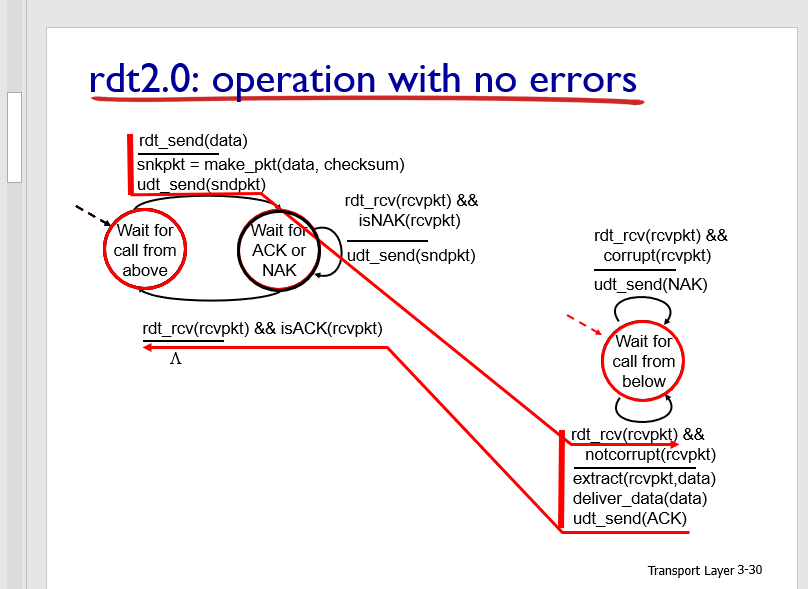
制造packet=make\_packet(data,checksum)

udt\_send(packet)//把packet通过unreliable cHannel 发出

发完Package以后进入wait for ack or nak状态

如果rdt\_rcv到receive\_packet并且是NAK， 重复发出send\_packet

如果rdt\_rcv收到的receive\_packet是ACK，什么也不做，回到wait for call from above 状态



receiver

只有一个状态就是wait for call from below

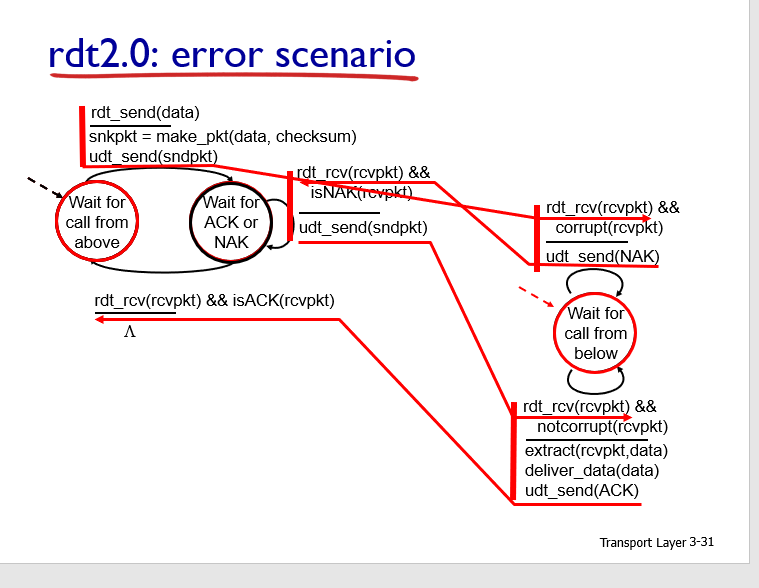
如果receiver收到了rcvpkt(receiver的rcvpacket是sender的sndpacket)

并且检查过了没坏，

那么就抽理数据，并且把deliver\_data(data)

然后send ACK

SENDER wait到了ACK，进入WAIT FOR CALL FROM ABOVE状态



如果收到的rcvpkt且是错误的，sendnak，并且还是保留在这个阶段，

SENDER wait到了NAK，重新发出 sndpkt， 然后继续wait for ack or nak

一直循环，直到收到not corrupt

rdt2.0有一个重大的问题

如果ACK/NAK也corrupt了呢。

sender不知道receiver到底发生了什么 //既不属于ACK，也不是NAK

不能只是重新发送，因为可能重复duplicates// receiver收到了正确的包并且已经拆解成data，只是ACK出了问题，这时候你又发了一个，就有重复的包·

handling duplicates:

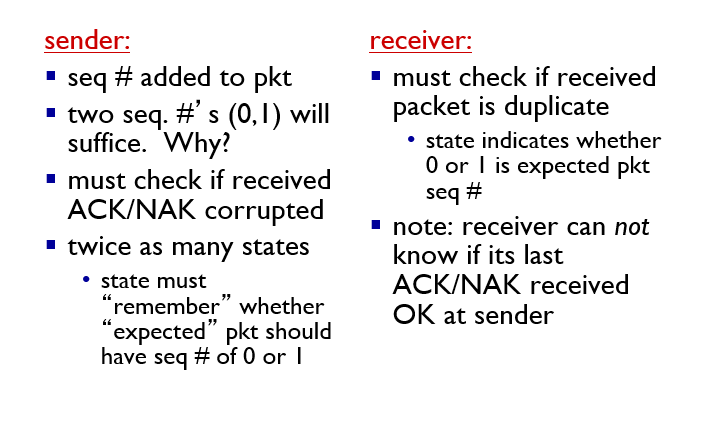
sender 重新send现有packet当ACK/NAK corrupt(损坏)

但是sender给每个packet增加一个**sequence number序列号**

receiver discards（不继续向上传给application layer） duplicate packet 重复的包

stop and wait:

sender发出一个packet，等response



sender给pocket加上seq#

只要0与1两个sequence就够了

必须check ACK/NAK对不对

state数量翻倍，因为需要记住expected 发出的packet应该是seq 0还是1

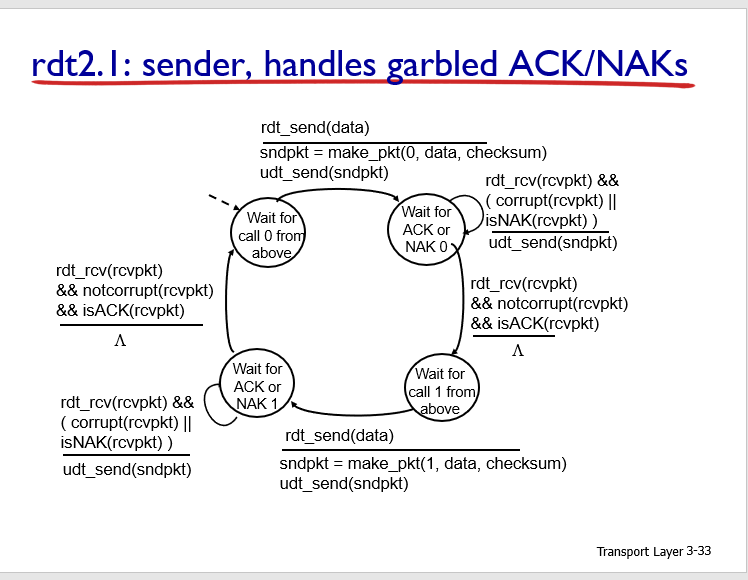
//一组是0,一组是1

receiver，必须检查Packet是否多余

通过比较0与1是否是期待的packet

也要翻倍

注意：receiver本身并不知道踏上一个ACK/NAK是不是成功被sender接受



上面还是原来的

不过加了一个0

已开始进入的状态是wait for call 0,

如果等到application-layer rdt\_send(data)

制造一个序列为0的packet，并且通过udt\_send发出

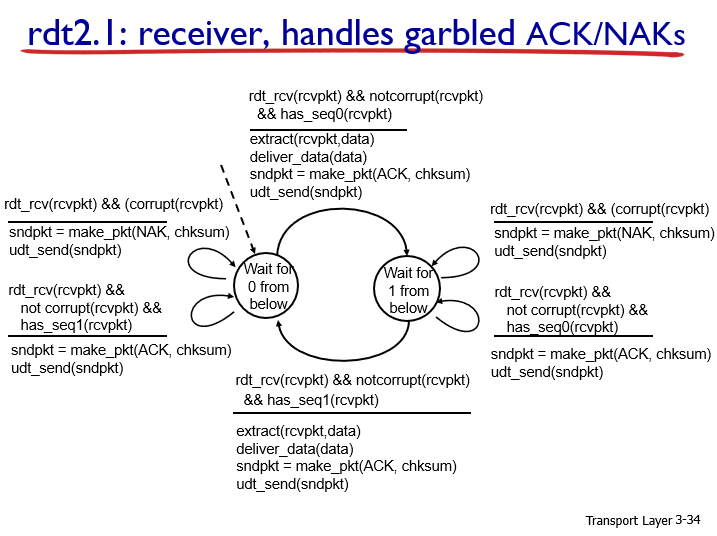
等待ACK/NAK 0 ,

rcvpkt是NAK或者是corrupt，重新发出sndpkt(这时序列是0)

如果是ACK，且ACK没坏，进入waait for call1 状态，如果 收到application layer让你send data， 发出1 data包

等待ACK or NAK 1

rcvpkt是NAK或者是corrupt，重新发出sndpkt(这时序列是1)



receiver

一开始是wait0状态

两种不变：

如果收到的packet是损坏的，重新发packet告诉他NAK//NAK没有序号，完全取决于sender的状态

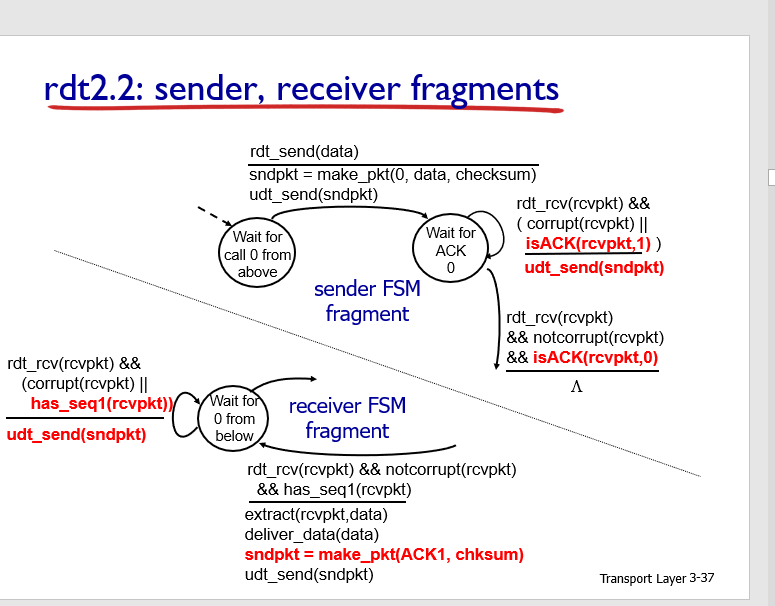
如果收到的packet没有损坏，但是序号是1（duplicate），//**说明sender 现在状态是1**，发出ACK ， 这时sender就会进入下一个状态0，

如果收到的是0，没有损坏，抽取信息，发出ACK， 进入wait for 1..

RDT2.2

nak-free protocol

与2.1大致相同，但是舍去了nAK, receiver每次都会发出上次成功收到的packet的number，附着在ACK中



sender等待，第一次发出0，然后等待ACK0

receiver 如果收到PACKET1,代表duplicate了,还是停留在rcvpkt的状态， 发出sndpkt，（ACK0发出ACK1代表原来的NAK）

然后收到ACK1，发现不是ACK0，重新发出 Packet0, 这时RECEIVER如果收到了packet0, 抽离，把sndpkt改成0，进入WAIT FOR 1 FROM BELOW..无限循环

总而言之，如果成功，例如成功收到0，记录下0， 然后进入状态1，如果没有收到状态1，那么就发回ACK0，

rdt3.0：

引入了packet loss的概念

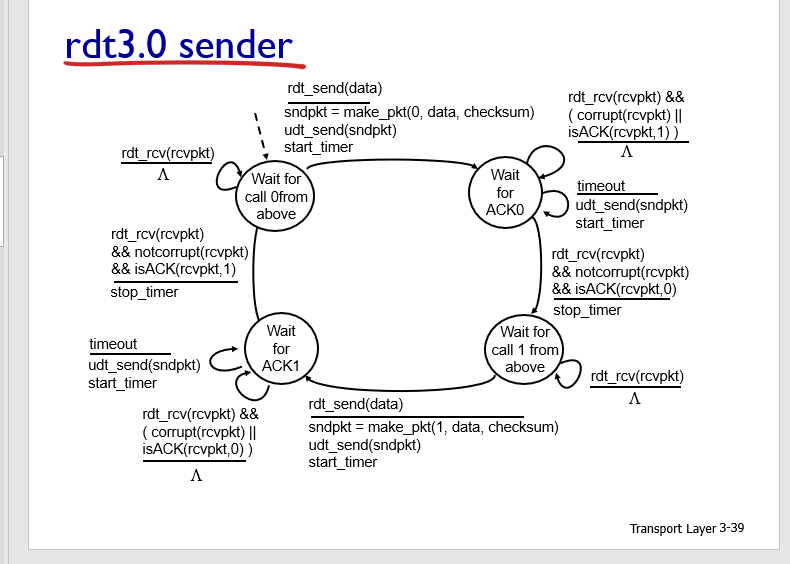
大概方法：

sender等一段合理的时间给ACK，

如果没有收到ACK(代表packet lostt receiver 没收到，orsender没收到receiver的ACK)，就再发一次

如果packet ACK只是delay(no lost) ,再发的就会duplicate，但是因为seq#,我们完全可以辨别duplicate

必须给sender增加一个·countdown timer



rdt 3.0 sender

如果有人要senddata,

创造一个packet0, 开始timer ，

如果收到的ACK有问题，或者是ACK1， **什么也不做**

直到时间为0，重新send packet0,start\_timer,

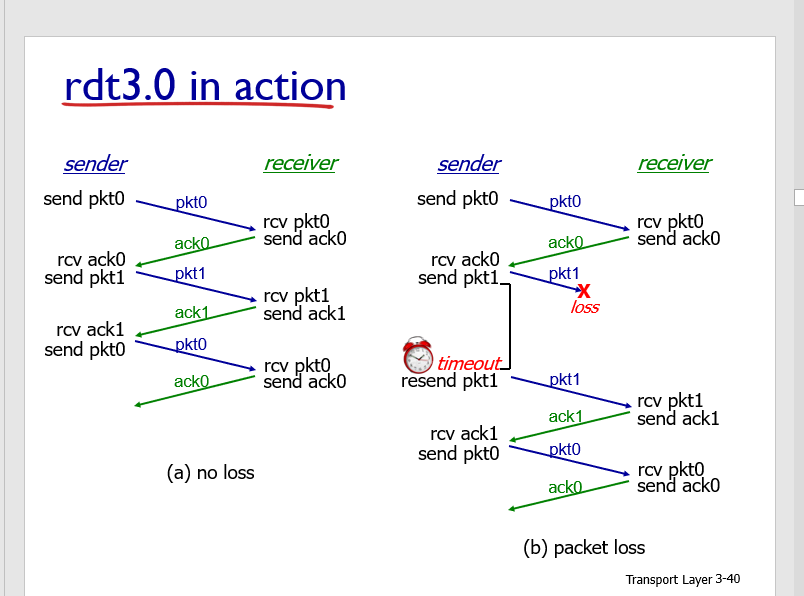
反正隔一段时间send一次

除非收到没损坏的ACK0，停止timer

进入等待call1,如果在等待call1的阶段收到了迟来的ACK，不管

只有application layer告诉你要send data了，发出packet0,进入wait for 1

.........



sender send pkt0

receiver receive 了pkt 0, 发出ack0,

成功接受ack0,发出pkt1,收到pkt1,发出ack1

收到ack1,发出pkt0

.......

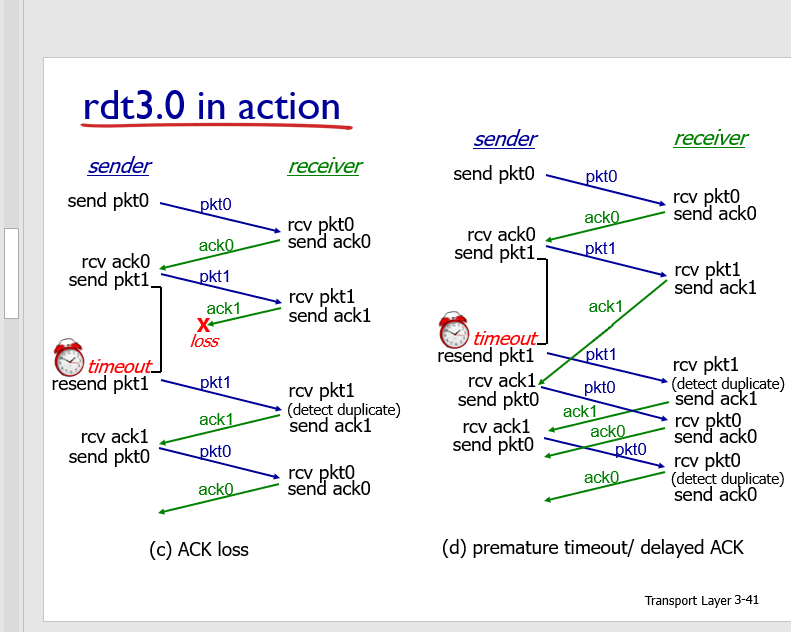
如果有packet loss

发出pkt0, receiver收到Pkt0,发出ack0

收到ack0,发出pkt1, receiver没接收到·

sender等了一段时间，重新发出pkt1

j继续循环



ACKloss

sender send pkt0

receiver接收到 pkt0, send ack0

sender rcv ack0, send pkt1,

receiver rcv pkt1, send ack1

但是ACK lost了

因此time out，重新发pkt1

receive pkt1，是duplicate，因此发上次成功提取的ACK,也就是ACK1

成功循环

如果只是delay

sender send pkt0

receiver 收到pkt0, send ack0

收到ack0,发送pkt1,

receiver收到pkt1，发送ack1

这个ack1延迟了，

因此sender没收到，再次发送pkt1,

receiver收到pkt1,是duplicate，再次发送过ack1

sender收到ack1,（他现在wait ack1的状态）， 成功，发送pkt0,

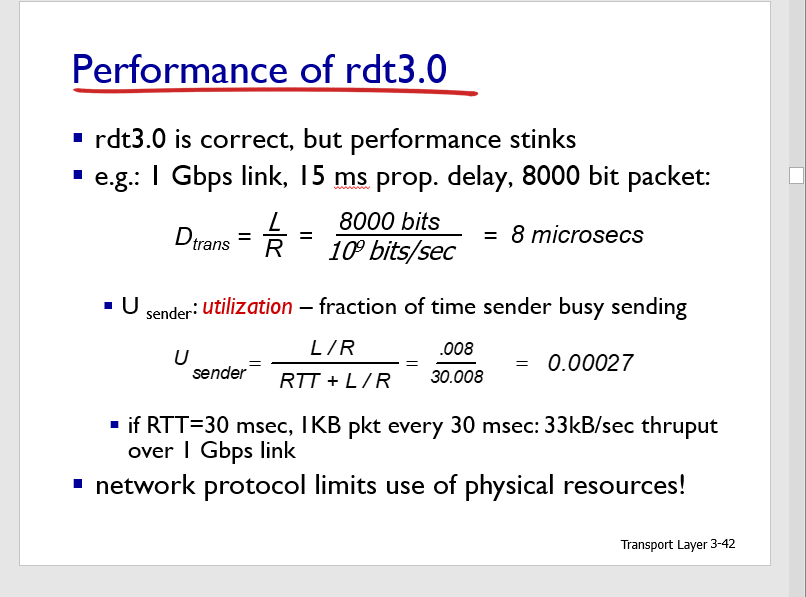
receive pkt0,发送ack0

这时收到了延迟的ack1，再次发送pkt0,

receive pkt0，是duplicate，再次发送上次的ack0

因此不会影响

rdt3.0能解决delay与packet loss问题，但效率比较低



d/trans,把packet传到线上的时间

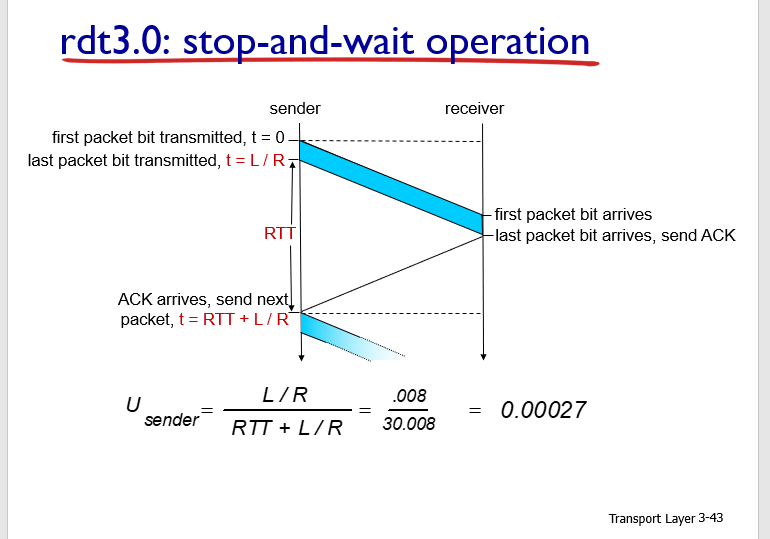
Usender:利用率：想要发送data时间，以及实际所用时间的比较

假设Dtrans很快

但是RTT很慢（一次发送到接受回应的时间）

那么速度就会很慢

STOP-AND-WAIT模式（慢）



也就是说我们实际上只用了t=0到t=L/R的时间在传递packet，

剩下的时间我们都在WAITING ack

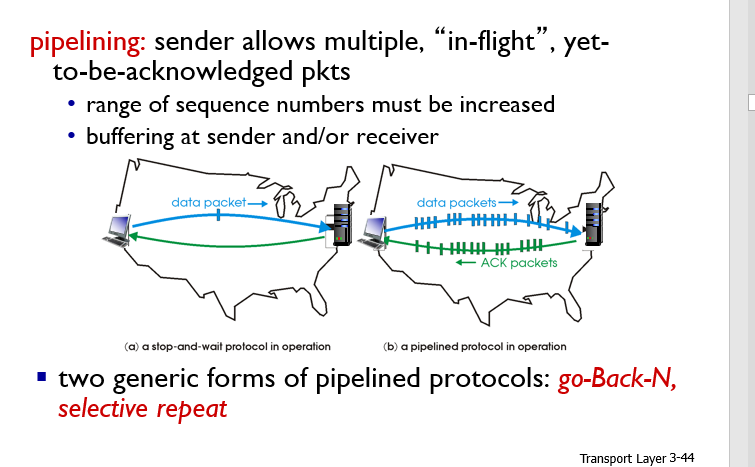
效率相当于竖着看 蓝色（L./R）/蓝色加白色(L/R+RTT)

因此我们使用pipelined protocols

sender发出多个等待着ACK的packets

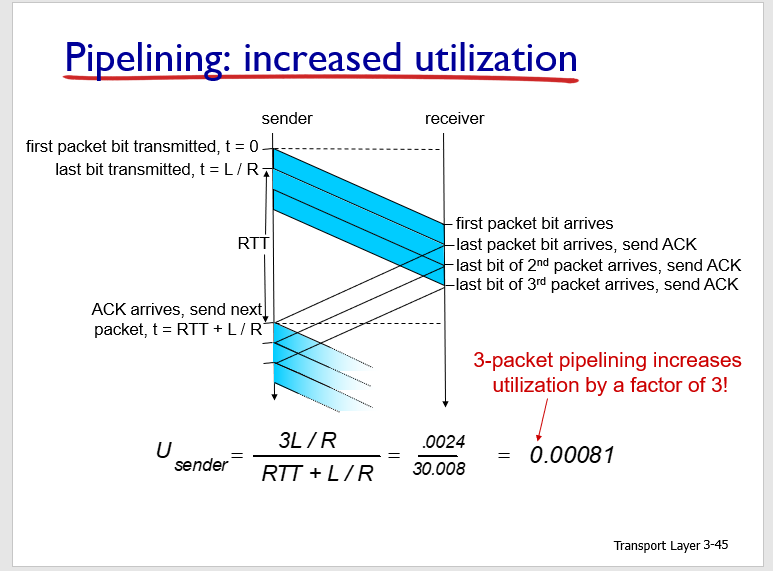
当然肯定要增加sequence number数量

需要用buffer在sender 与receiver， 来保持信息



可以发现一条线路密了很多

两种Pipelined protocols: go-back-n, selective repeat



我们如果同时发三个packet，利用率就翻了三倍

Go-Back-N

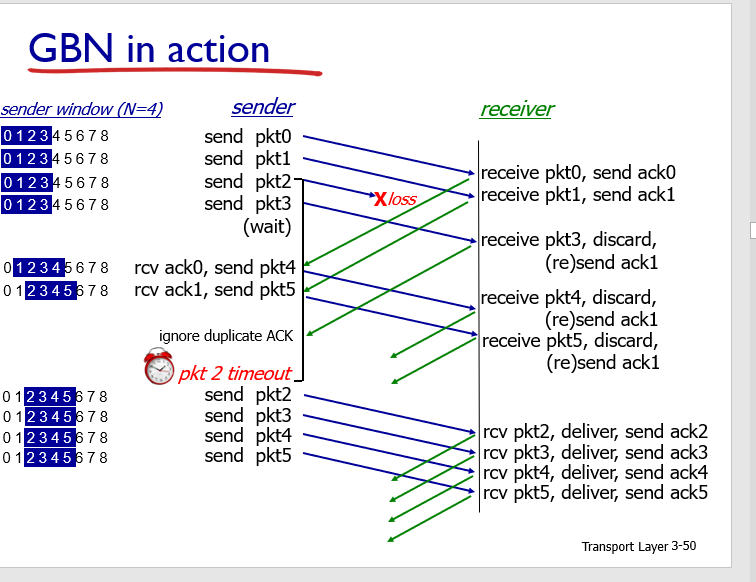
sender可以有同时有上限为N个的没有被ack的packets in pipeline

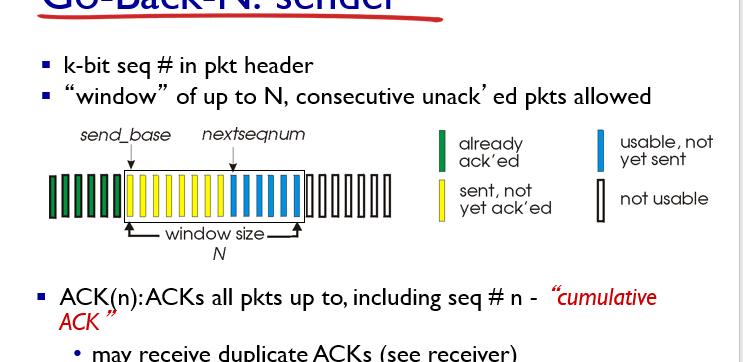
receiver

只发送连续的ack，例如成功接受012，这是准备接受3，但是收到了4，忽略，重新发上一个成功的2，然后让sender发3

sender有一个timer记载着最老的unacked packet

如果超出时间，把所有unacked packets重新传一遍





所以被承认的绿色部分是连续的，window平稳右移

Selective repead:

sender可以有N个unpacked packets在pipeline

receiver给每个正确的packet发单独的**individual ack**

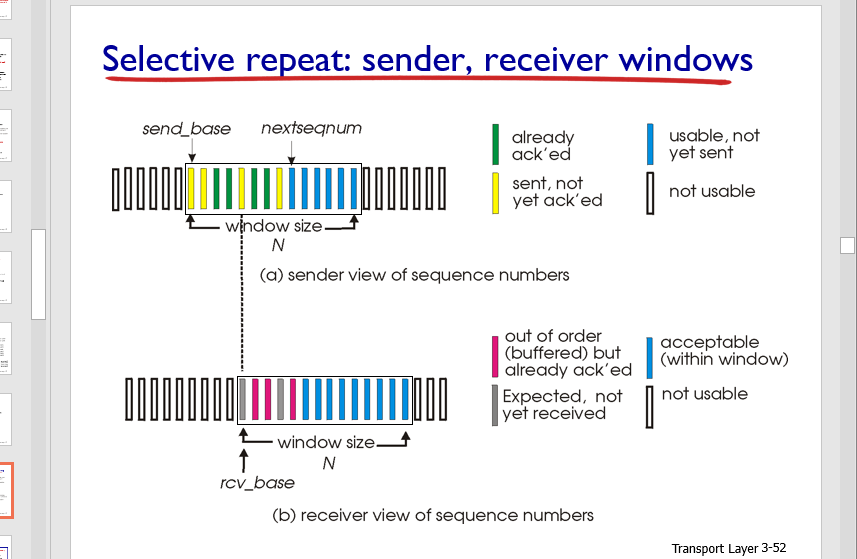
并且缓存需要的packet数据，为了最终传输给上层application layer process

sender给每一个unackd packet都保持着一个timer，如果超过时间，发送指定packet

sender方的window

是连续的N个seq number

只有有限的unACKed packets 会被发出



你可以发现windows理由的已经被ack，不用发出

receiver window:

红色的代表已经接收到得data，但是还是不连续的，因此暂时被缓存

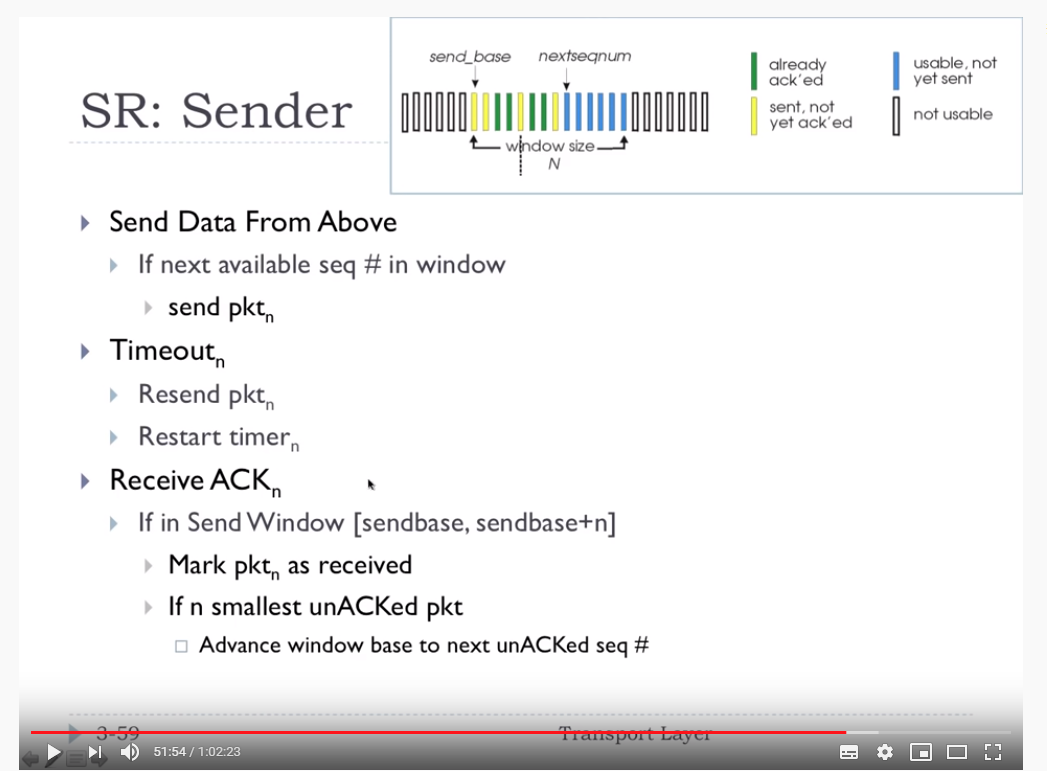
蓝色的代表可以接收到的packet阀内

灰色代表我现在需要，但是还没接收到

如果说我们接受到了第一个灰色的

那么灰变红，三个红色的就会被上传，不用缓存了

window移到第二个灰色的那里



sender的伪代码

分为三种情况

如果需要send data,且他的sequence number在window里(useable)

send packet n

Timeout:

重新send packet n

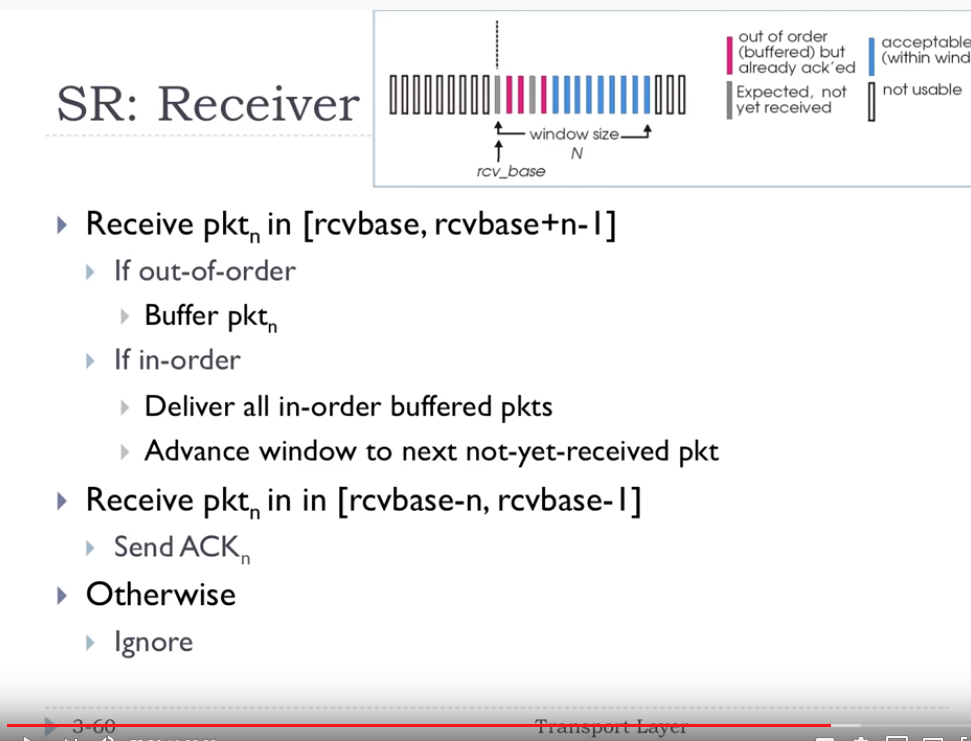
重新设置timer

收到ACKn

如果在window里

把这个packet标记为received

如果是最左边的unacked packet，把整个window右移到下一个unacked



Receiver

如果收到了是在receiver windows里，

如果还是没有顺序，缓存

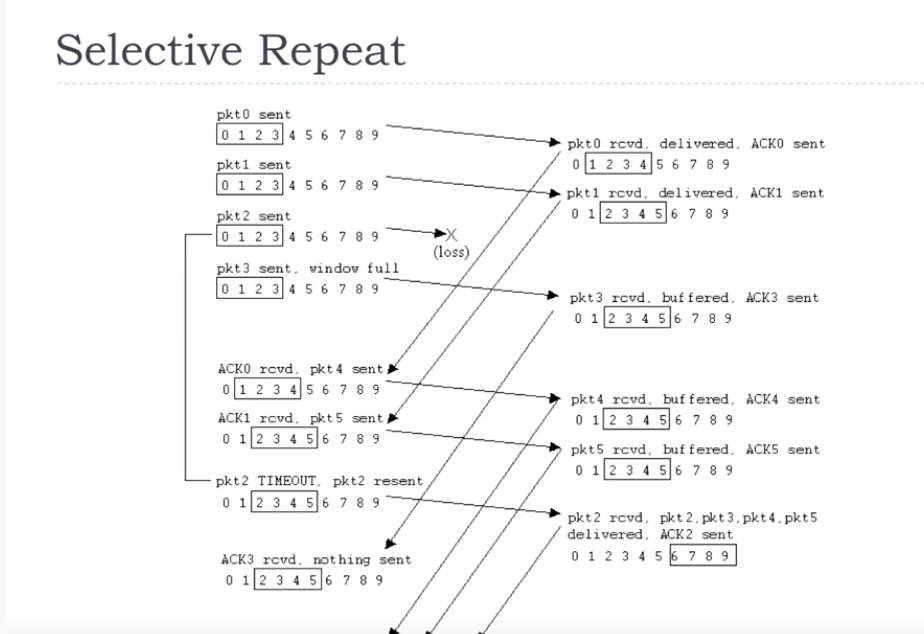
如果in-order //就是是最左边

把所有红色的packet都上传，

把window推到下一个灰色格子

同时如果收到packet,send ackn

否则，ignore



例子

左边sender window n=4

那么自然发送packet 0123,

receiver 收到0，发送ack0, in order, 右移一格

收到pkt1,发送ack1, in order，右移一格

发送pkt2, lost了，

收到pkt3,因为没有pkt2, buffer,发回ack3

然后收到ack0的瞬间，就可以左边右移一格

可以发送pkt4,

收到pkt4,buffer, 发送ack4

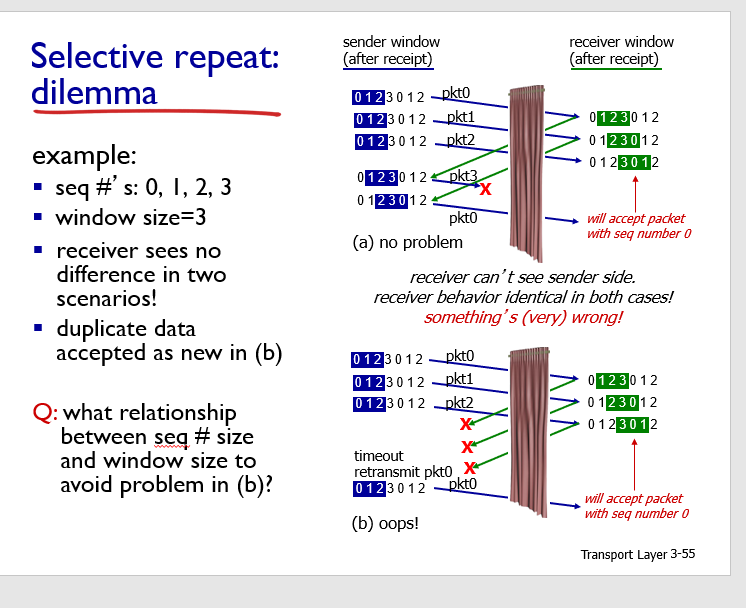
收到ack1,右移，可以发送pkt5,

收到pkt5,buffer,发送ack5

这时发现pkt2 timeout，重新发送pkt2, pkt2 received，

in order，全部上传2345，发回ack2

收到ack2，右移



window size必须小于等于 sequence number的一半

Selective repeat