Computer Networks

Chapter 6: Transport Layer

(Version May 15, 2023)

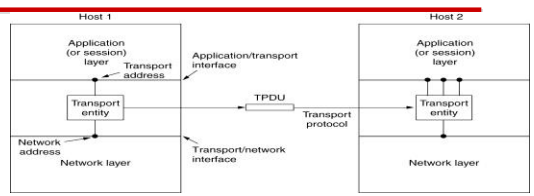
Xiao Mingzhong

CIST, Beijing Normal University

Transport layer Service

- The transport layer protocols provides logical communication between two application processes (not hosts):
  - Provides reliable connection-oriented services
  - Provides unreliable connectionless services
  - Provides parameters for specifying QoS
- depending on the services offered by the network layer, the added functionality in the transport layer can be big or small.

Transport layer vs. Network Layer



The network, transport, and application layers.

- the network layer is in the hands of carriers
  - Clients have no say in what the carrier actually offers.
- To develop applications that are independent of the particular service offered by a carrier, a standard communication interface is needed (such as socket interface)
  - The transport layer implements such interface

Transport layer interface

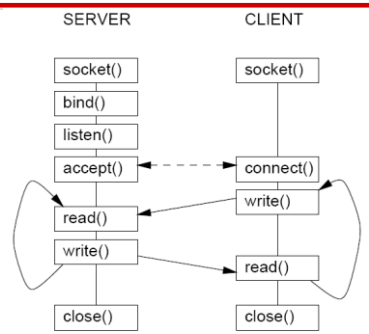
- Example: consider the Berkeley socket interface, which has been adopted by all Unix systems, as well as windows:

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Socket communication

- The client and server each bind a transport-level address and a name to the locally created socket.
- The server must listen to its socket, thereby telling the kernel that it will subsequently wait for connections from clients.
- After that, the server can accept or select connections from clients.
- The client connects to the socket.
  - It needs to provide the transport-level address by which it can locate the server.
- After a connections has been accepted (or selected), the client and server communicate through read/write operations on their respective sockets.
- communication ends when a connection is closed.

Connection-Oriented Socket communication



- Question: what about connectionless communication? There is no connection -> no need for listen, accept, and connect.

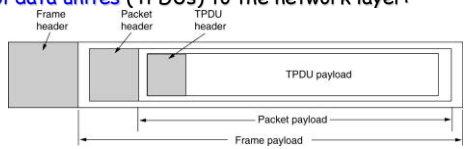
Elements of Transport Protocols

- Addressing
- Connection Establishment
- Connection Release
- Flow Control and Buffering
- Multiplexing
- Crash Recovery

use

Some observations

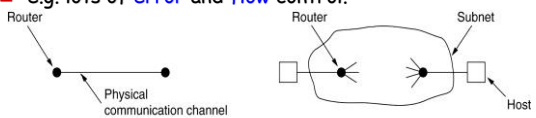
- Note 1: messages sent by clients are encapsulated as transport protocol data unites (TPDUs) to the network layer:



- Note 2: A real hard part is establishing and releasing connections. The model can be either symmetric or asymmetric:
  - Symmetric: one side sends a disconnect request, and waits for the other to acknowledge that the connection is closed.
    - Yes, there are some problems with this model. In fact, it turns out it is impossible to implement. //需要双方都执行disconnect
  - Asymmetric: one side just closes the connection, and that's it.
    - Yes, it's simple, but you may lose some data this way.
    - Not really acceptable. //任一方发起

Transport Protocols

- Note 3: Transport protocols strongly resemble those in the data link layer:
  - e.g. lots of error and flow control.



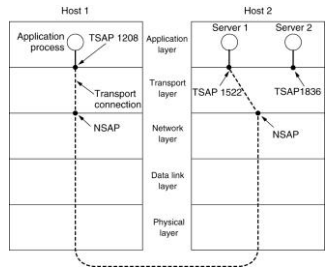
(a) Environment of the data link layer. (b) Environment of the transport layer.

(a) (b)

- Big differences when it comes to solutions!
  - Explicit addressing
  - Establishing, maintaining, and releasing connections
  - The many connections require different solutions
  - Handle effects of subnet storage capabilities

Addressing(编址)

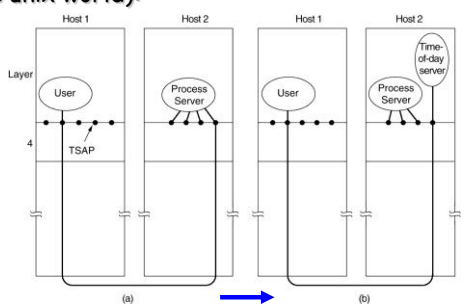
- Note: each layer has its own way of dealing with addresses. In TCP, a transport service access point is an IP address with a port number.



- Question: how do we get to know where the other party is?

Service Locations --- Fixed Addresses

- General solution: have a single process, located at a well-know address, handle a number of services (inetd in the unix world):



Service Locations ---Unknown Addresses

- Problem: sometimes you just can't have a process handle all services,
  - e.g. because the service requires special hardware (file server) , and cannot just be created on-the-fly when someone wants to talk to it.
  - → find address of the server
- Solution: you'll have to use a name server.

  Great, so how do we find the name server?

  Fixed address.
  - Examples: DNS(network level), name server(transport level),Skype mapping nickname to IP address(application level)

Connection Establishment

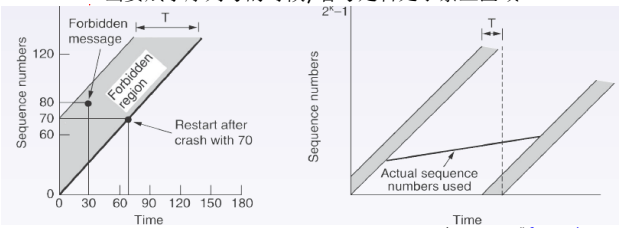
- Basic idea: 为建立连接，向对方发送一个 connection request. 然后对方接受连接请求并返回一个ack.
- Problem: suppose you don't get an answer, so you do another request.
  - first request消失 →no harm done
  - The ack 消失 →this can probably be detected.
  - Your first request 还在前往对方的路上
    - **接收方可能收到两个连接请求，迷茫！！！**
    - **Why?** The network has storage capabilities, and unpredictable delays. This means that things can pop up out of the blue. (延迟的重复分组突然冒出来)

抗重(1/2)

- Solution: 限制TPDUs的生存期 --- 如果TPDU最大的生存时间提前知道，那么能确保：前面的包被丢弃从而不会对后面的包造成影响。
  - 3种可能方案: 1)受限的子网设计, 2)每个包设置hop counter, and 3) each packet加上时间戳 (要求全局时钟同步)
- Basic idea: lifetime + sequence numbers (序列号要防回转)
- New problem: 主机崩溃,重启，它的TPDU序列号从哪开始？
  - 你不能等最大的packet lifetime T然后又从头开始: in wide area systems, T may be too large to do that.
  - 问题的关键点是 you must avoid that an initial sequence number corresponds to a TPDU still floating around. →So, just find the right initial number.

抗重(2/2)

- Solution: sequence numbers 根据时钟信息赋予(假定时钟不会停)
  - 建立连接的时候, 时钟值的低k位，用作 initial 序列号
  - 不同主机时钟不必synchronized
  - 存在forbidden region问题:
    - 当要赋予序列号的时候, 看号是否处于禁止区域



- Attention: 序列号使用速度过慢于时钟滴答, we may enter the region "from the top(left)". 同样地，太快makes you enter the region "from the bottom".

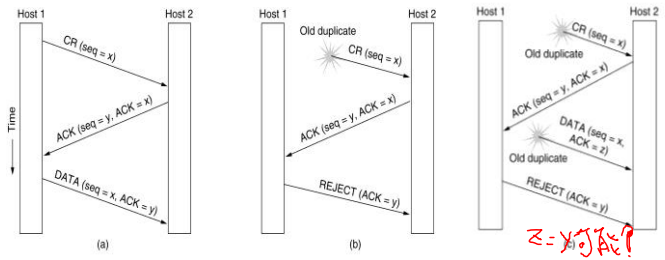
无错连接的建立(1/2)

- Problem: Great, we have a way of avoiding duplicates, but how do we get a connection in the first place?
  - host1给host2发送连接请求，包含建议初始序列号
  - Host2回送认可(connection accepted TPDU)
  - 若请求丢失，一个延迟重复的请求突然出现在host2（原因很多，如：重放攻击、传错等），则会建立一个不正确的连接(至少初始序列号不等)
  - 放弃"双方"初始序列号相同的要求!

//第五版：被连接方不记忆过往连接，所以连接请求是新或旧无法识别出来。只好问发送方，不认可就不真正建立连接,即是要：三次握手。见 fig.b.

无错连接建立 (2/2)

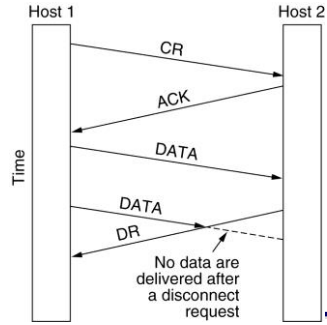
- Solution: Three-way handshake.



(a) Normal operation,
(b) Old CONNECTION REQUEST appearing out of nowhere.
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

无错连接释放?

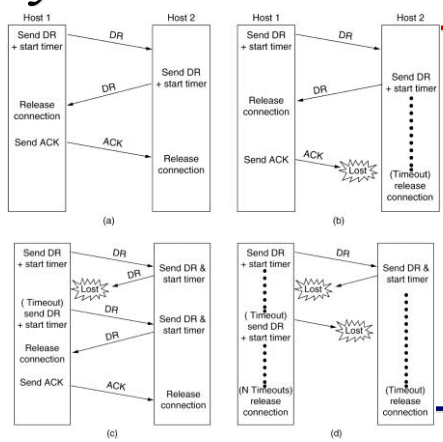
- Asymmetric release: one party just closes down the connection. May result in loss of data:



对称连接释放 (1/2)

- Big problem: 我们是否能设计一个释放连接的方案使得双方总能达成一致？ The answer is simple: NO.
  - Normal case: host 1 sends disconnect request. Host 2 responds with a DR. host 1 acknowledges, and ACK arrives at host 2.
  - ACK is lost: what should host 2 do? It doesn't know for sure that its DR came through.
  - Host 2's DR is lost: what should host 1 do? Of course, send another DR, but this brings us back to the normal case. This still means that the ACK sent by host 1 may still get lost.
- Pragmatic solution: use timeout mechanisms. This will catch most cases, but it is never a fool-proof solution: the initial DR and all retransmissions may still be lost, resulting in a half-open connection.

Symmetric connection release (2/2)



(e) half-open connection

Flow control and buffering

- Main problem: 主机可能维护有很多连接，为每个连接分配固定大小的缓冲区不可行 to implement a proper sliding window protocol → we need a dynamic buffer allocation scheme.
  - 在一个不可靠的网络场景中，网络层提供不可靠的数据报传输服务，the sender will have to buffer TPDUs until they are acknowledged.
  - The receiver may decide to drop incoming TPDUs if it has no buffer space available.
  - 在一个可靠的网络场景中，the sender will still have to buffer a TPDU until it is acknowledged: The network only acknowledges successful receipt (packet), not delivery (TPDU!).
- Solution: the sender and receiver need to negotiate the number of TPDUs that can be transmitted in sequence, only because buffer space no longer comes for free.

Buffer 预留

- Basic idea: 发送方请求接收方预留一定buffer，接收方做出决断。此后，接收方根据buffer可用情况做调整。
- Question: what can we do about the potential deadlock?
  - Just use yet another timer mechanism

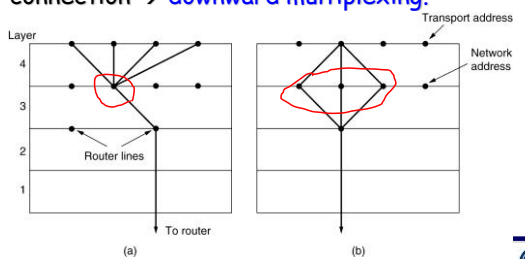
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	•••	<ack = 6, buf = 4>	←	Potential deadlock

Flow control – the network

- Problem: 考虑了收发双方根据缓存可用情况等因素传输速率的调整问题，网络的容量也是需要考虑的因素：网络可能不能满足收发双方的传输需要！
- Issue: 若网络每秒能传输c TPDUs for a stream，花费r seconds to transmit, propagate, queue and process the TPDU, and to send an ACK，那么发送者仅需 c.r buffers. More buffers is overkill of the network.
- Solution: let the sender estimate c and r and adjust its own number of buffers.

Multiplexing

- Basic idea 1: 假定网络仅提供有限数量的虚电路，or 用户 doesn't want to pay so much，那么可以使用a single circuit for several connections → upward multiplexing.
- Basic idea 2: 若用户想有更多的带宽，单一网络虚电路又不能满足，那么可以 use several circuits for a single connection → downward multiplexing.



Crash recovery (1/2)

- Problem: 假定服务器收到一个 TPDU，然后按要求执行一个operation and returning an acknowledgment(结果).
  - 若服务器before, during, or after its response崩溃的话，发送主机咋办?
- Situation: 假定发送方知悉服务器刚从崩溃中恢复过来，Should the sender retransmit the TPDU it just sent, or not?
  - 发送方可能两种状态:
    - S0: sender had no outstanding (unacknowledged) TPDUs
    - S1: sender had one outstanding TPDU(未决tpdu)

Crash Recovery (2/2)

Strategy used by receiving host

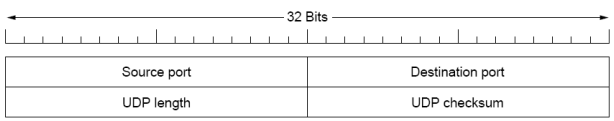
Strategy used by sending host	First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

不管如何正确地编写客户和服务器程序的代码(传输层)，总是存在使协议无法正确地恢复运行的情形，即主机的崩溃和恢复对于上面的层来说不可能是透明的。

User Datagram Protocol (UDP)

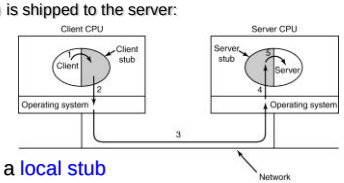
- Essence: the user datagram protocol is essentially just a transport-level version of IP.



- Observation:
  - UDP is simple: no flow control, no error control, no retransmissions
  - UDP packets cannot be larger than 64K
- Question: so why not use IP instead?
  - Because we still need the port fields to deliver the packet to the correct application

RPC

- UDP is used by the Remote Procedure Call (RPC)
  - a client-server communication in which a procedure is made available to remote client.
  - The call (including its parameters) is shipped to the server:

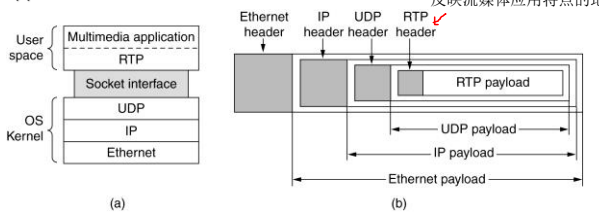


1.Client calls the procedure at a local stub
2.Client stub marshalls request: it puts everything into a (UDP) message
3.The message is transferred over the network
4.The server stub unmarshalls the message…
5. … and calls the local implementation of the procedure.

RTP(1/2)

- Problem: can we support multimedia streaming over the Internet? The real-time transport protocol provides some best-effort support.

反映流媒体应用特点的地方



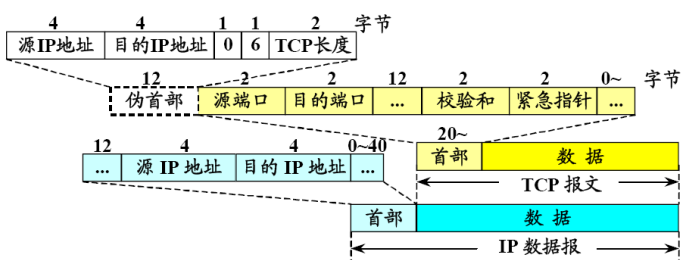
- Essence: RTP essentially just multiplexes a number of multimedia streams into a single UDP stream. The receiver is responsible for compensating missing packets (which is highly application dependent).

RTP(1/2)

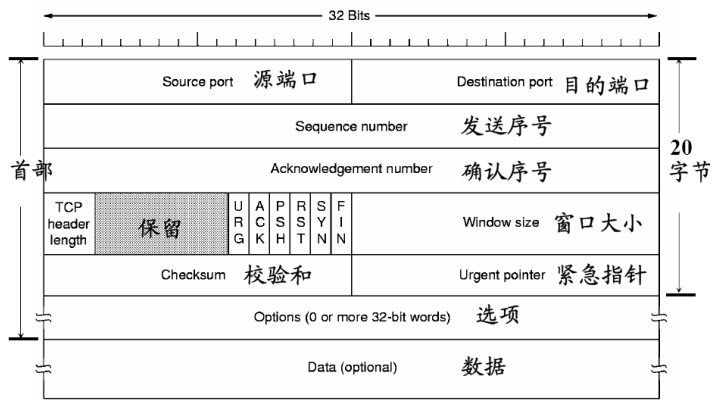
- Real-time: RTP packets can be timestamped:
  - packets belonging to the same substream can receive a timestamp indicating how far off they are with respect to their predecessor.
  - This approach allows the system to reduce jitter.
  - In addition, timestamps can be used to synchronize multiple substreams(e.g., video with two audio channels).
- RTCP (Realtime Transport Control Protocol)
  - P454:反馈\流同步\命名源
  - RTP=RTCP+RTP

传输控制协议－TCP

- 面向连接的服务;
- 全双工点对点通信; （E2E）
- 完全可靠（无丢失、无重复、无乱序）;
- 可靠的连接建立和连接释放。



TCP报文格式

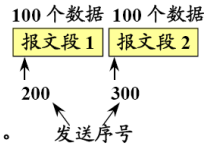


选项长度= $4 \times N$ 字节，N为整数

TCP的序号和确认

TCP是基于字节流的。对每一个字节编一个号。

在建立连接时，双方商定初始序号。



- 发送序号（4字节）
  - 报文段中的第一个数据字节的序号。
  - 可对4GB数据进行编号，可保证序号重复时，旧序号的数据已在网络中消失。
- 确认序号（4字节）
  - 期望下次收到的第一个数据字节的序号（当前报文段中的最后一个数据字节的序号+1）；
  - 并对收到的报文段表示确认。
  - 采用了捎带确认技术和累积确认技术。

- TCP首部的长度（4 bit）
  - 长度单位为32位字，包含可选项域；
- 6位的标志位，置1表示有效
  - URG：发送紧急数据。表明此报文不按排队顺序来传送，优先发送。它与紧急指针配合使用；
  - ACK：确认序号是否有效；
  - PSH：指示接收方将数据不做缓存，立即向上递交给应用进程进行处理立刻上送；
  - RST：由于不可恢复的错误重置连接；
  - SYN：用于连接建立指示；
  - FIN：用于连接释放指示。

- 窗口大小（2字节）
  - 接收端可以接收的字节数，发送端依据它调整发送窗口。
- 校验和（2字节）
  - 算法：所有16位字以补码形式相加，对其和求补。
  - 校验范围：TCP报文的首部、数据和伪首部。
- 伪首部
  - 仅参与校验和的计算。即不向下传送，也不向上递交。
  - 第4字段是IP数据报的首部里的协议字段值（协议类型：UDP=17、TCP=6）。

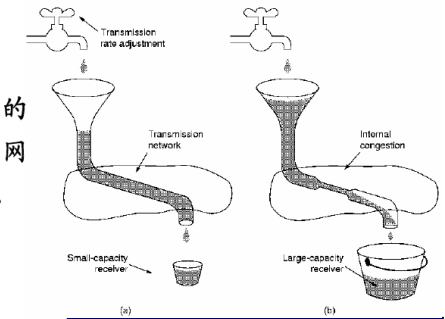
- 紧急指针（2字节）
  - 当前报文段的第一个数据到紧急数据结束位置的偏移量，
  - 即，最后一个紧急数据的序号=发送序号+紧急指针。
  - 当窗口大小=0时，也可发送紧急数据。
- 选项（长度可变，$4 \times N$字节，N为整数）
  - TCP只规定了一种：最大报文段长度MSS。
  - MSS告知对方：本方缓存所能接受的能力。
  - MSS的默认值=536（字节）。
  - 若MSS选择较小，网络利用率将降低。

发展起来的选项：窗口尺寸和重传（P460）

TCP拥塞控制

- 出现拥塞的两种情况
  - a.快网络小缓存接收者
  - b.慢网络大缓存接收者

导致网络拥塞的两个潜在因素是：网络能力和接收能力。



- TCP处理第一种拥塞的措施
  - 在连接建立时声明最大可接受段长度；
  - 利用可变滑动窗口协议防止出现拥塞；

称为流控也许更合适！

- TCP处理第二种拥塞的措施
  - 发送方维护两个窗口：可变**发送窗口**和**拥塞窗口**，按两个窗口的最小值发送；**即按有效窗口发送！**
  - 拥塞窗口依照**慢启动算法**变化。

- **慢启动算法**

1 连接建立时拥塞窗口初始值为该连接允许的最大段长，阈值为64K；

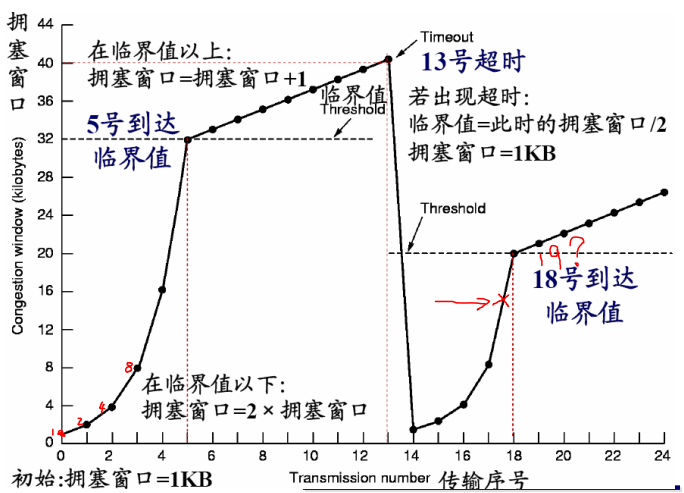
2 发出一个最大段长的TCP段，若正确确认，拥塞窗口变为两个最大段长；

3 发出（ 拥塞窗口/最大段长）个最大长度的TCP段，若都得到确认，则拥塞窗口加倍；

4 重复上一步，直至发生超时或拥塞窗口等于接收方声明的接收窗口大小；

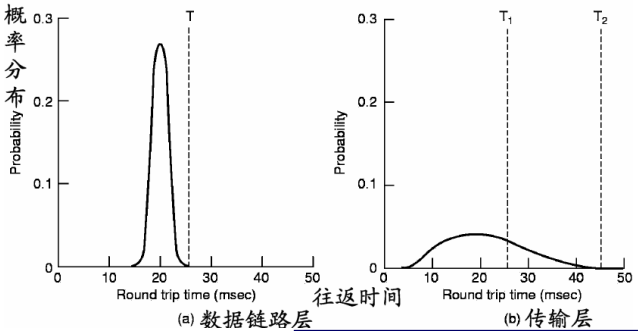
5 当超时发生时，阈值设置为当前拥塞窗口大小的一半，拥塞窗口重新设置为一个最大段长；

6 拥塞窗口按2、3步骤重新指数形增长，直至达到阈值，从此时开始，拥塞窗口线形增长，一次增加一个最大段长；直至超时或拥塞窗口等于接收方声明的接收窗口大小，发生超时时转5。



TCP的重传机制

- TCP每发送一个报文段，就启动一个重发定时器；
- 如果该报文段得到确认，就关闭这个重发定时器。
- 与数据链路层相比，往返时延的方差很大。



传输层的重传时间如何确定?

自适应算法

- 设置一个RTT变量－往返时间估计，
- 测量实际的往返时间：
  M=每一个报文发出时间－收到相应的确认报文时间
- 进行加权平均计算

$$RTT = \alpha RTT + (1-\alpha)M,\ \ \alpha \in (0,1)$$

问题：当前的RTT=50ms，后续的M分别为40、60ms，若α=0.9，新的RTT=?

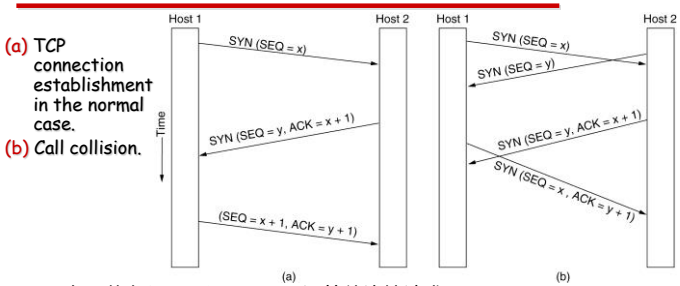
- 实际设置重传时间=βRTT，β>1

  TCP标准推荐 β=2。 $\longrightarrow$ $RTT = \alpha RTT + (1-\alpha)M$, $D = \alpha D + (1-\alpha)|RTT - M|$, $timeout = RTT + 4 \cdot D$

其他3个定时器p472 ！！！

Karn改进：重传数据段的M，不用于更新RTT(因为无法判断“确认”所属！),但超时间隔加倍，直至有数据段能一次通过为止

TCP连接的建立



- 服务器执行listen和accept原语等待连接请求；
- 客户执行connect原语，发生（a）三次握手。
- （b）只会建立起一个（x，y）连接
- 连接初始序列号，基于每4us一个时钟嘀嗒的方案（P462）

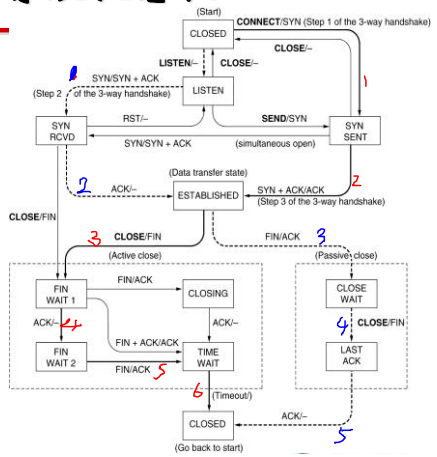
TCP连接的释放

1. 任何一方都可以发送一个设置了FIN位的TCP数据段，表明不再发送数据；
2. 当FIN数据段被确认，这个方向上就停止发送数据；
3. 另一个方向还可发送数据，直到要求释放连接和停止发送。

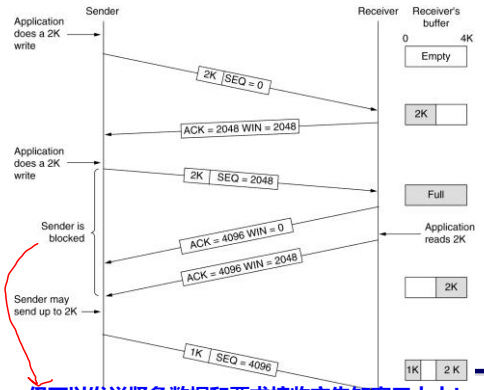
- 所以：释放一个TCP连接可能需要4个或3个TCP数据段。
- 当两倍最大分组生存期内FIN的应答没有到来，FIN的发送方直接释放连接。另一方最终也会注意到这一情况，从而因超时释放。

TCP连接管理有限状态机

The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.



TCP传输策略



- Basic idea: the receiver sends an acknowledgement for the next byte that can be sent in the current stream, and the maximum number of bytes that may be sent.

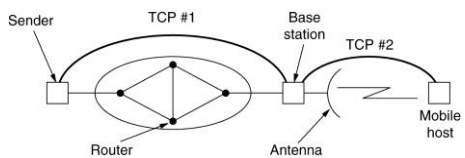
仍可以发送紧急数据和要求接收方告知窗口大小!

补充说明

- Important: the TCP entity is not obliged to immediately transmit data that the application hands over: it can do as much buffering as it likes. Same goes for acknowledgments.
- Example: interactive, character-oriented applications. Rather than sending one byte at a time, buffer as much characters as possible until the previous batch is acknowledged. Note: we're always stuck to at least 40 bytes of overhead per TPDU. //Nagle
- Example: avoid the silly window syndrome where the server is reading one byte at a time (and acknowledges one at a time). Instead, the receiver should wait until it can receive a reasonable amount of bytes in a row. //Clark

Wireless TCP

- Problem: TCP assumes that IP is running across wires. When packets are lost, TCP assumes this is caused by congestion and slows down. In wireless environments, packets get lost due reliability issues. In those cases, TCP should do the opposite: try harder.



- Solution 1: split TCP connections to distinguished wired/wireless IP:第一个连接上的超时，会导致发送方减速，第二个会加速。
- Solution 2: let the base station do at least some retransmissions, but without informing the source. Effectively, the base station makes an attempt to improve the reliability of IP as perceived by TCP.

事务型TCP*

- 解决RPC短消息问题
  - 请求和应答的消息非常小,以至于可以放到一个分组中.
  - 应答消息非常大,需多个分片,还要可靠传回.咋办?

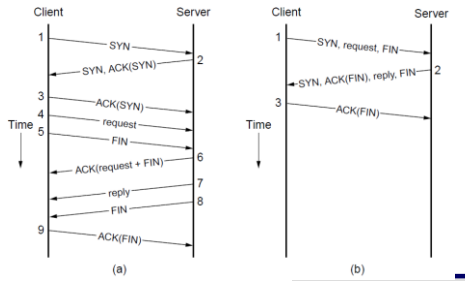


Fig. 6-40. (a) RPC using normal TCP. (b) RPC using T/TCP.

Summary

- 提供面向连接、可靠的双向E2E字节流传输服务;
- Transport address consists of a 16-bit port number.
- No support for multicasting or broadcasting.
- A TCP TPDU is called a segment, consisting of (minimal) 20-byte header, and maximum total length of 65535 bytes. A segment is fragmented by the network layer when it is larger than the network's maximum transfer unit (MTU), 主机具备分段重组的能力

北京师范大学 BEIJING NORMAL UNIVERSITY