传输层的需求

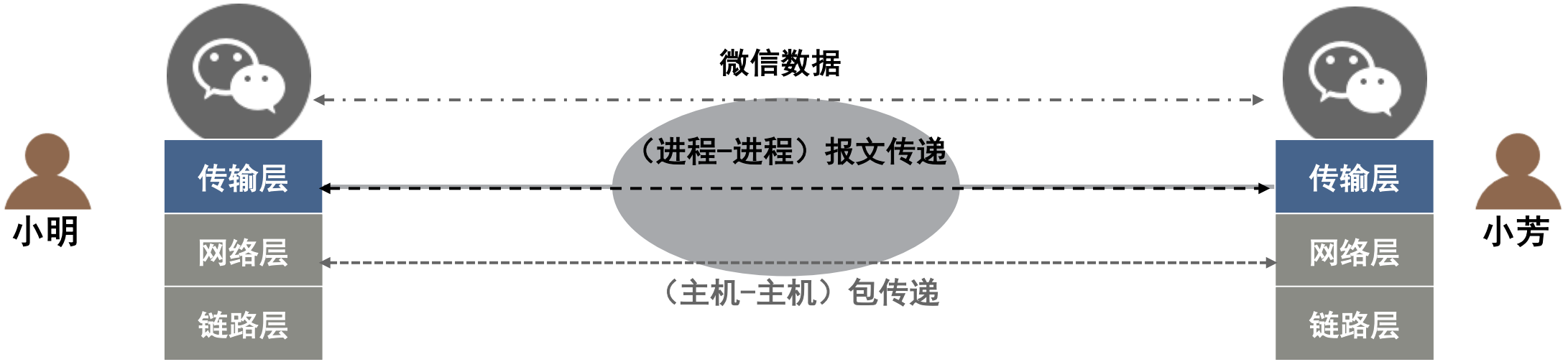
70/100/140 167/169/172 142/104/78 139/0/18 148/7/9 138/139/133 187/156/127 76/77/50 114/114/114

网络应用与网络性能

网络速度好慢哦，
与其这么干等着能
否干点别的呢~

- 应能同时支持多个网络应用
- 应能检验传输是否出错

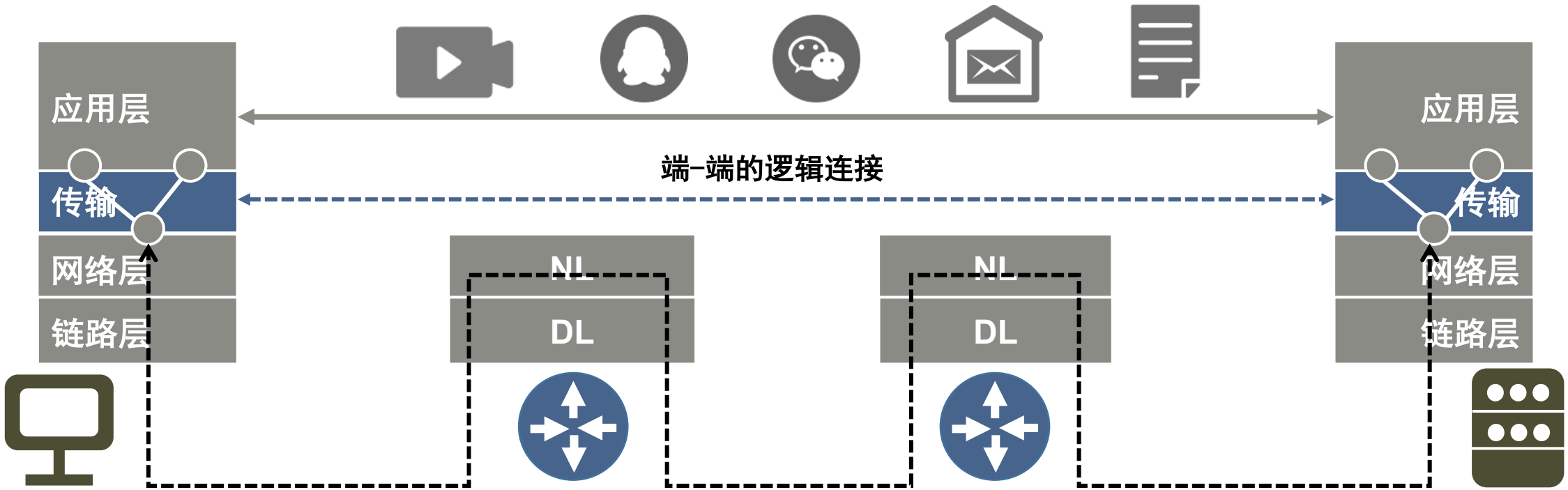
咦，你说的什么呀，
怎么还有个乱码，
我怎么看不明白呢？



注意：微信的真实实现涉及服务器，
这里只说明两者在通话

传输层概述

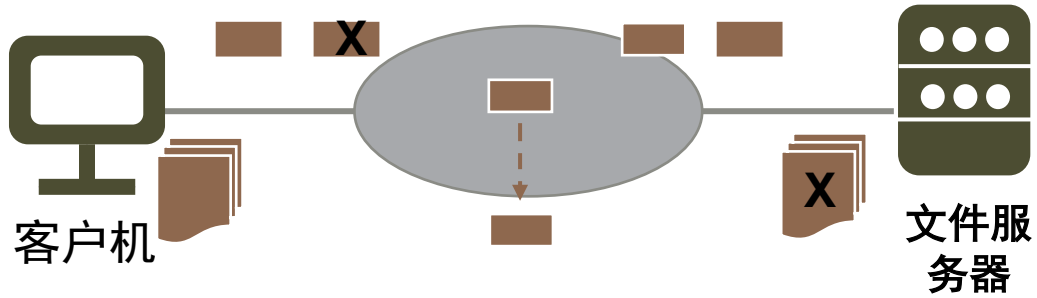
传输层协议能提供应用的多路复用/分用服务、可靠数据传送、带宽及延迟保证等服务质量保障。



传输层的作用

网络不可靠

- 网络层提供的服务不可靠(丢包、重复)
- 路由器可能崩溃
- 传输线路中断…



- 当数据传输过程中网络连接中断，可与远程传输实体建立一新的网络连接，在中断处继续数据的传输
- 传输层可检测到包丢失、损坏、乱序等差错情况，采取相应措施
- 传输服务原语独立于网络服务原语，因而应用程序可采用标准的传输原语

小明和小芳作为值日生的职责

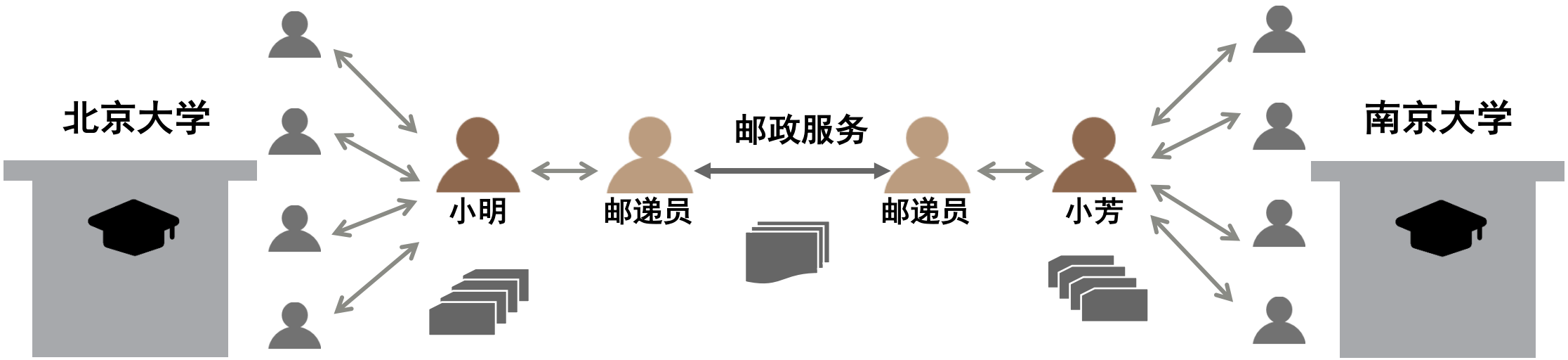
假设：小明和小芳所在班级组成共同兴趣小组
现在：

- 每周每位同学要给对方小组每位同学写一个报告，交流学习心得
- 报告通过邮政信件传递

试问：什么方式投递信件效率最高？

小明/小芳作为值日生：

- 承担着同学和邮递员之间的桥梁
- 挨个从同学手中收集信件
- 把信件一次交给邮递员
- 从邮递员手中接收信件
- 负责把信件分发到每个同学

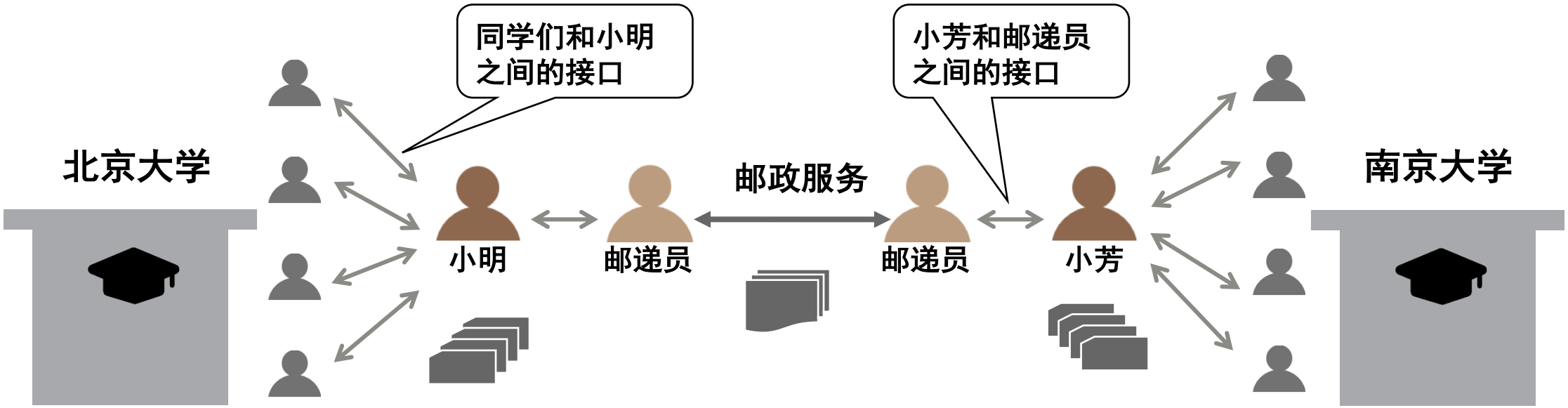


传输层与网络层的关系

同学们↔ 小明/小芳
vs
应用层↔ 传输层

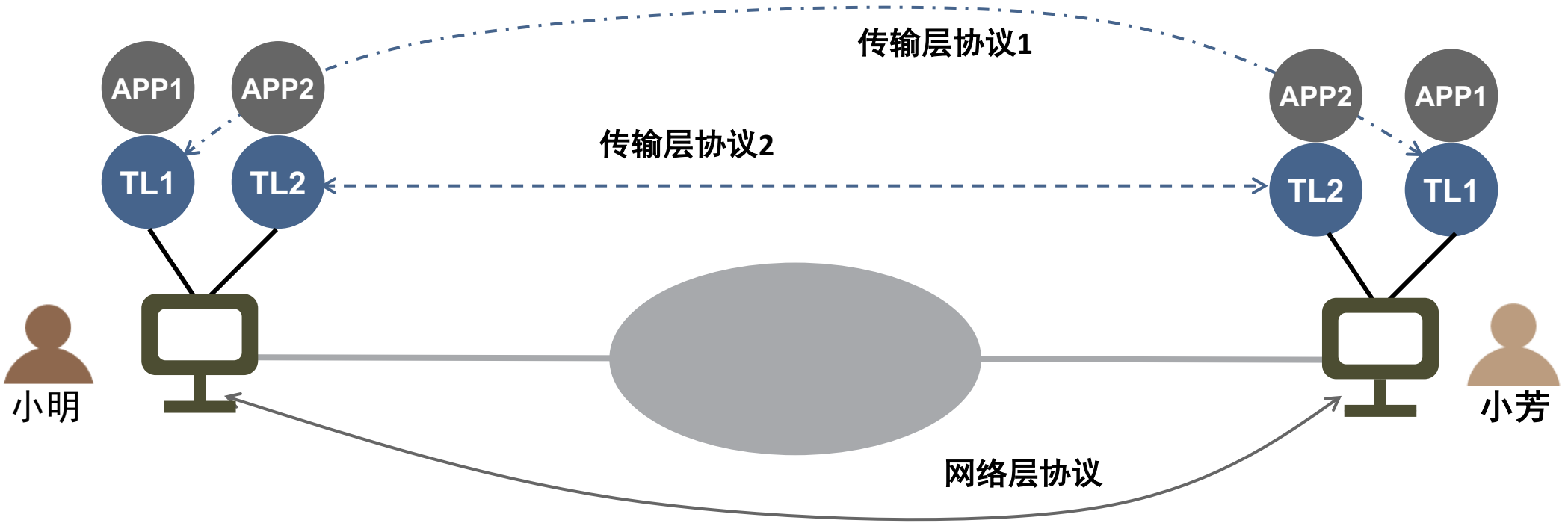
主机	北京大学、南京大学
进程	N个北大学生、N个南大学生
报文	信件
NL协议	传统邮政服务
TL协议	小明、小芳

小明/小芳↔ 邮差
vs
传输层↔ 网络层

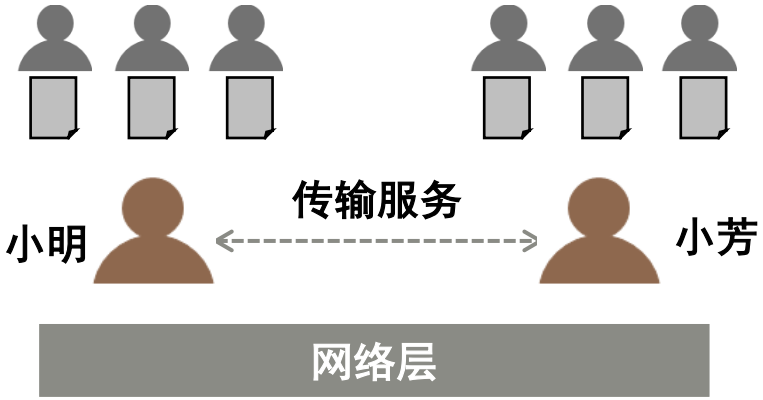


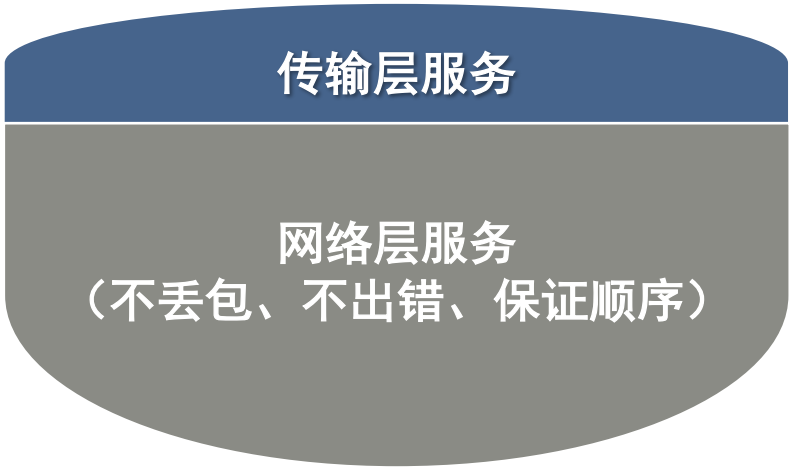
传输层和网络层的分工

- 网络层协议处理主机之间通信的事务
- 传输层协议处理应用进程之间通信的事务



传输协议面临的问题





基于可靠有序网络服务

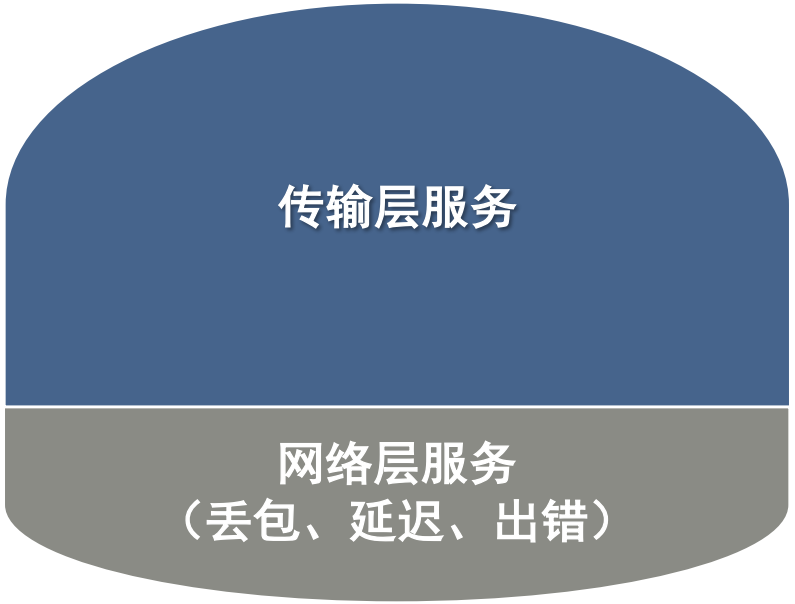
- 寻址（定位应用程序）
- 多路复用（为多个应用服务）
- 流量控制（发送接收匹配）
- 连接建立/释放

传输协议要解决的问题取决于网络底层所能提供的服务质量。

传输协议面临的问题（续）

基于不可靠的网络服务

- 寻址（定位应用程序）
- 多路复用（为多个应用服务）
- 流量控制（发送接收匹配）
- 连接建立/释放
- 有序传送（保证数据顺序）
- 重传策略（报文丢失出错后）
- 重复检测（必须丢弃重复报文）
- 系统崩溃恢复

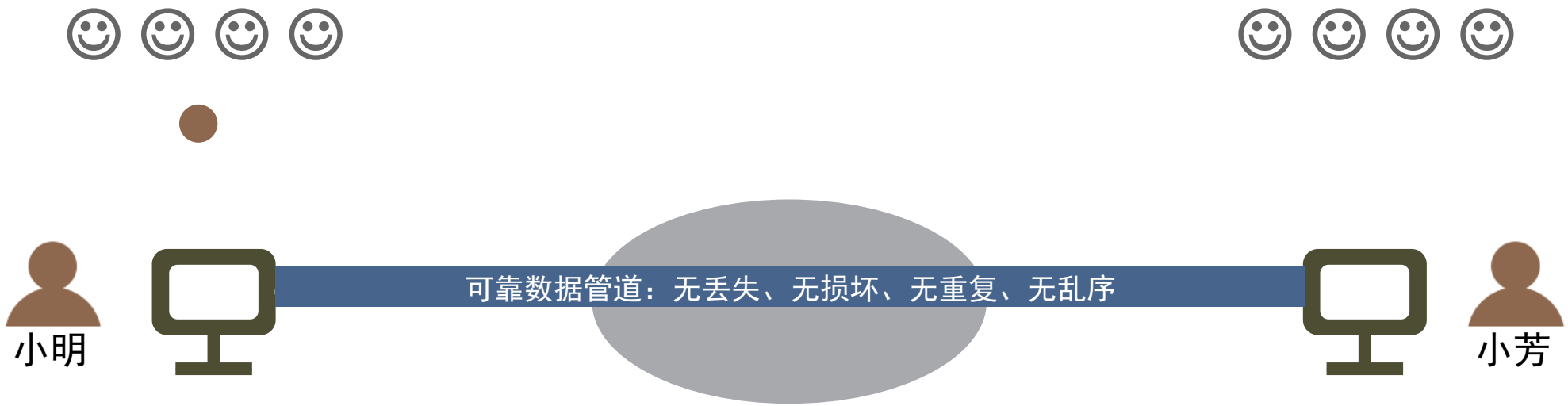


传输层的基本功能

面向连接的传输层服务

面向连接：通信双方要先建立一条逻辑链路才能开展数据传输业务。大多数面向连接的服务提供了可靠的通信。

- 小明和小芳先要协商建立一条逻辑链路用于信件传递
- 同学们交给小明/小芳发送的信件能到达目的地，并且保持发送顺序。



北京大学

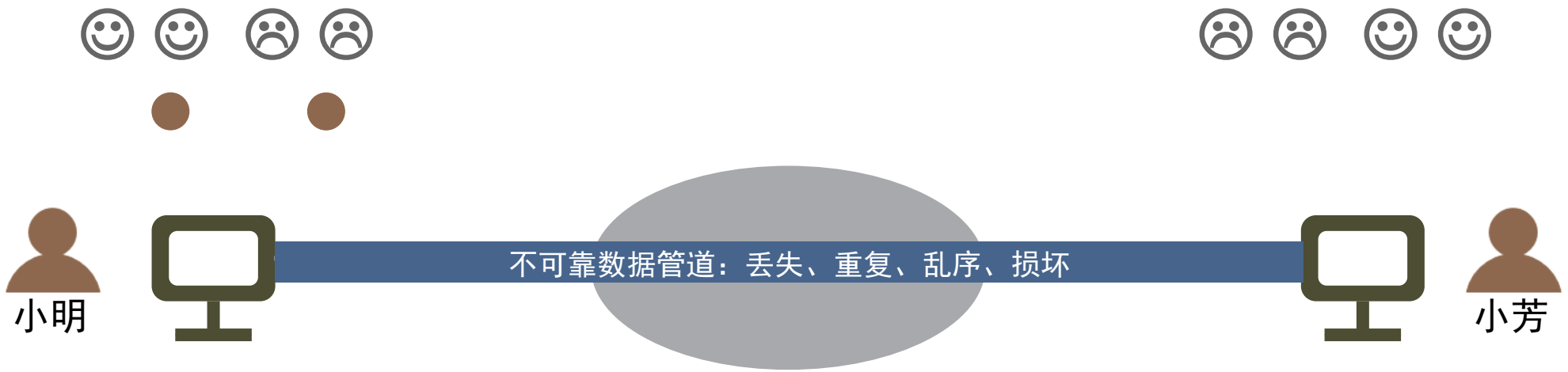
☠注意：连接与可靠性是两个不同的概念。

无连接的传输层服务

无连接：通信双方无需事先建立连接协商通信事宜就能直接发送报文。大多数无连接通信是不可靠的。

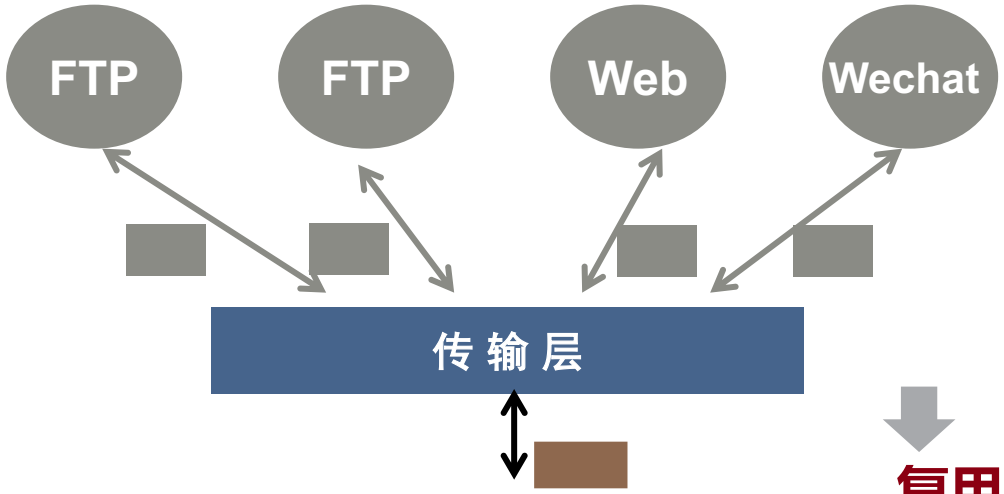
应用领域

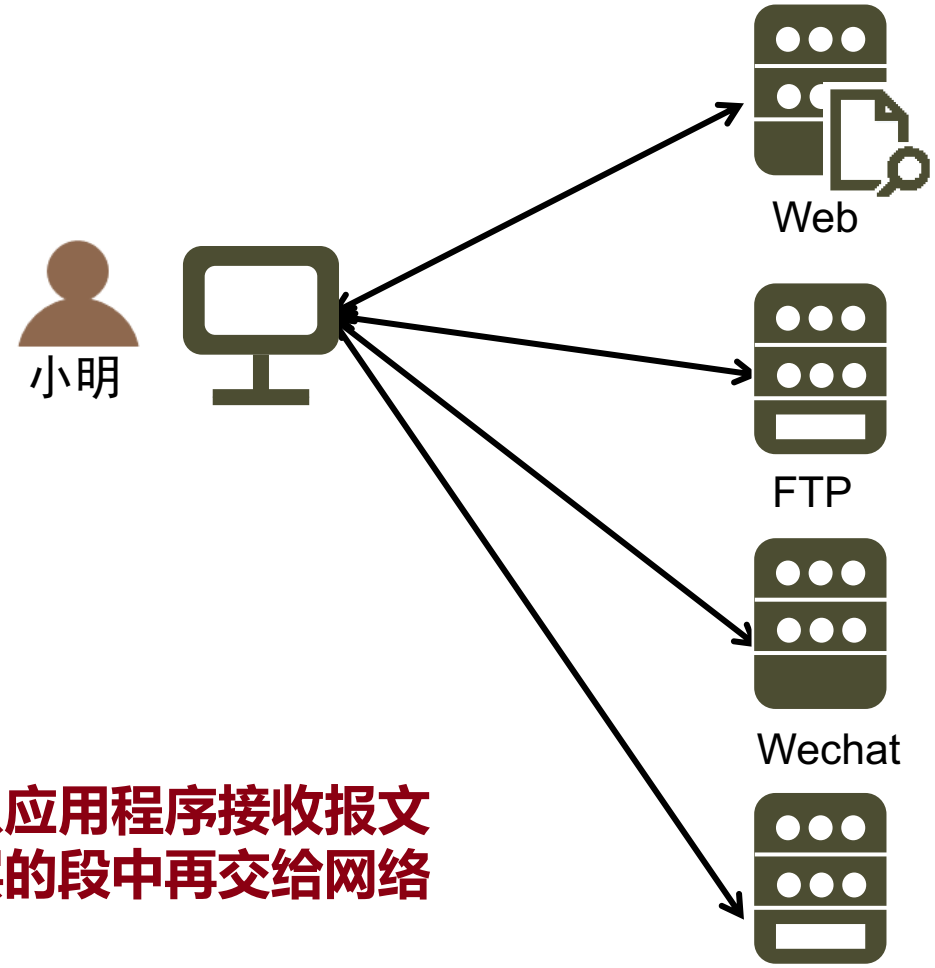
- 内部数据收集
- 外部数据发布
- 请求 – 响应
- 实时流媒体应用



传输层的多路复用与分用

分用：当传输层从网络层接收数据后，必须将数据正确递交给某个应用程序。



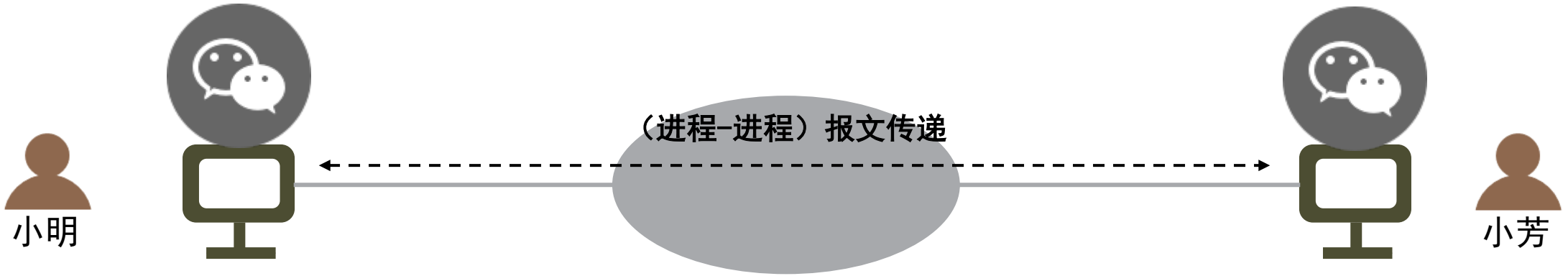


复用：当传输层从应用程序接收报文后要封装在传输层的段中再交给网络层发送。

面向连接与可靠性

面向连接服务	无连接服务
••建立连接 ••传输报文 ••拆除连接	••传输报文

可靠数据传输	不可靠数据传输
••报文不丢失 ••报文不重复 ••报文不损坏 ••报文不乱序	••报文可能丢失 ••报文可能重复 ••报文可能损坏 ••报文可能乱序



- 连接性是传输层向上层用户提供的服务的使用方式
- 可靠性是传输层向上层用户提供的服务的质量保障

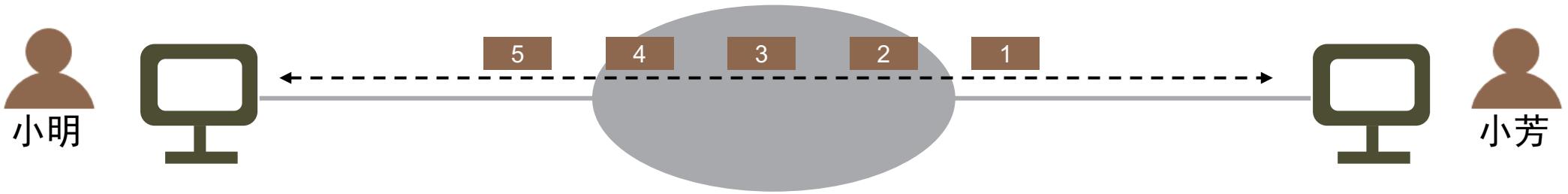
传输层的差错控制

差错控制指传输层具备检测所传报文是否出错并施行控制的能力。

- 给报文编号
- 设置超时计时器
- 序号检测和计时器超时

差错控制

- 报文丢失（超时机制）
- 报文重复（序号检测）
- 报文损坏（计算校验和）
- 报文乱序（顺序检测）



- **差错控制是数据传输可靠性保障的基础**
- **差错控制是传输传输服务质量保障的前提**

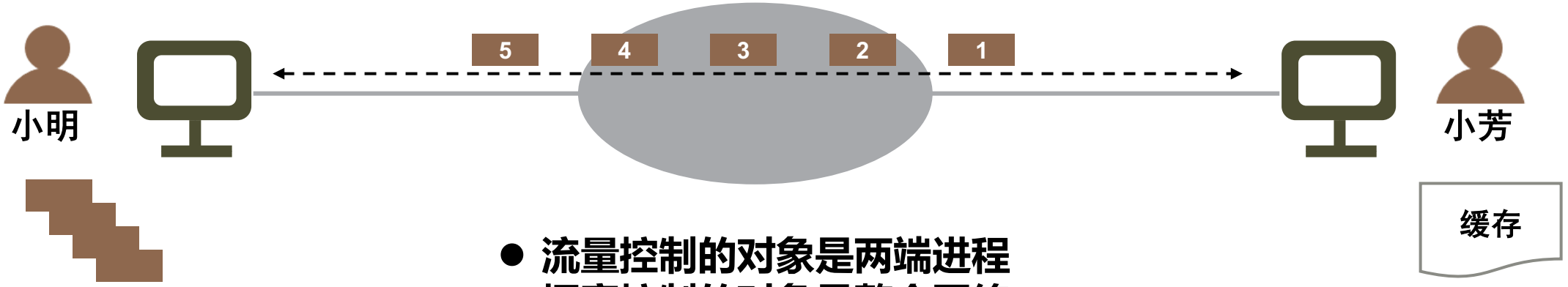
流量控制与拥塞控制

流量控制

发送方和接收方的计算、缓存和收发能力不一致都将造成报文发送和接收在速度上的差异

拥塞控制

网络中任何一个区域的报文转发不畅都有可能造成报文丢失、时延过长，任其发展后果严重



- 流量控制的对象是两端进程
- 拥塞控制的对象是整个网络

传输层服务模式

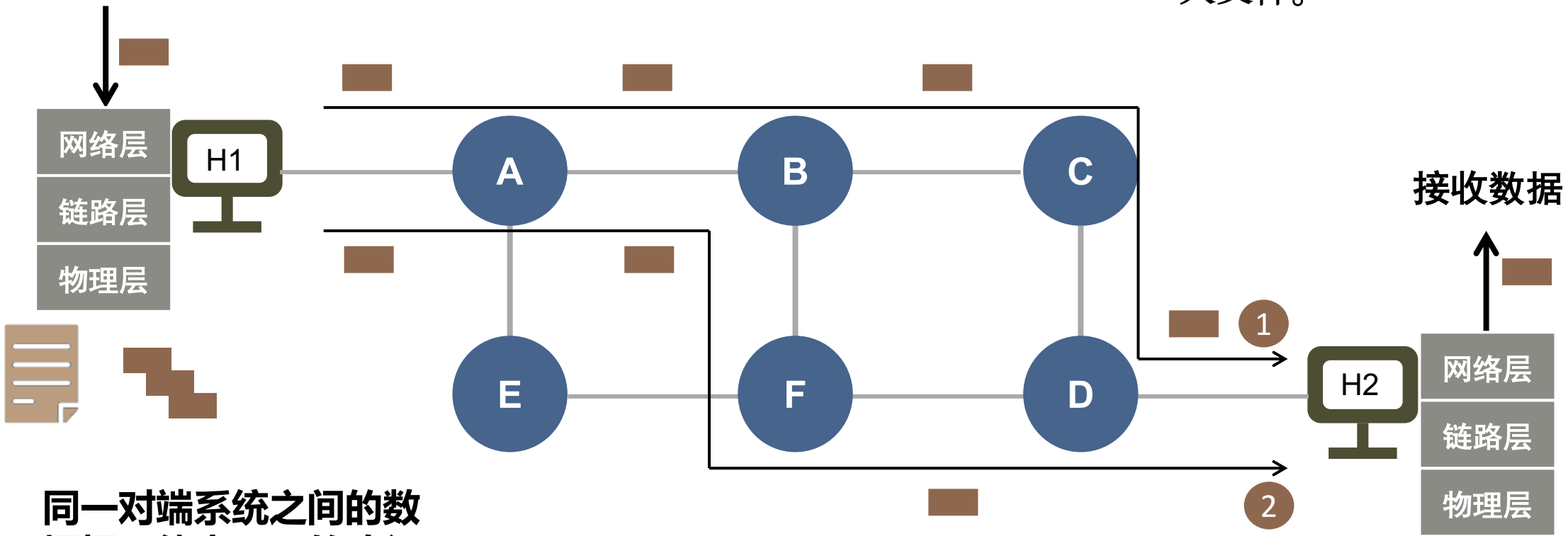
VS.

网络层存储转发

数据报子网——内部结构

发送数据

假设: H1给H2发送一个大文件。



同一对端系统之间的数据报可能走不同的路径。

数据报子网——实现

数据报子网的实现

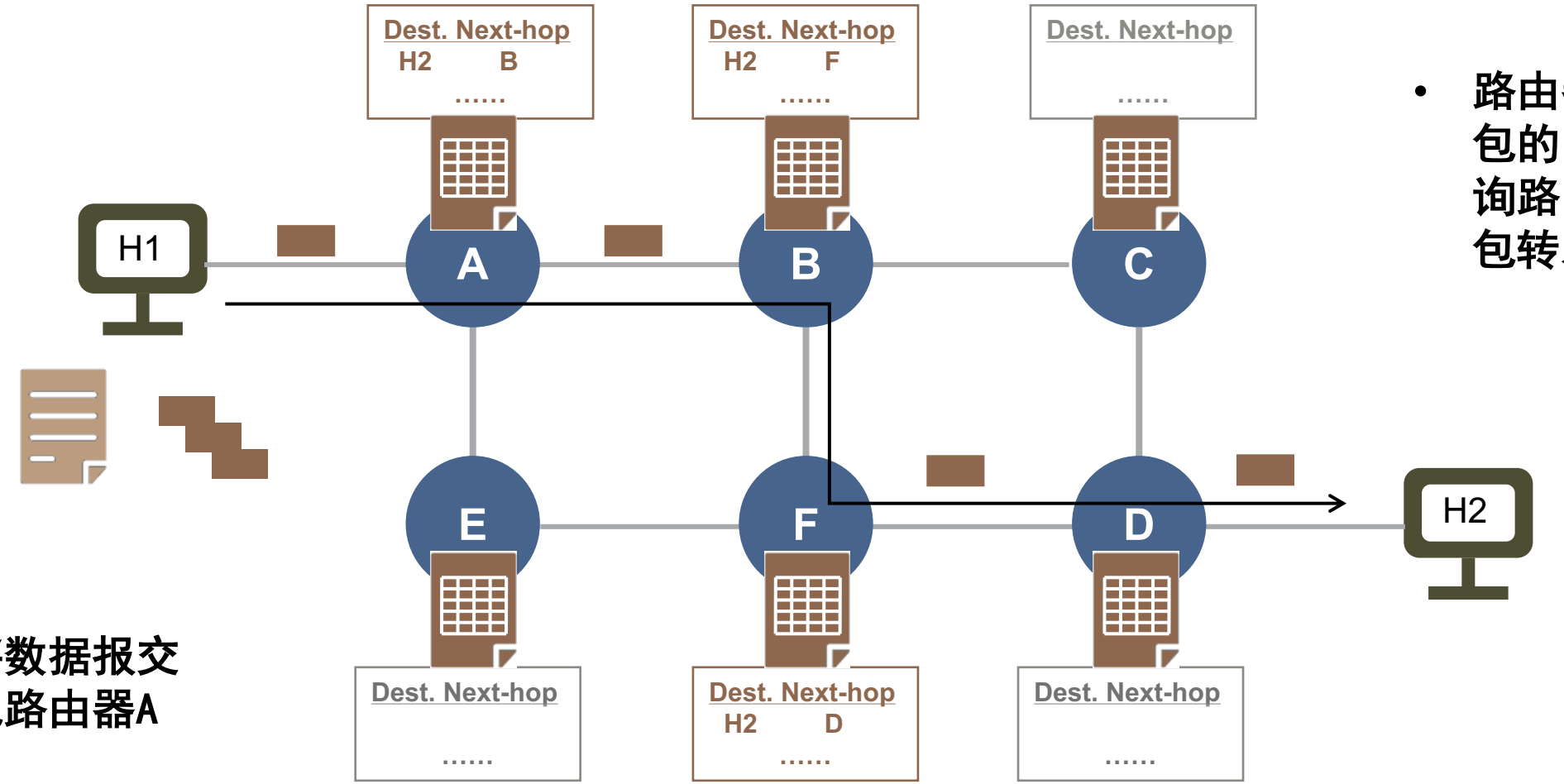
- 每个包必须包含目标端的完整地址(如IP地址)
- 路由器用一张表(路由表)指出通向目标端的出境线路
- 当一个包入境时，路由器查找路由表并将包沿出境线路发出，无须修改包中的任何内容。

路由表

目标地址	下一跳	度量
B	接口1	10
C	接口1	12
D	接口2	30

- 目标端地址指明从本地出发可达的目标地址（通常是网络号）
- 下一跳指去往目标地址的下一站（哪个邻居），与网络接口相连
- 路由度量标明了到目标地址的性能(距离，成本…)

数据报子网实现示例

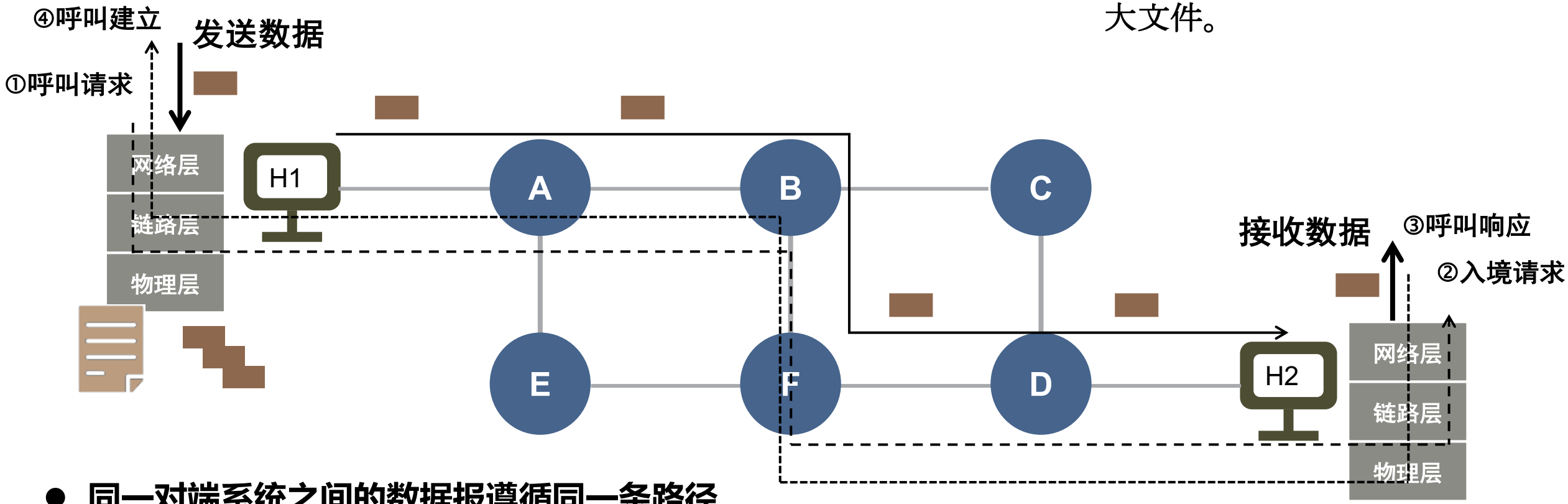


- 主机将数据报交给本地路由器A

- 路由器根据入境包的目标地址查询路由表确定将包转发到下一跳

虚电路子网——内部结构

假设: H1给H2发送一个大文件。



- 同一对端系统之间的数据报遵循同一条路径
- 路由器依据包头的虚电路号转发

虚电路子网——实现

虚电路转换表

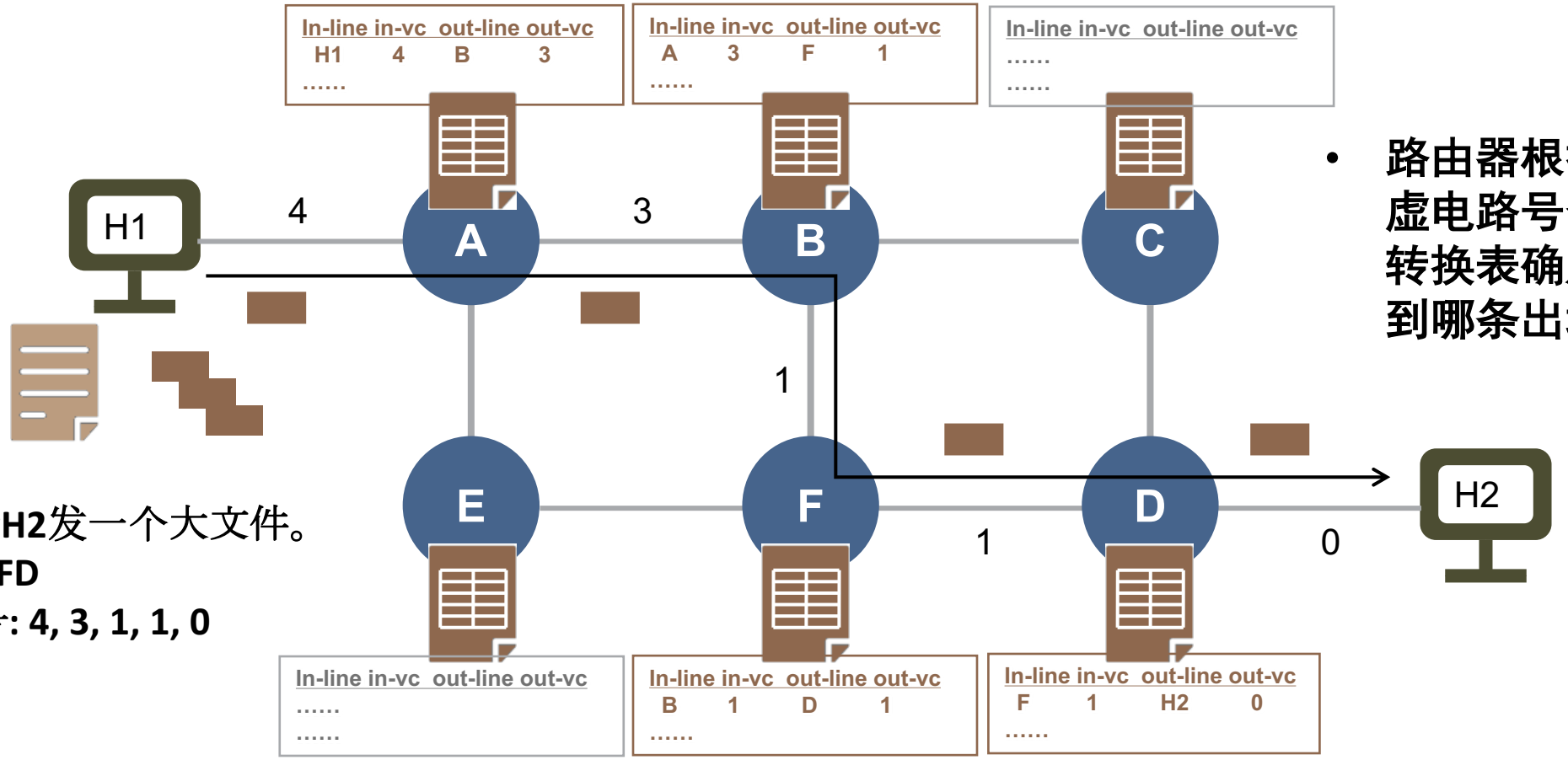
- 建立虚电路时选择一个当前未用的最低虚电路号
- 数据报报头包含一个虚电路号
- 转发数据报时要修改报头中原来的虚电路号
- 每当建立了一条新的虚电路时在表中添加一项
- 每当终止一条虚电路时在表中删去相应条目

虚电路转换表

入　　境		出　　境	
入境线路	虚电路号	出境线路	虚电路号
A	0	B	0
A	1	C	0

- 入境线路指接收包的链路，连接转发包到本地的邻居节点
- 出境线路指转发包的链路，连接包被转发到下一跳的邻居节点
- 虚电路号指复用同一条链路的电路编号

虚电路子网实现示例

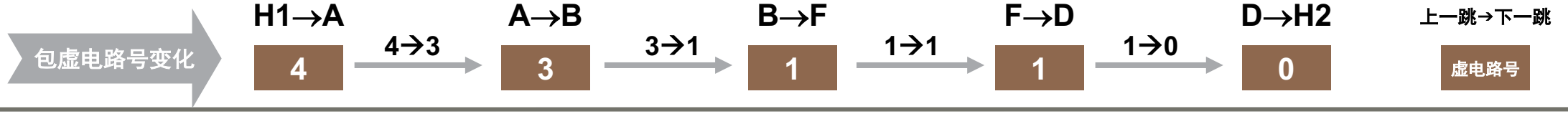


- 路由器根据入境包的虚电路号查询虚电路转换表确定将包转发到哪条出境线路

假设: H1要给H2发一个大文件。

路径: ABFD

虚电路号: 4, 3, 1, 1, 0



网络的外部操作/使用模式

有连接服务

- 源端执行一个目标端的呼叫请求
- 提交给网络的所有报文都标识为属于某个特定逻辑连接，并被顺序编号

无连接服务

- 不仅独立处理报文，且不保证按序或可靠地传送报文

通信子网

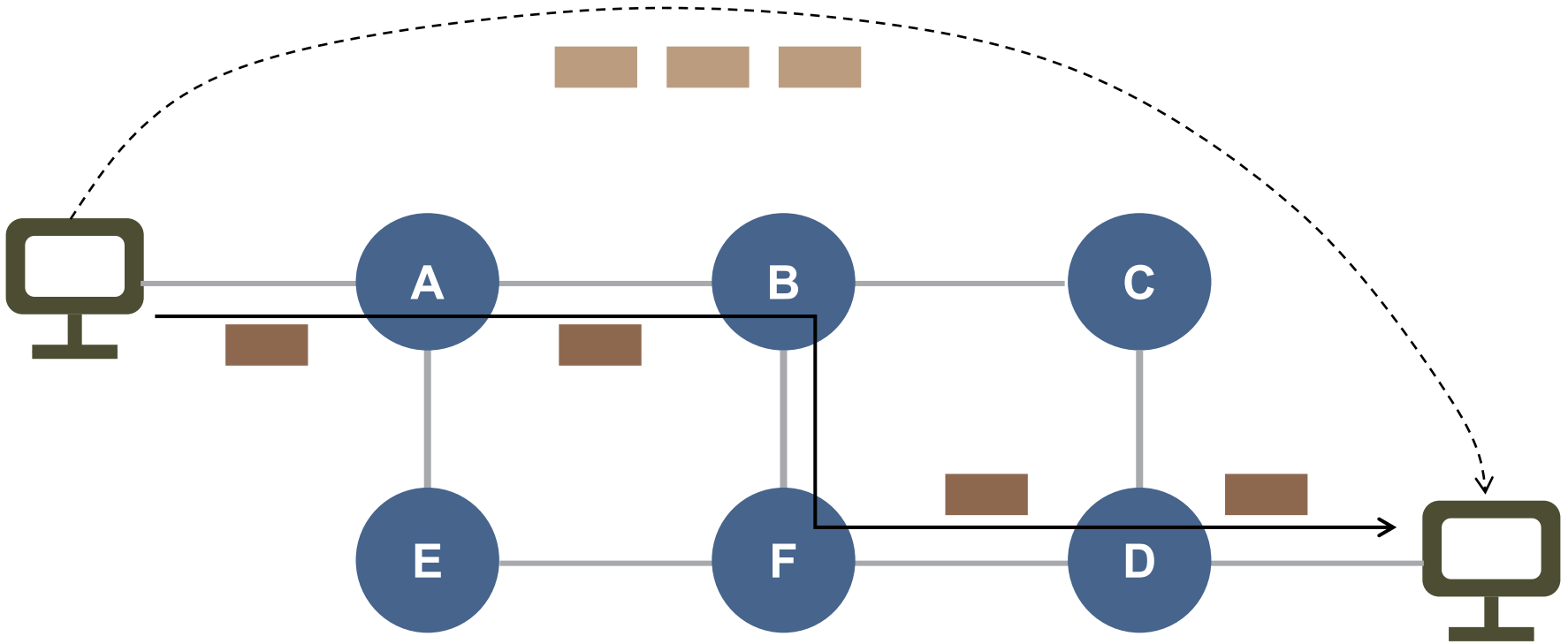
（数据报/虚电路）

- 用户通过服务接口选择特定的方式使用网络服务
- 网络内部什么结构用户不知道也无需知道

有连接服务vs.虚电路结构

●面向连接的服务用虚电路实现

AAL vs. ATM

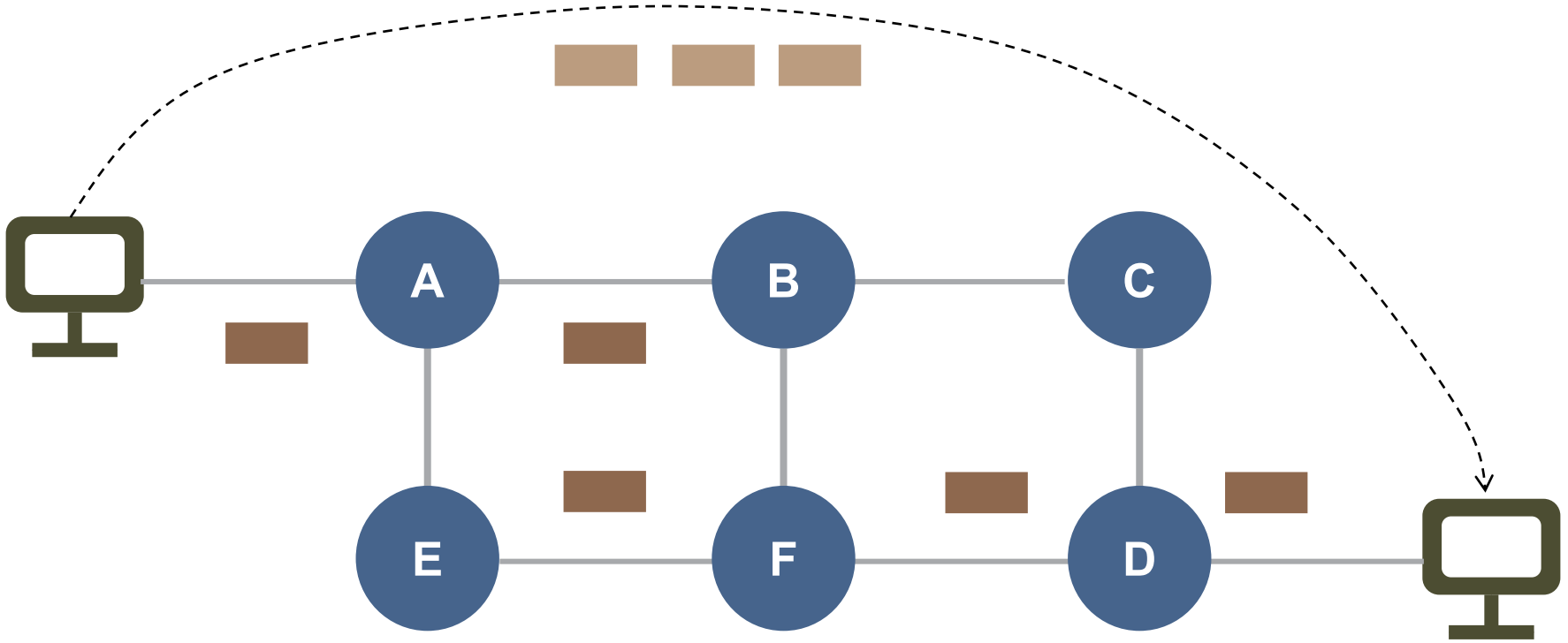


- 网络单独处理每个包，根据虚电路转换表转发包。
- 同一条外部连接上的报文走相同的路径（虚电路）。

有连接服务vs.数据报结构

- 面向连接的服务用数据报实现

TCP vs. IP

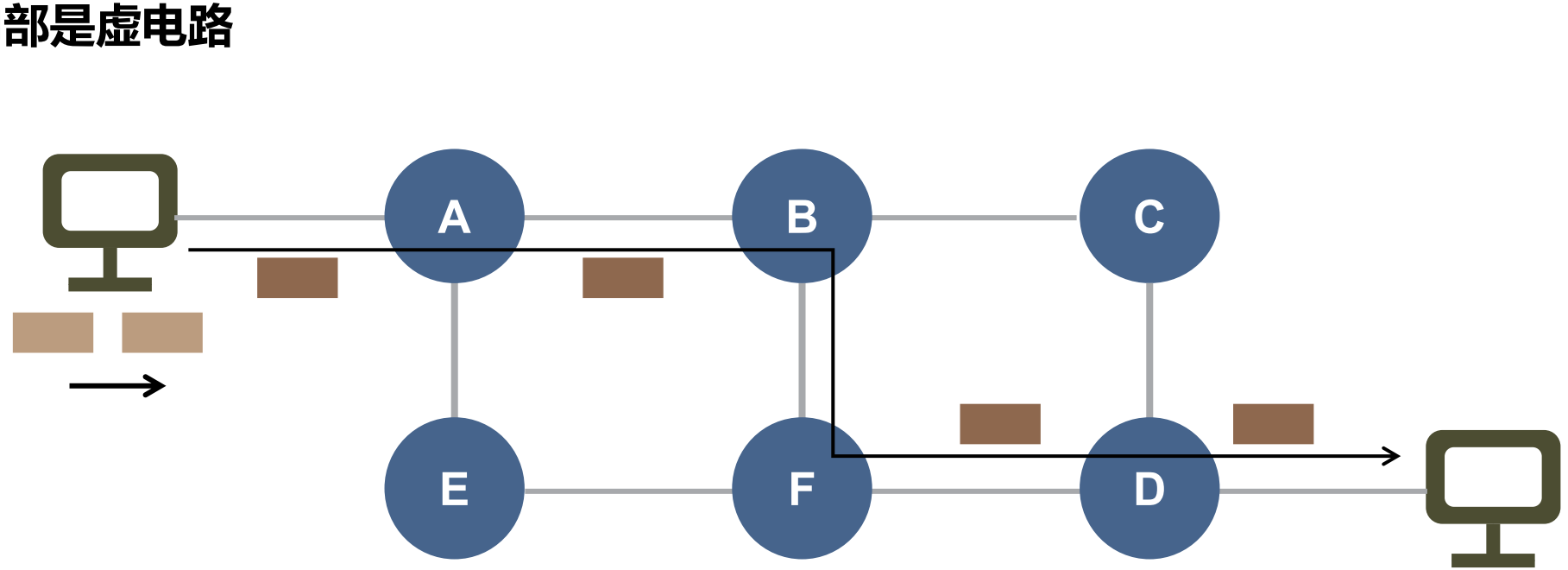


- 网络单独处理每个包。同一条外部连接上的报文可能走不同的路由
- 需要时在目标节点缓冲包，以便按正确次序递交给上层用户。

无连接服务vs.虚电路结构

- 用户看不到任何连接但网络内部是虚电路

IP vs. ATM

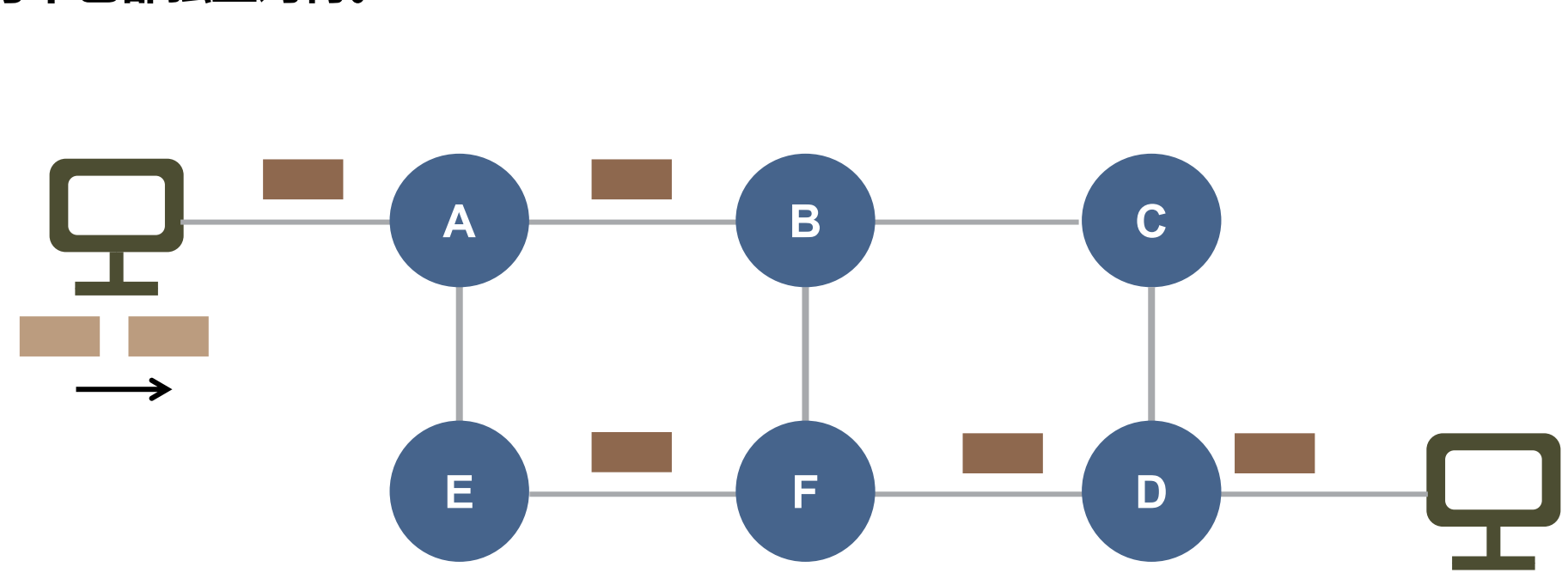


网络为传递报文在两点间建立一条虚电路，所有包都沿着这条虚电路传递到目标端。

无连接服务vs.数据报结构

- 无论从用户还是网络观点看，每个包都独立对待。

UDP vs. IP



用户传输报文不需要建立连接，网络传递包也不需要建立虚电路。

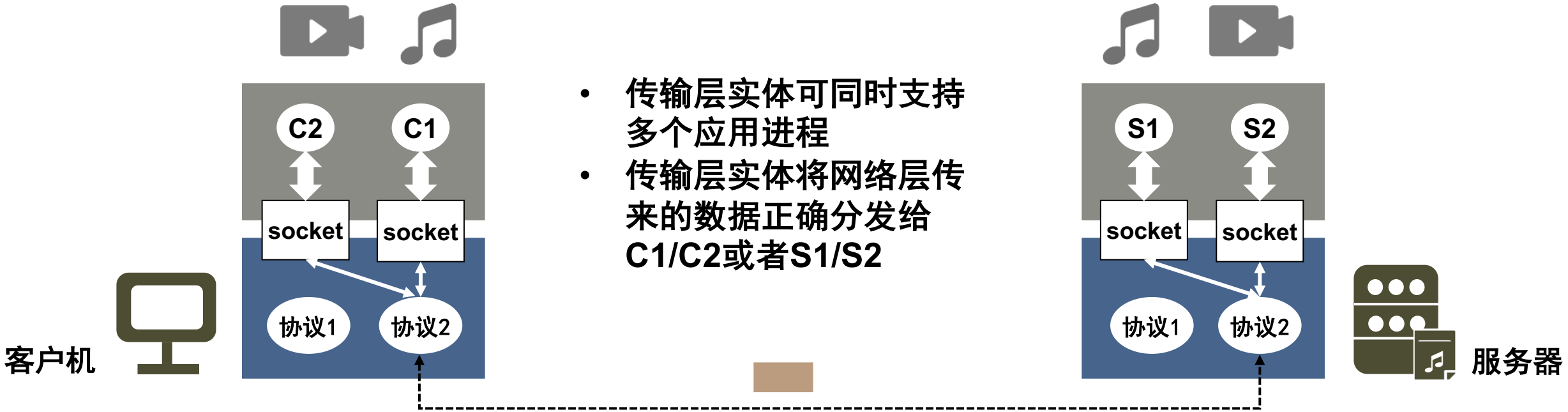
应用接口与传输层的多路复用

应用进程的编程接口

socket：应用进程进行端-端通信时访问传输层的软件接口

使用socket的通信过程

- 应用程序发送时将数据放入socket
- 应用程序接收时从socket提取数据
- 收发主机任意时刻可有多个socket



- 传输层实体可同时支持多个应用进程
- 传输层实体将网络层传来的数据正确分发给C1/C2或者S1/S2

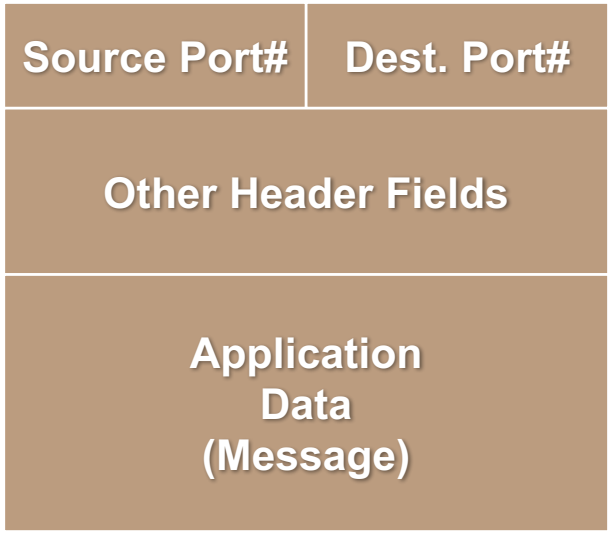
传输层的端口

端口号：用来标识某个特定的应用进程的ID。

端口的作用

- 应用层通过端口将数据交给传输层发送
- 传输层通过端口将数据交给应用层
- 端口号仅本地有效

传输层报文概念格式



- 端口号是socket的一部分
- socket唯一标识了一个应用进程
- 每个报文必须有字段描述传递数据的socket

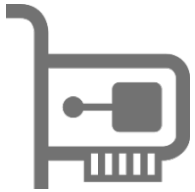
- 源端口号指明了发送报文的应用进程
- 目标端口指明了接收报文的应用进程
- 其他头字段包含传输层协议控制信息
- 应用数据就是传输层报文的有效载荷

传输层协议数据单元的封装

? 这些协议数据单元分别以什么地址传递？

端口号

协议地址



MAC地址

报文 | 报文头 | 有效载荷

包 | 包头 | 有效载荷

数据帧 | 数据帧头 | 有效载荷

无连接的多路复用和分用

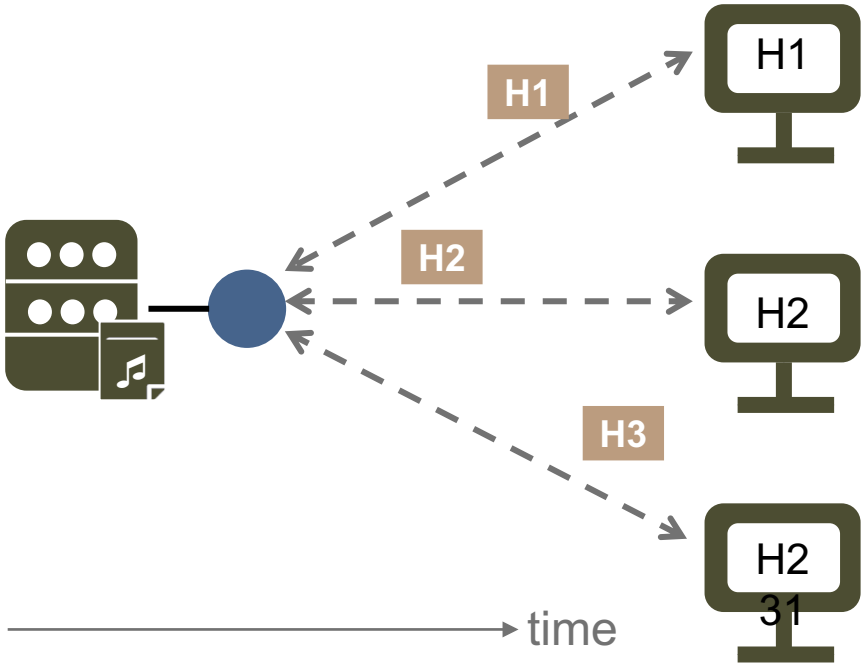
无连接socket由二元组标识
<Dest. IPaddr, Dest. Port#>

重复型客户机-服务器模型

重复型客户机/服务器模型：服务器在任何时刻只能为一个客户服务

服务特点

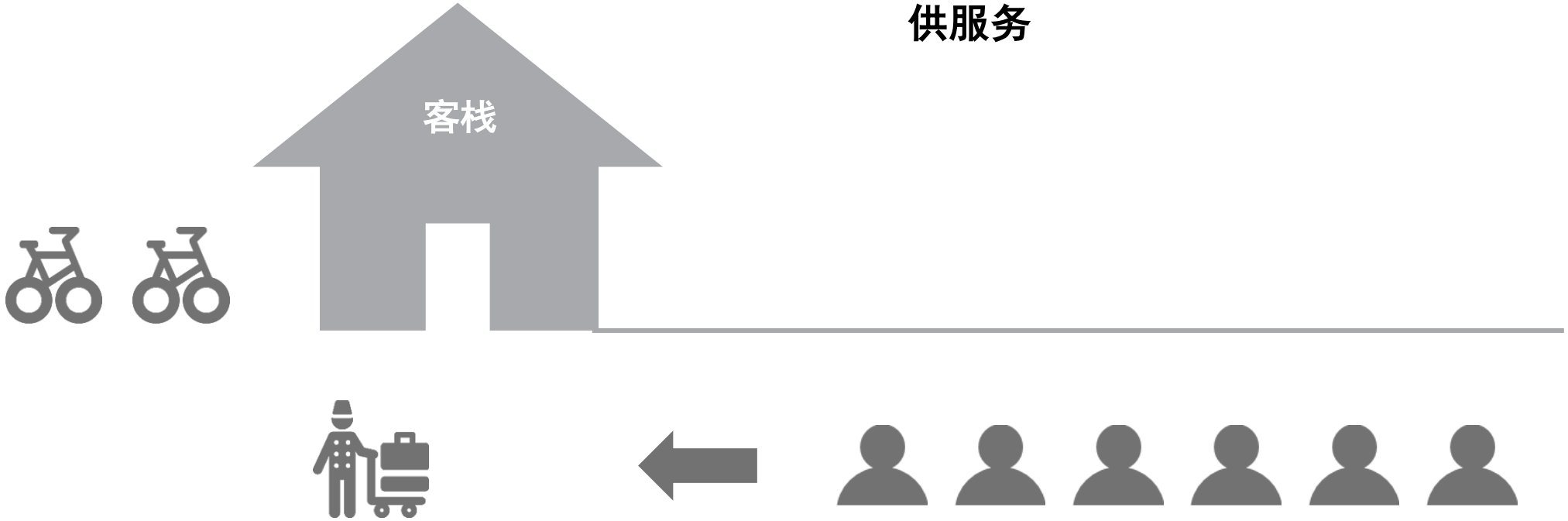
- 等待一个客户请求的到来
- 处理客户请求
- 发送响应给客户



- 所有客户的请求在一个队列中排队
- 服务器只需要一个端口服务所有客户

重复型客户机-服务器模型类比

- 只有一个服务员，依次为每一个客户服务
- 服务期间服务员不能为其他客户提供服务



面向连接的多路复用和分用

面向连接的socket由四元组

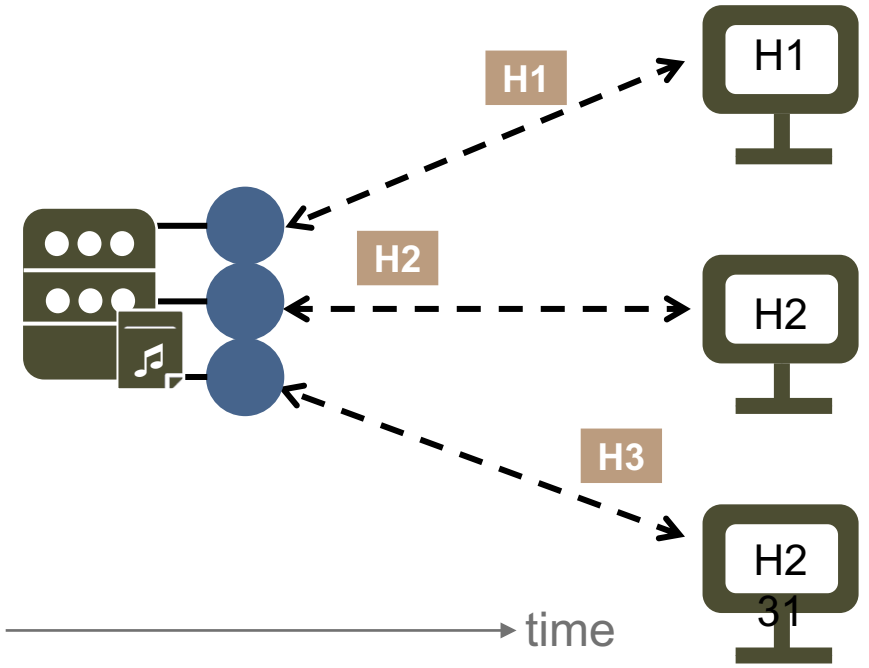
< Source Addr, Source Port#, Dest. Addr, Dest. Port# >

并发型客户-服务器模型

并发型客户机/服务器模型：服务器可以同时为多个客户机服务

服务特点

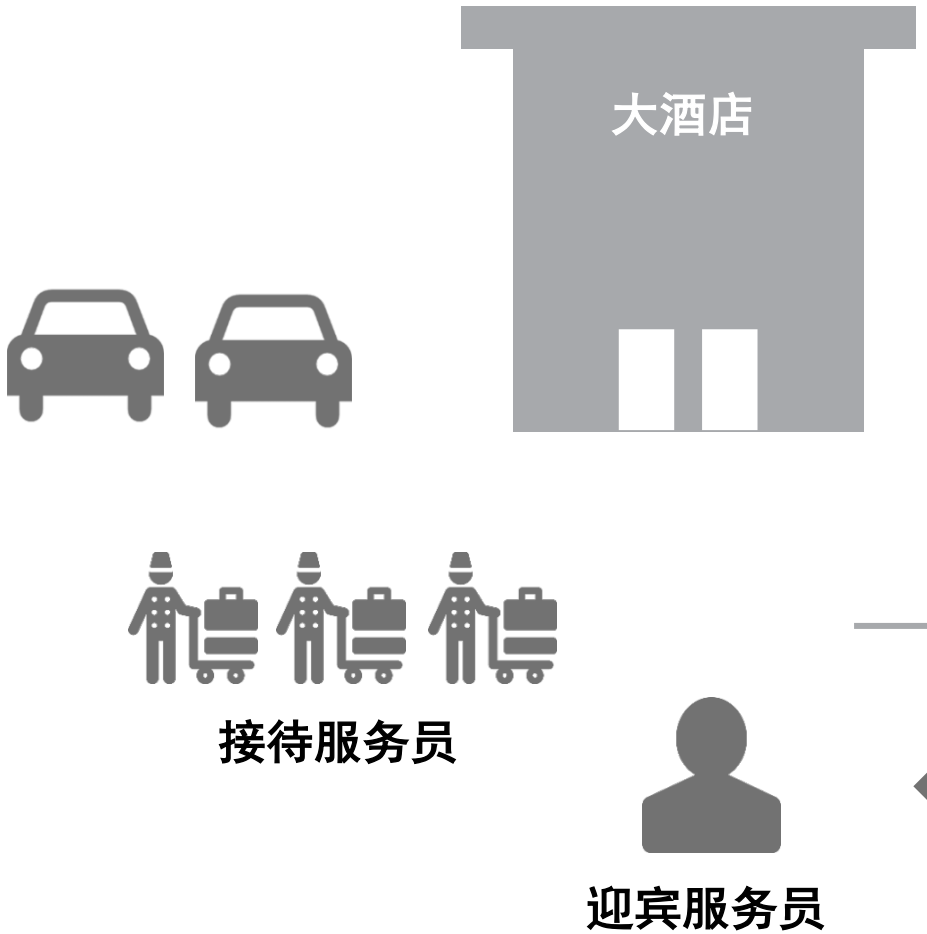
- 服务器等待客户机请求的到来
- 每到一个请求便启动一个新的服务器
  - ✓ 新服务器处理该客户的全部请求
  - ✓ 处理结束后终止该服务器



传输层根据socket的四元组将来自网络层的数据报分用到相应的socket

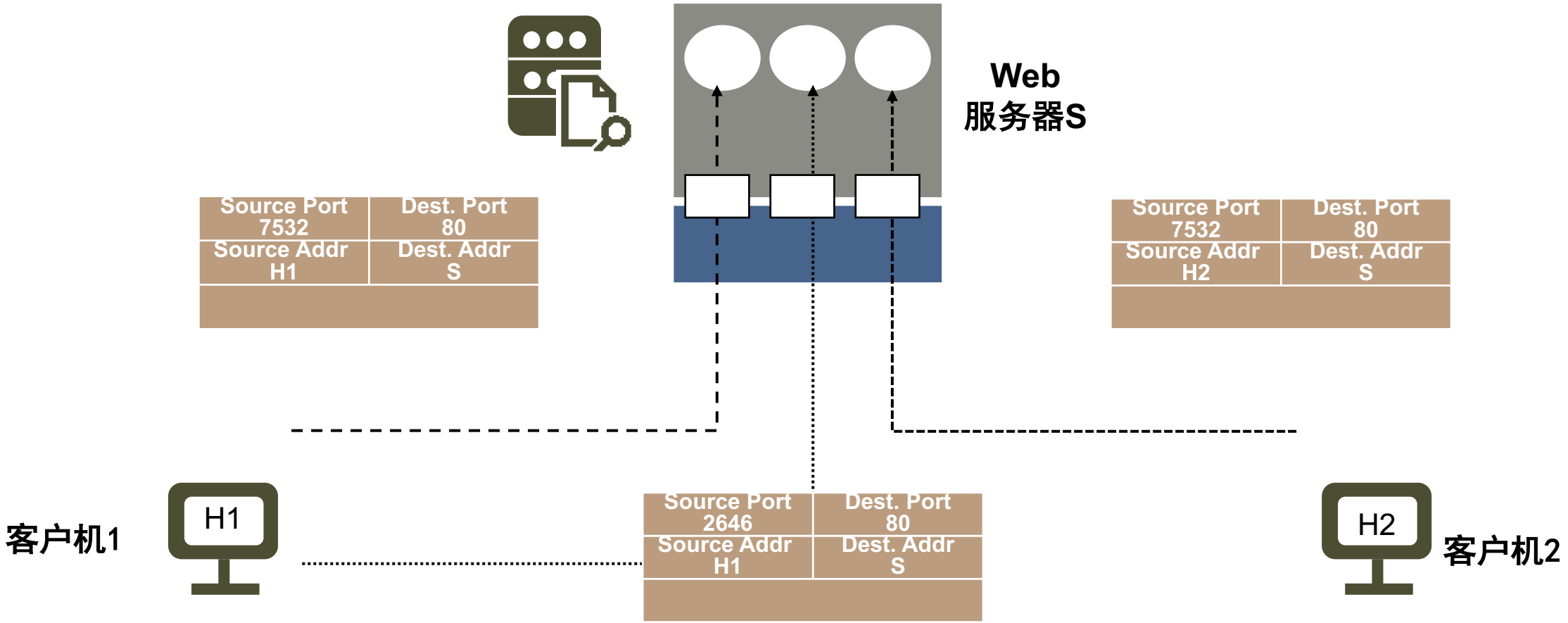
- 服务器有一个迎宾端口负责“引导”
- 每个客户都有自己对应的服务器
- 服务器需要多个端口同时为客户服务

并发型客户-服务器模型类比



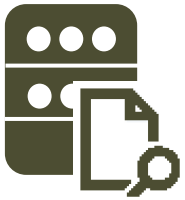
- 有一个负责接待的“迎宾”服务员
- “迎宾”服务员负责为每个客户指定一个具体负责的“接待”服务员
- 每个“接待”服务员为一个特定的客户服务

两个客户机同时与服务器连接

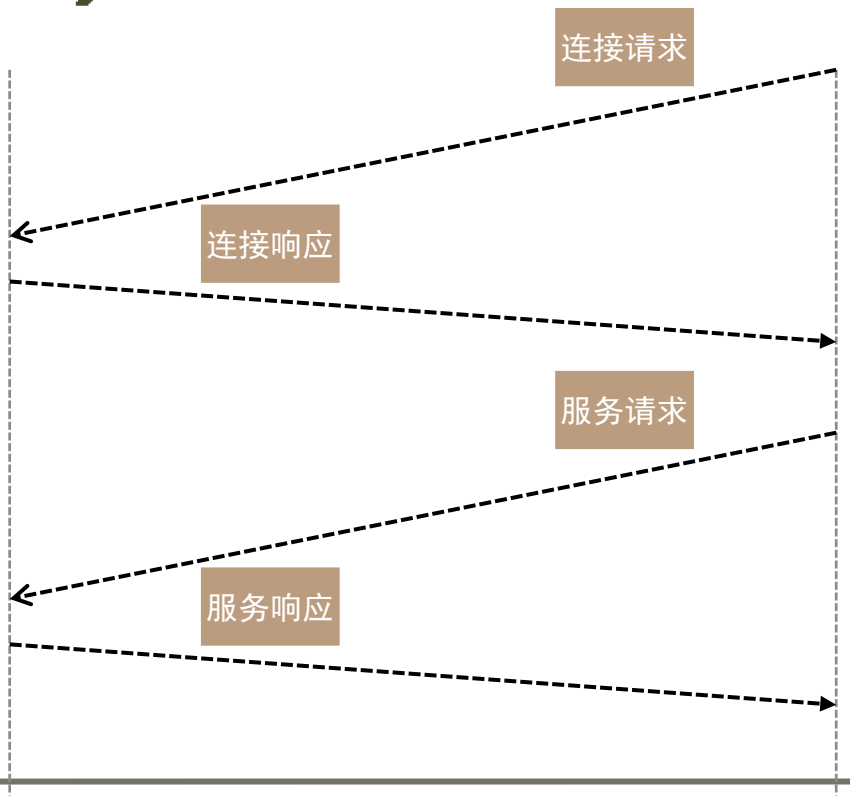


面向连接的多路复用过程

服务器建立一个专门用来接受客户连接请求的“迎宾”端口



客户机创建一个用来向服务器发送请求的“客户”端口



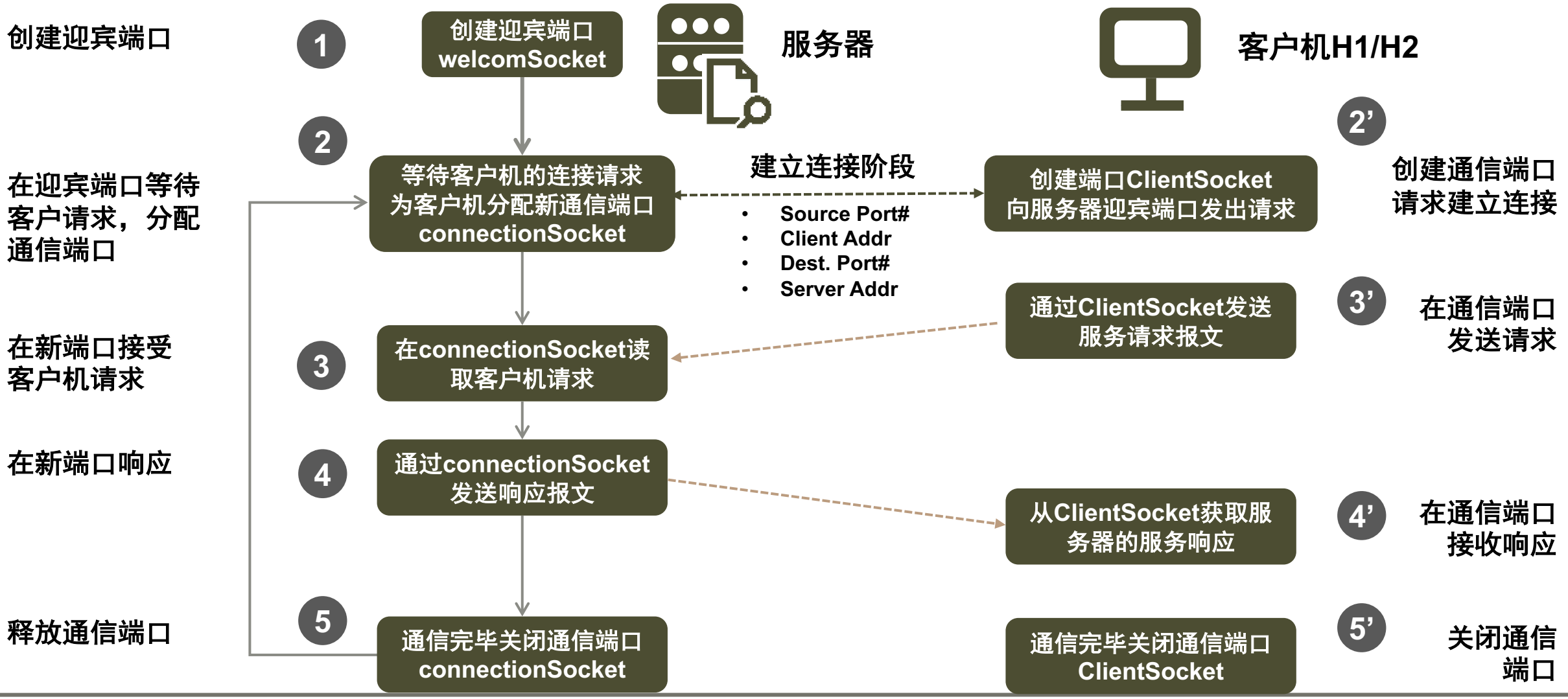
第一阶段 连接请求与响应

客户机向服务器的“迎宾”端口发送连接请求

第二阶段 服务请求与响应

客户机向服务器的“通信”端口发送服务请求

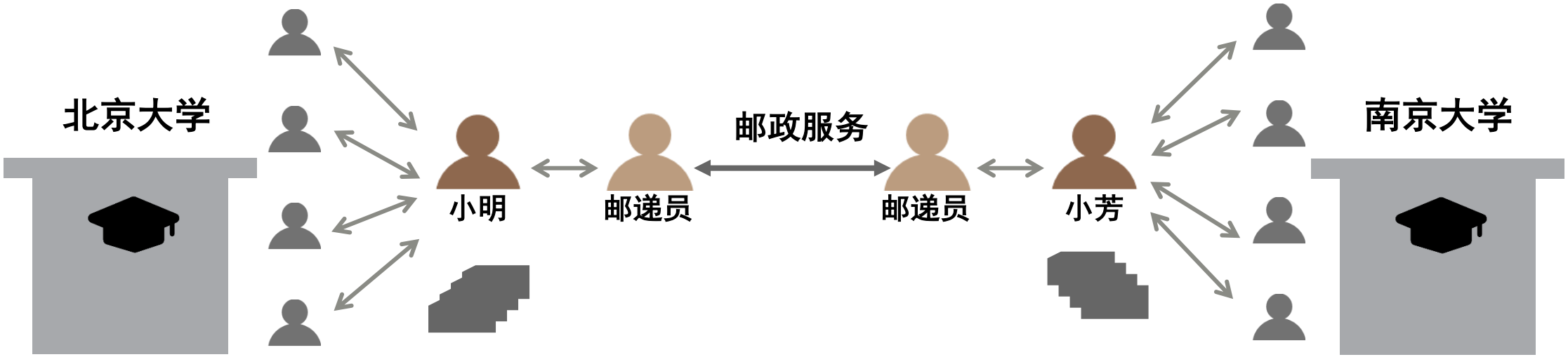
面向连接的多路复用和分用概念模型



传输层概念的示例说明

70/100/140 167/169/172 142/104/78 139/0/18 148/7/9 138/139/133 187/156/127 76/77/50 114/114/114

小明和小芳作为值日生



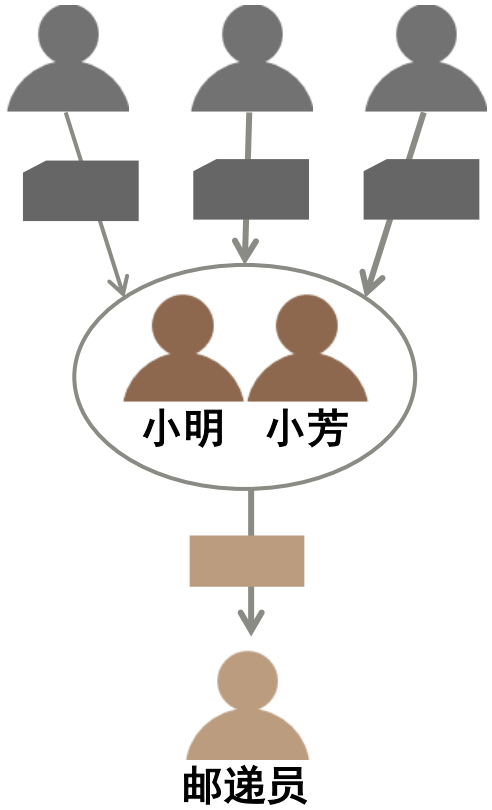
假设：

- 每周每个学生（1，2，3...）给对方学生的每个学生（A, B, C, ...）发送一份学习报告；
- 收到报告的同学修改报告后返回给发送学生。

主机	北京大学、南京大学
进程	N个北大学生、N个南大学生
报文	信件
NL协议	传统邮政服务
TL协议	小明、小芳

应用层报文　传输层报文

应用层传输层网络层接口



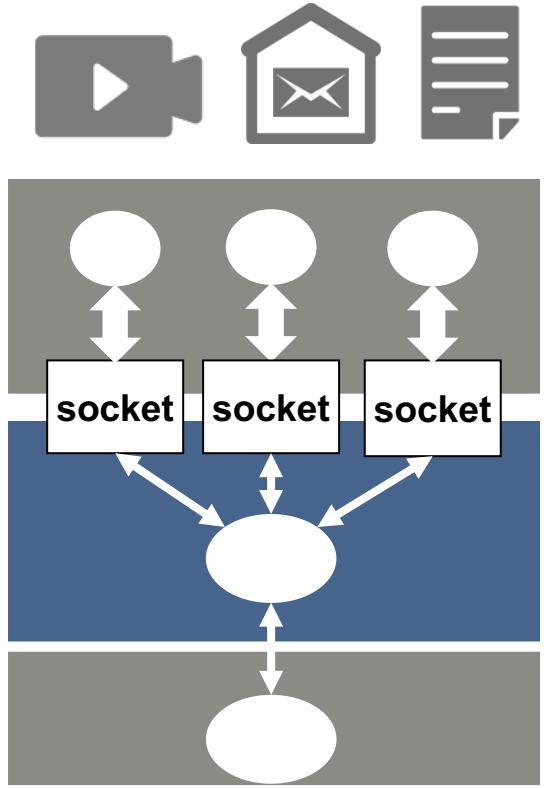
- 小明（小芳）的同学
- 该同学的学习报告

端口号

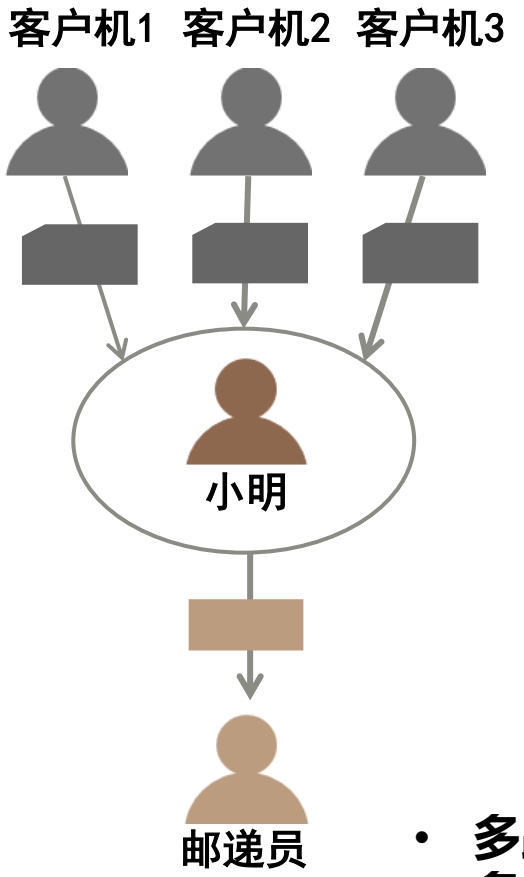
- 小明（小芳）
- 装入了报告的小信封（标识了发送同学和接收同学的姓名）

协议地址

- 邮递员
- 装入了小信封的大信封内（标识了本地地址省市县街道名门牌号）

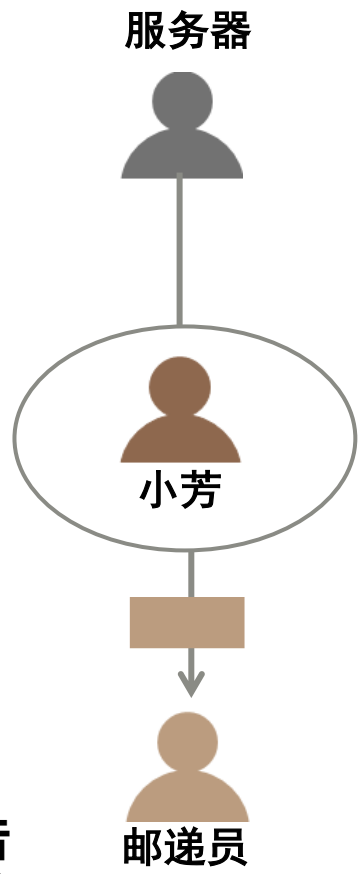


重复型客户机-服务器模型



客户机（每个发信的同学）

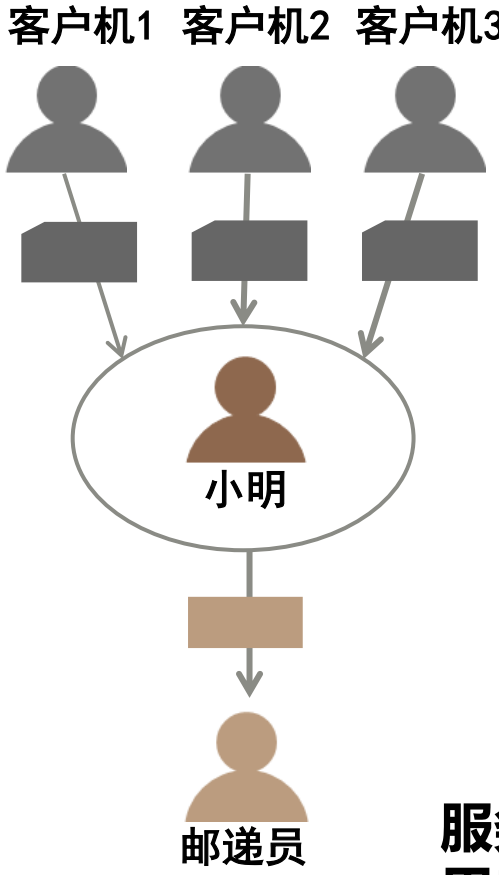
- 小明从同学处挨个收集发给对方某同学A的学习报告
- 小明将报告装入小信封（标注收发姓名）
- 小明将小信封交给邮递员
- 邮递员把小信封装入大信封（标注收发地址）
- 邮递员根据信封上的地址把大信封从路由表指示的方向发出去



- **多路复用：多个同学都通过小明给对方学校的同一个同学A发送学习报告**
- **多路分用：当小明收到对方A同学返回的信件时根据标注的收发姓名分发给每一位同学**

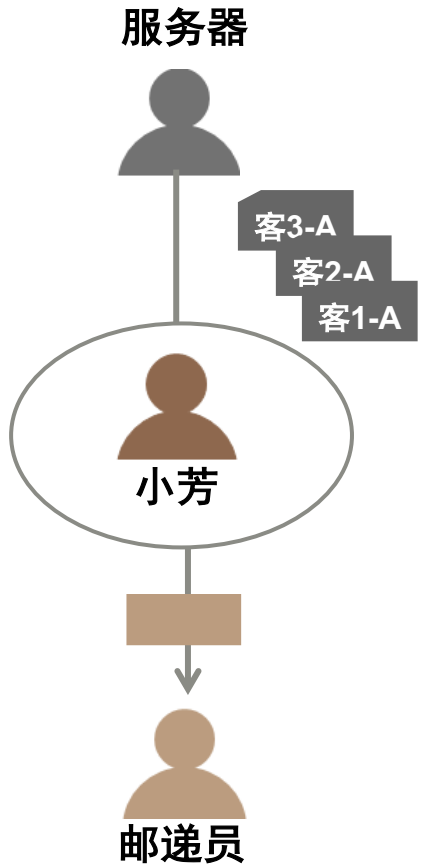
应用层报文　传输层报文

重复型客户机-服务器模型



服务器（每个收信的同学）

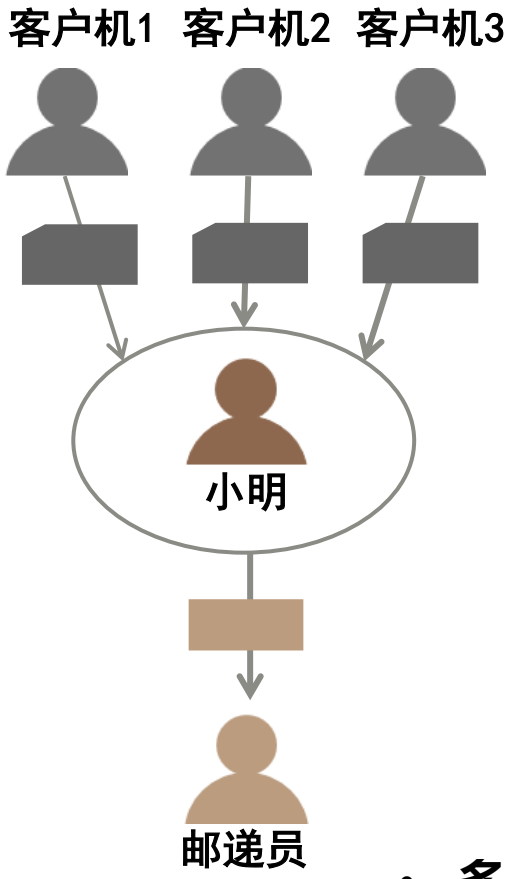
- 邮递员收到一个大信封
- 邮递员拆开大信封，从中取出小信封，交给值日生小芳
- 小芳接收小信封
- 小芳拆开小信封，从中取出学习报告，分发给小信封上姓名标识的同学
- 小芳必须在上交了一封信后，才能处理下一封信



服务器端一个队列意味着小芳必须亲自处理信件，采用串行工作方式，处理完一封信才能处理下一封信。

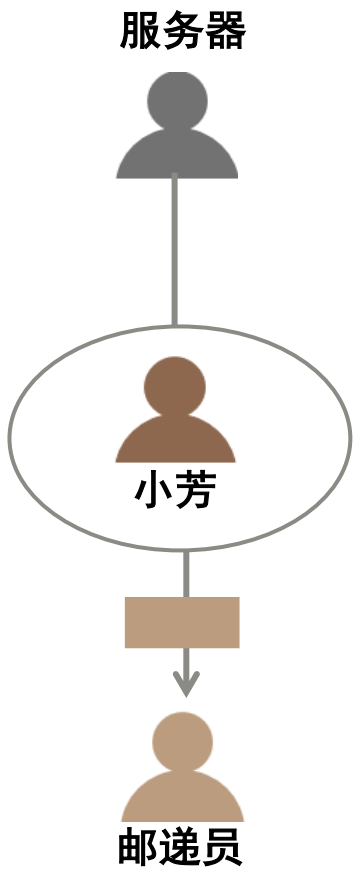
应用层报文　传输层报文

并发型客户-服务器模型



客户机（每个发信的同学）

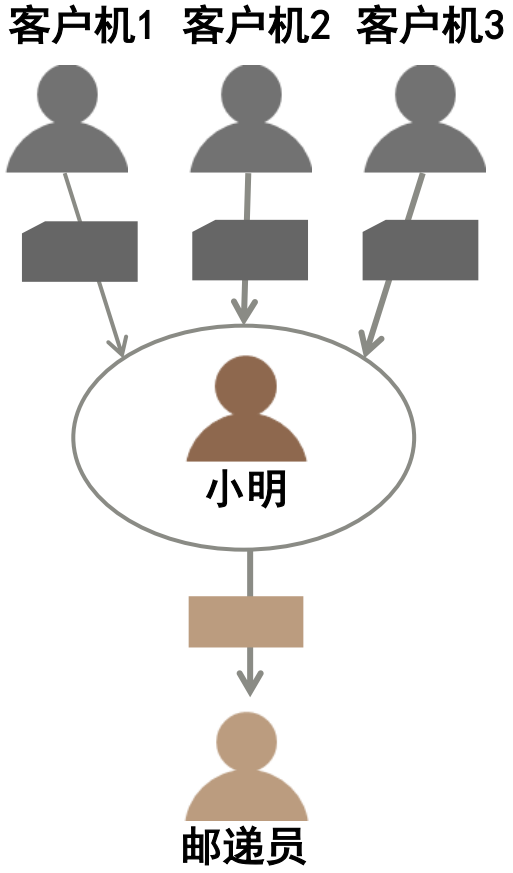
- 小明收到某个学生（进程1，进程2…）发来的学习报告，首先向对方A同学的代理小芳发送连接请求
- 收到小芳回复的肯定响应后，从响应报文得到一个服务专员的名字（新端口号）
- 小明将应用进程发来的学习报告封装在小信封中交给邮递员
- 邮递员把小信封装入大信封（标注收发地址）
- 邮递员根据信封上的地址把大信封从路由表指示的方向发出去



- **多路复用和分用都在客户机这端，小明要记住每个同学发来的报文，并收到服务器返回的响应报文后准确地分发给每个同学。**

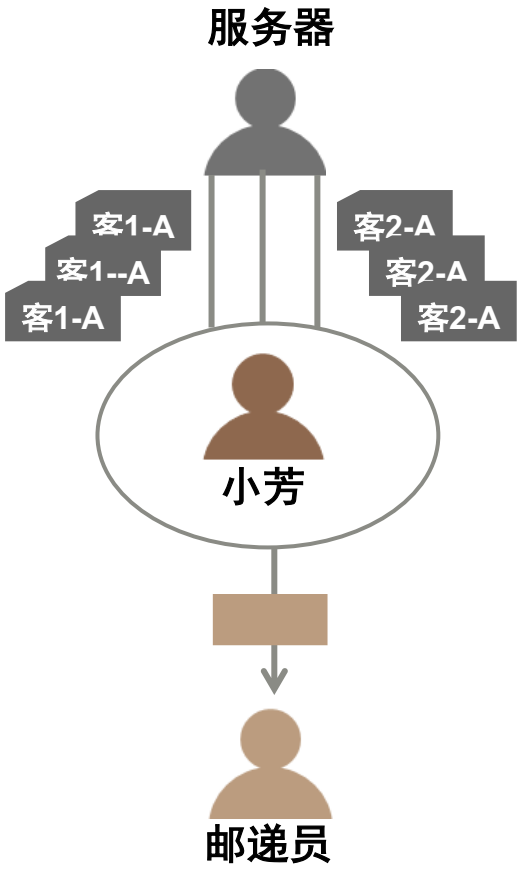
应用层报文　传输层报文

并发型客户-服务器模型



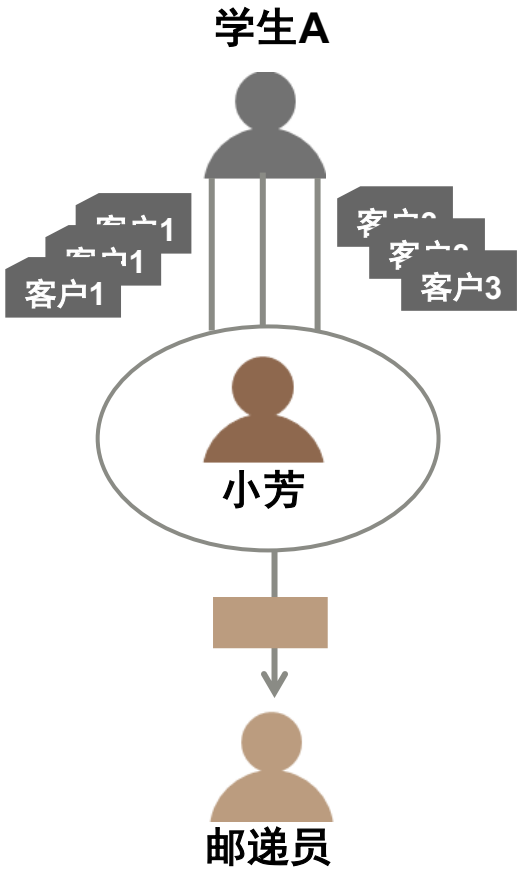
服务器（每个收信的同学）

- 小芳首先通过邮递员收到小明发来（A学生）建立连接请求
- 小芳为该请求分配一个服务专员，并将该专员名字通过响应报文告诉小明
- 小芳只负责接收请求建立连接的报文
- 主动请求连接同学发来的所有信件将由分给他的服务专员接收，并传递给服务器（A学生）。



服务器端的多个队列（一个发送同学一个队列）意味着多个服务专员可并发处理信件。

面向连接的多路复用和分用



小明代表

小芳代表

连接建立阶段

针对所有发送学生

发送连接请求的socket

< 北京大学, 学生, 南京大学, 学生A >

小明代表

服务专员1

数据通信阶段

发送信件时采用的连接

< 北京大学, 学生1, 南京大学, 学生A >

小明代表

服务专员n

发送信件时采用的连接

< 北京大学, 学生n, 南京大学, 学生A >

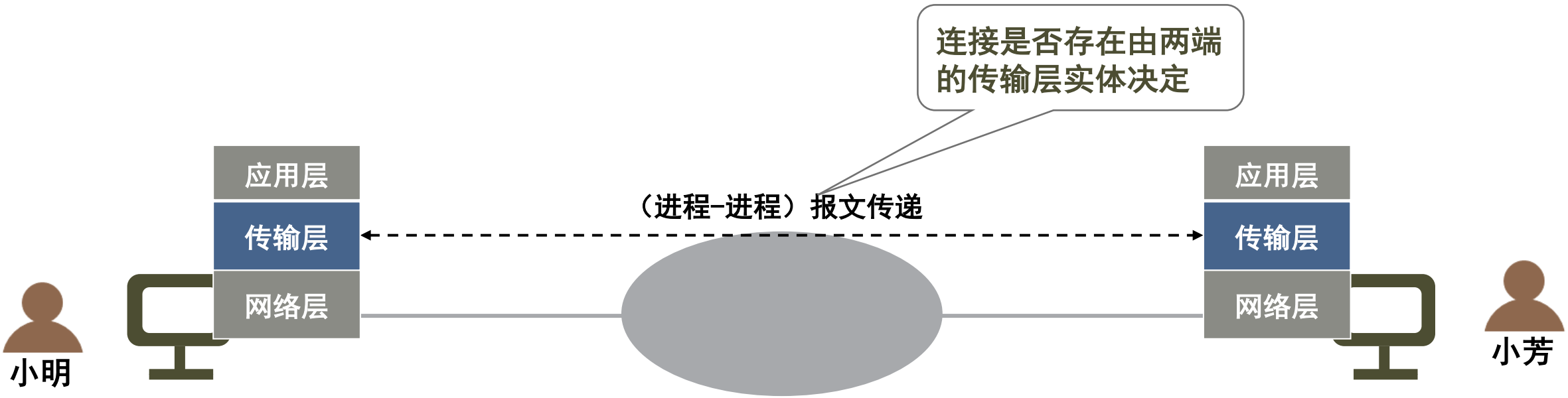
基于可靠网络通信的连接建立

传输层连接

与网络层虚电路的区别

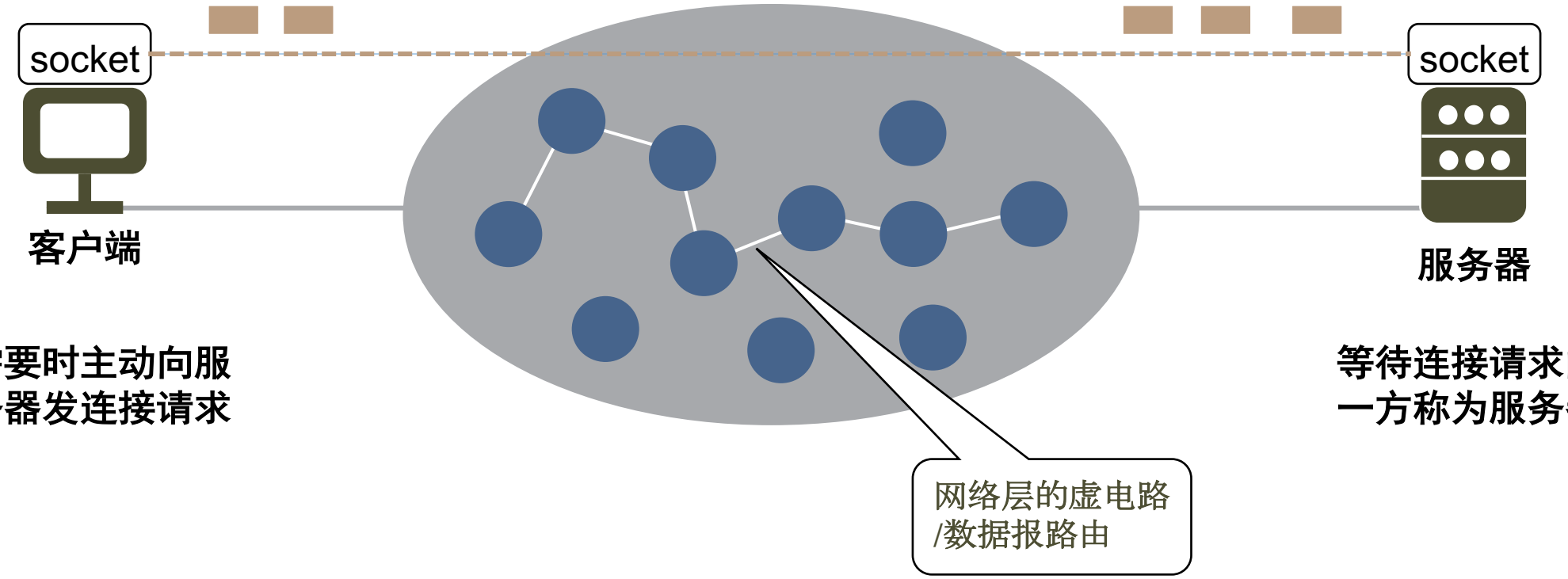
- 网络层的虚电路由每个路由器上的虚电路转换表项维护
- 传输层的连接仅由两个主机上的传输实体维护

- 每一端确保另一端的存在
- 允许两端协商传输参数
- 触发传输实体资源的分配

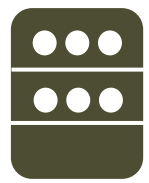


面向连接的网络通信模型

无论网络层的虚电路/路由有无变化，传输层的连接不变。



面向连接的网络通信模型*



服务器

客户端

服务器

客户端

1 为入境请求创建一个socket， port=x:
welcomeSocket = ServerSocket()

2 等待连接请求的到来:
connectionSocket = welcomeSocket.accept()

建立连接

3 创建一个与服务器连接的socket, port=x:
ClientSocket = Socket()

请求服务

用ClientSocket向服务器发送连接请求

从connectionSocket读取客户端的服务请求

4 把服务响应写入
connectionSocket

返回服务结果

从ClientSocket读取返回的服务结果

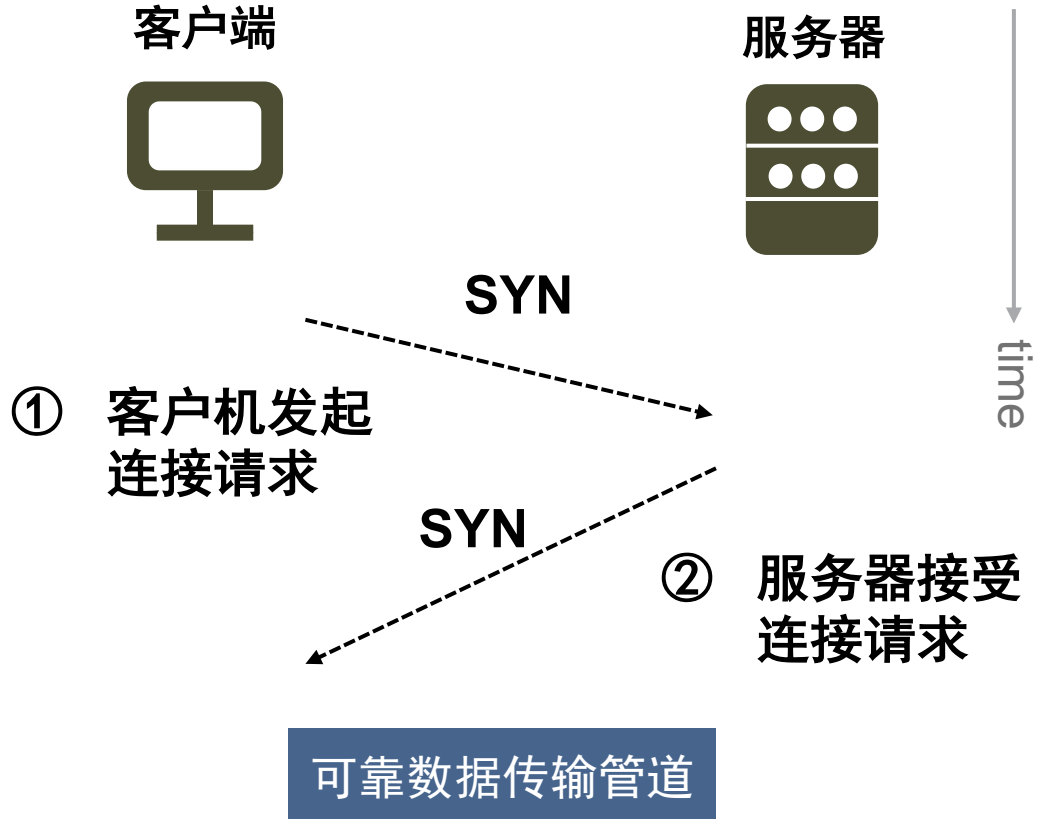
关闭socket
connectionSocket

关闭socket
ClientSocket

基于可靠网络服务的连接建立

“二次握手”方式

- 发起连接请求的传输实体向另一方发送同步(SYN)请求
- 被请求传输实体将该请求排入队列直到传输层用户发出Open

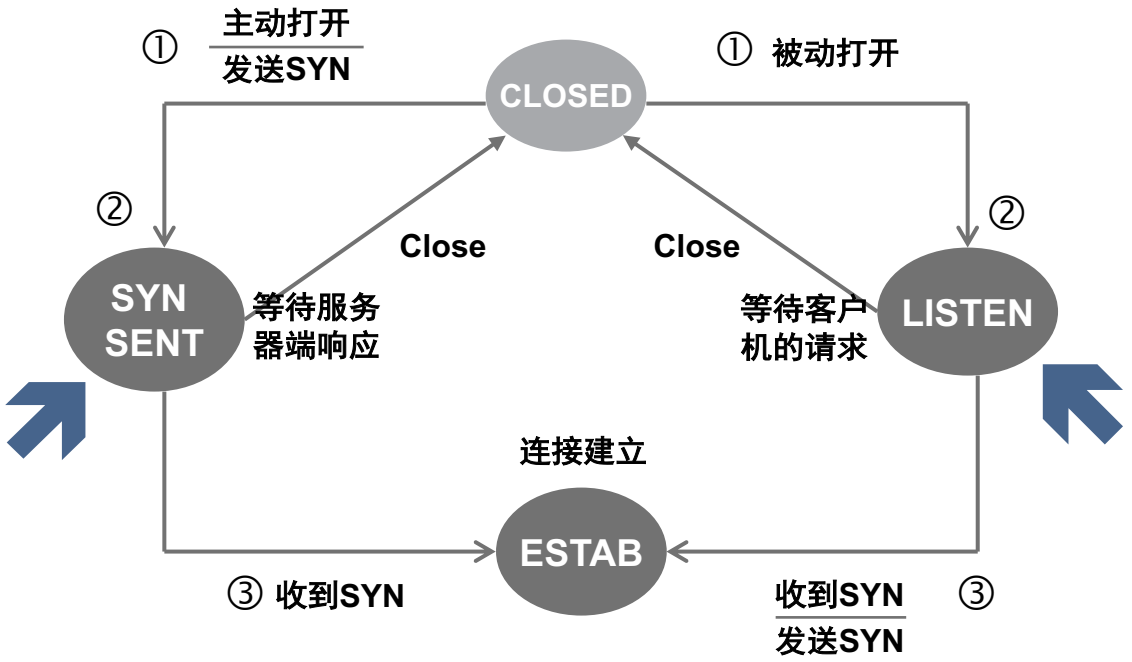


- 被请求传输层实体只能接收连接请求，无权决定是否接受该请求
- 传输层实体将请求转达给上层用户后，等待上层用户的决定，并将决定返回给请求方

二次握手的连接状态机——建立连接

客户端

① 客户端的应用层首先打开一个与服务器的连接
② 传输层实体向服务器端传输层发送建立连接请求SYN报文，进入SYN SENT状态
③ 收到服务器返回的SYN报文，意味着双方的连接已建成，进入ESTAB状态



通常连接是双向的，一旦双方进入ESTAB状态，就可以发送/接收数据。

服务器

① 服务器端的应用层首先打开一个众所周知的端口
② 在该端口上准备就绪，等待客户机的请求，进入LISTEN状态
③ 收到连接请求报文SYN后，返回一个SYN，表示同意建立连接，进入ESTAB状态。

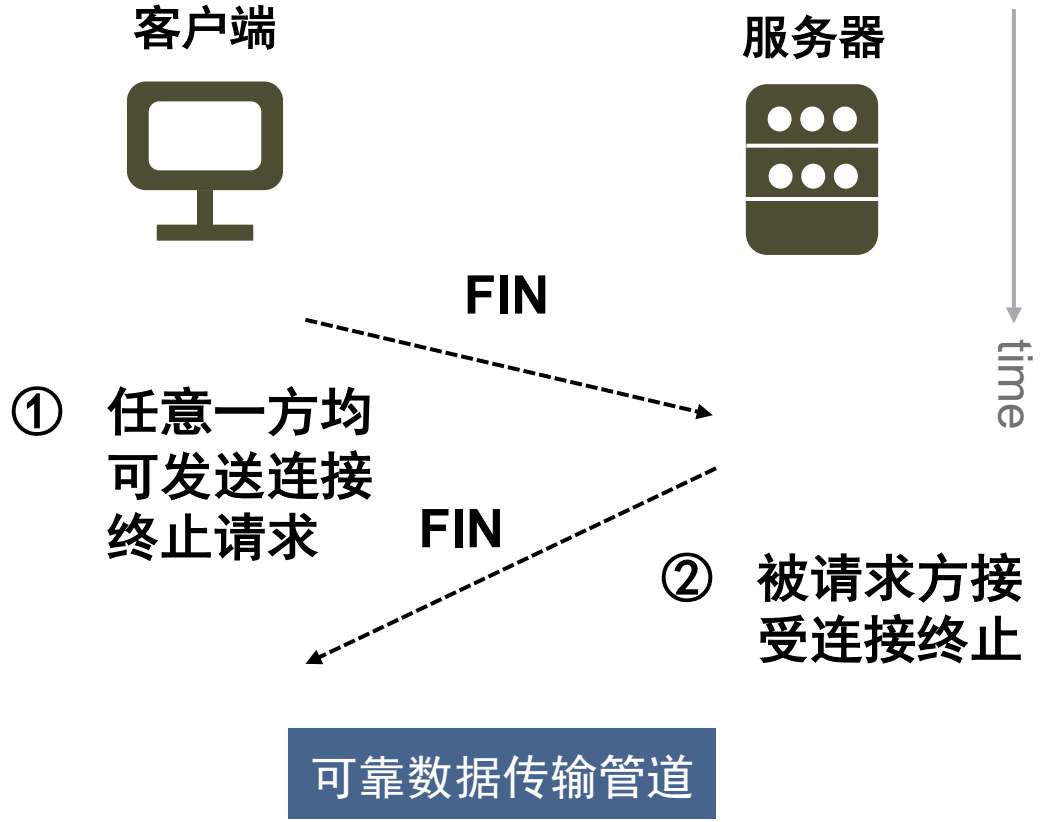
事件
动作

表示发生什么事件后采取什么动作

基于可靠网络服务的连接释放

“二次握手”方式

- 发起连接终止的传输实体向另一方发送连接终止(FIN)请求
- 被请求传输实体将该请求排入队列直到传输层用户发出Close

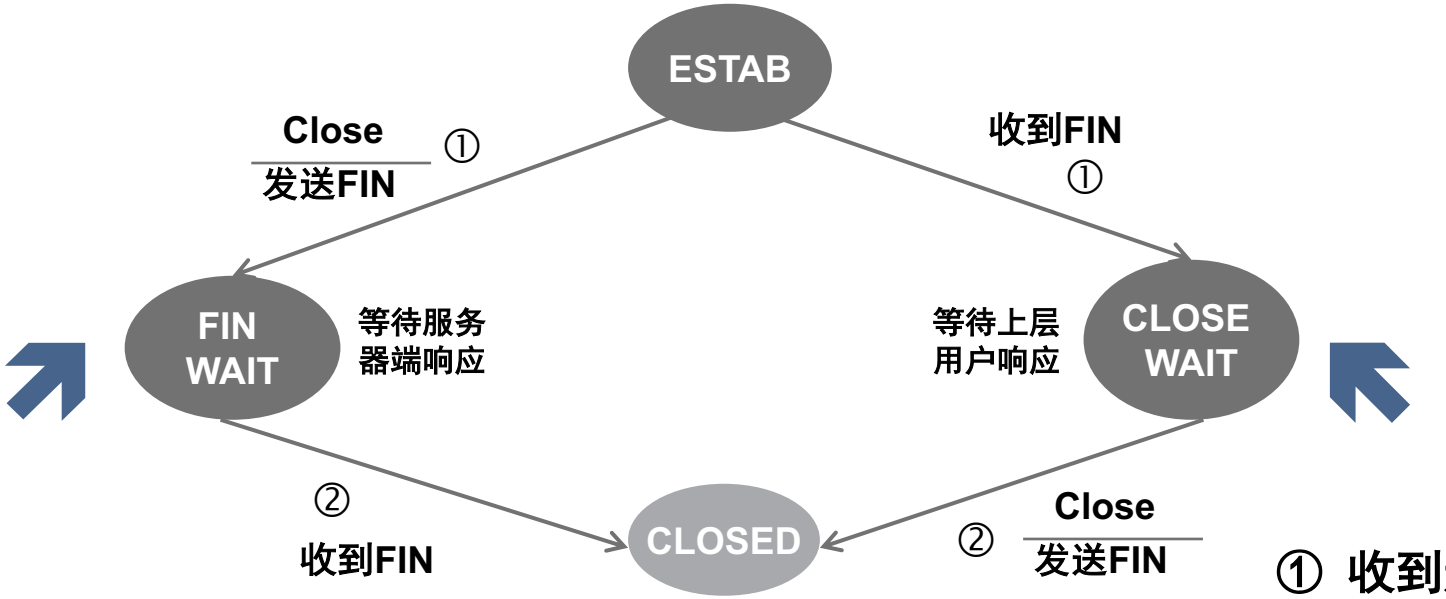


通常连接是双向的，一方发出FIN后，能否继续接收对方发来的数据由具体协议规定。

二次握手的连接状态机

客户端

服务器



① 客户端的应用层close端口后，传输层实体向对方发送连接终止FIN请求报文，进入FIN WART状态等待对方响应。

② 收到返回的FIN报文后意味着对方同意终止连接，关闭连接。

① 收到通信对方发来的连接终止FIN请求报文后，通知应用层，进入CLOSE WART等待其关闭端口。

② 应用层close端口意味着可以终止连接，以FIN报文响应，关闭连接。

事件
动作

表示发生什么事件后采取什么动作

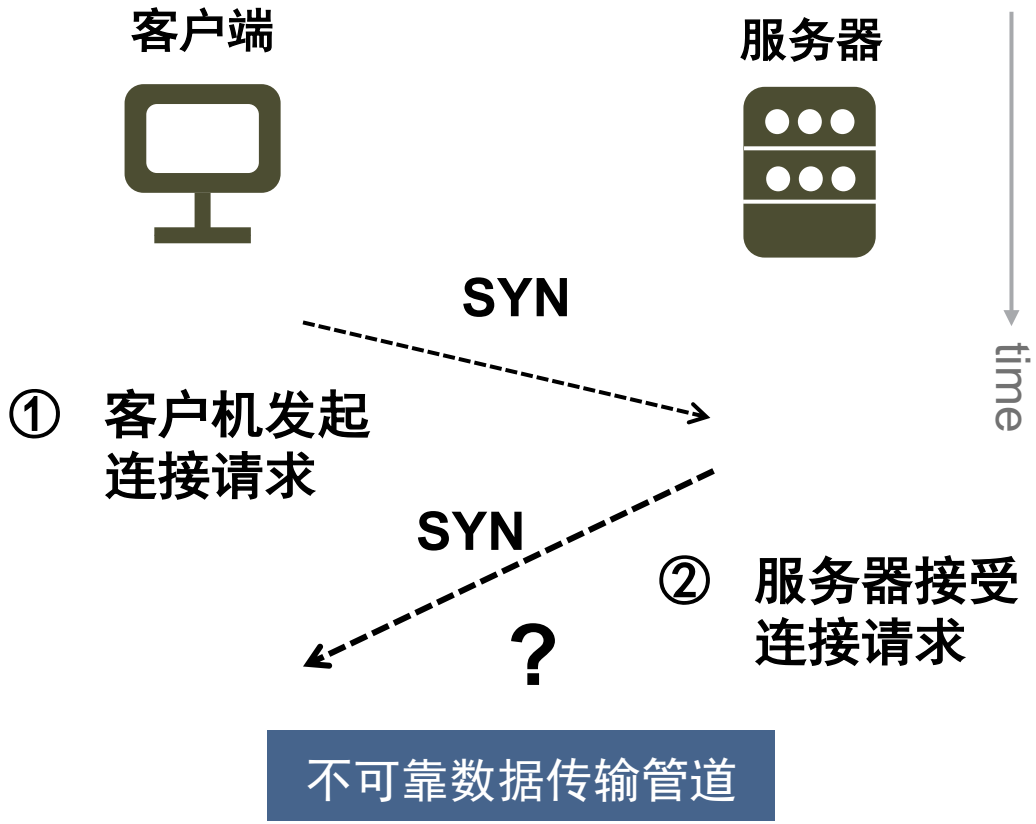
基于不可靠网络通信的连接建立

基于不可靠网络服务的连接建立

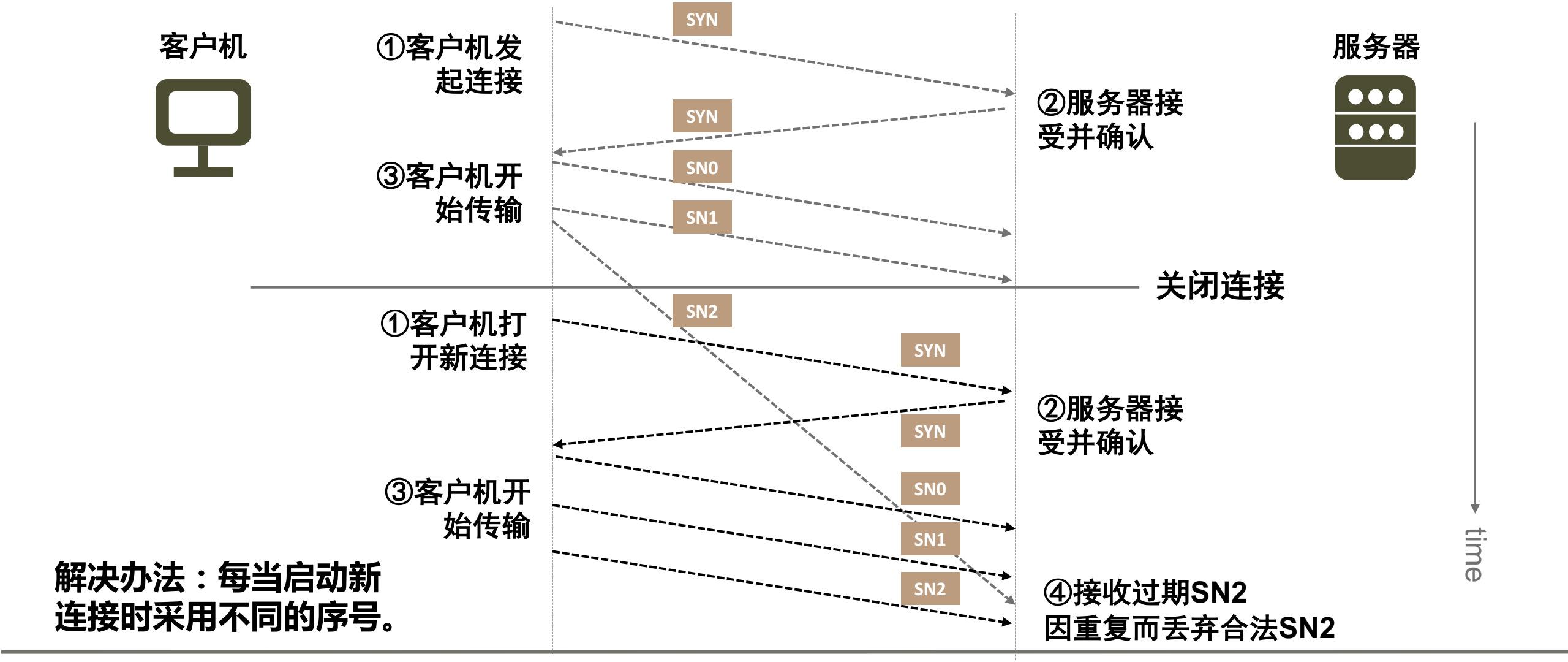
可能发生的错误情况

- 连接发起方的请求SYN丢失
- 连接接受方的应答SYN丢失
- 出现重复SYN

？ 不可靠传输通道在建立连接过程中意味着什么？



序号固定从0开始对连接的影响



解决办法：每当启动新连接时采用不同的序号。

SYN 连接请求报文　　SN0 数据报文

废弃的连接请求SYN对连接的影响

客户机

服务器

废弃

SYNi

①服务器接受连接请求

- 得知客户机的初始序号为i

①客户机发起新连接请求

SYNj

SYNk

②客户机得知服务器的初始序号为j

②服务器丢弃重复连接请求SYN

现在两端都认为已建立合法连接

SNk+1

③因为乱序服务器拒收序号为k+1的报文

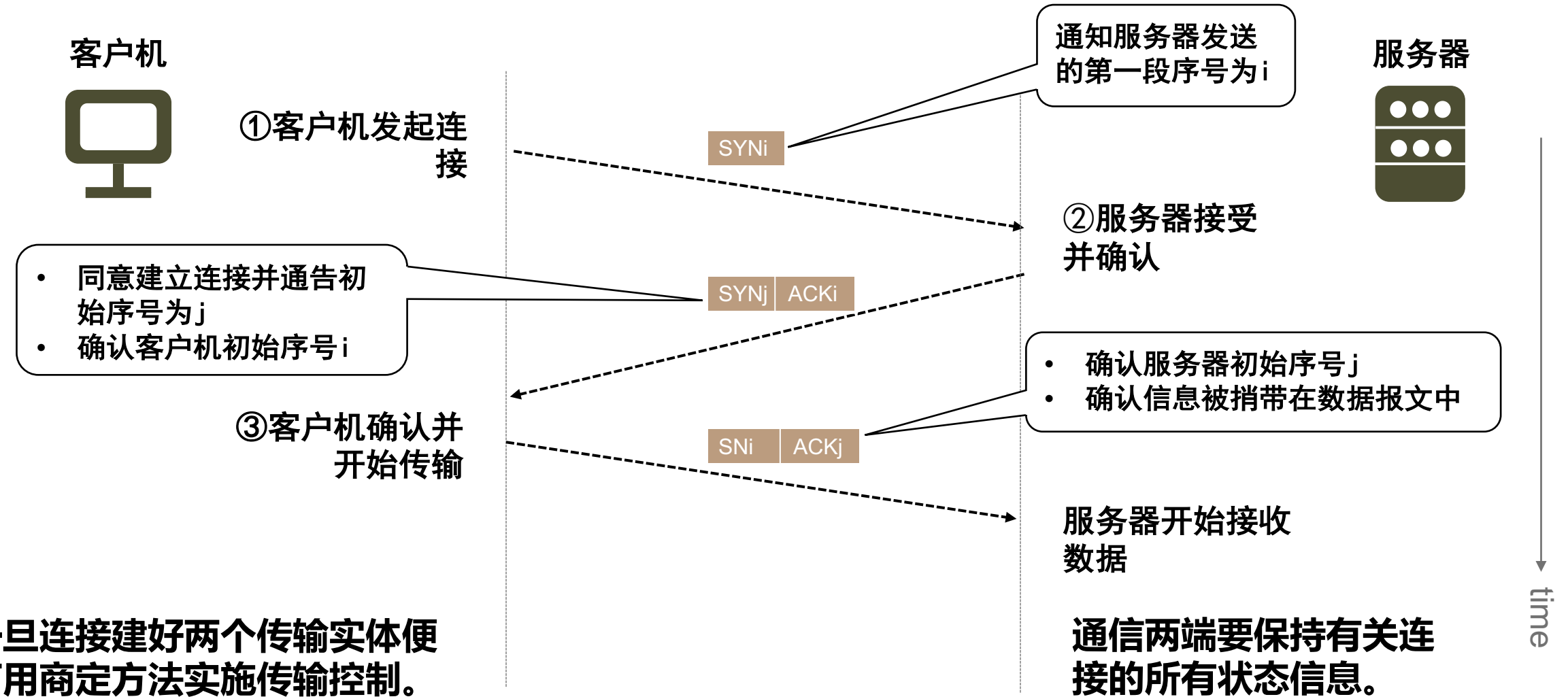
time

解决办法：每一边都显式确认对方的SYN和序号.

北京大学

SYN 连接请求报文　　SN 数据报文

“三次握手”的连接建立过程



一旦连接建好两个传输实体便可用商定方法实施传输控制。

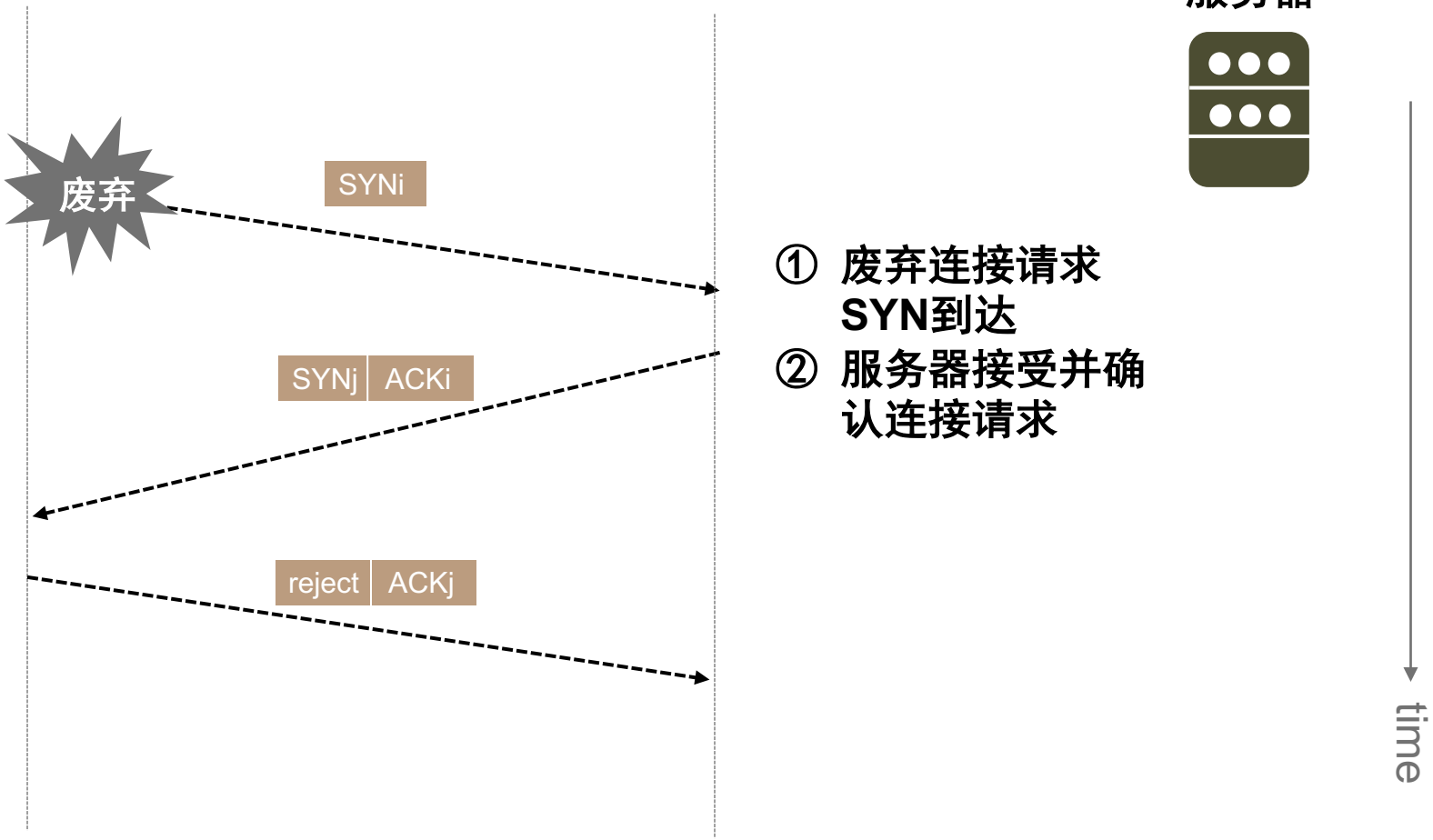
通信两端要保持有关连接的所有状态信息。

SYN 连接请求报文　　SNO 数据报文

“三次握手”对重复连接请求SYN的处理

客户机

服务器



① 废弃连接请求SYN到达

② 服务器接受并确认连接请求

③ 客户机意识到服务器确认的是一个已经废弃的连接请求

④ 客户机拒绝服务器的连接

SYN 连接请求报文

reject 拒绝请求

“三次握手”对废弃连接响应SYN的处理

客户机

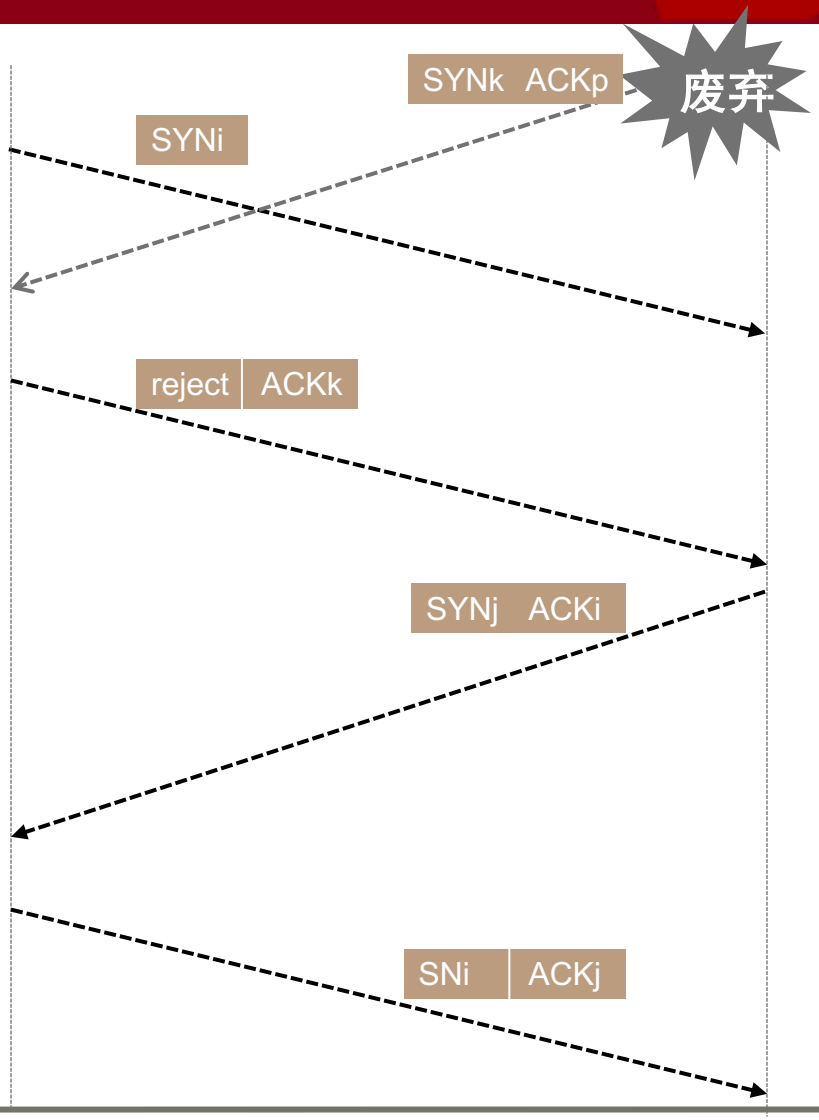
服务器

① 客户机发起新的连接请求

② 废弃连接响应SYN到达，客户机拒绝

③ 客户机等待服务器对新请求的确认

⑤ 客户机确认并开始传输



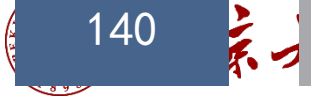
② 服务器收到客户机的最新连接请求

③ 服务器接受客户机连接请求

废弃的SYN响应不会对新连接的建立产生影响。

SYN 连接请求报文　SNO 数据报文　reject 拒绝请求

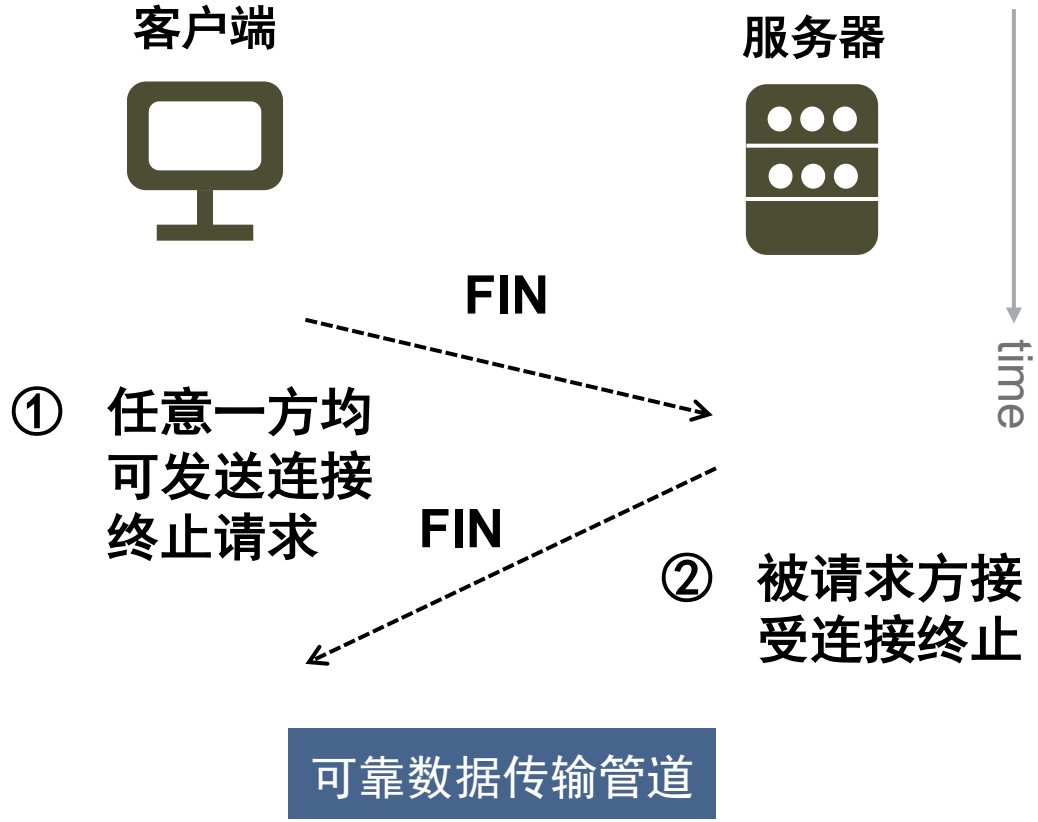
连接管理之连接释放



70/100/140 167/169/172 142/104/78 139/0/18 148/7/9 138/139/133 187/156/127 76/77/50 114/114/114

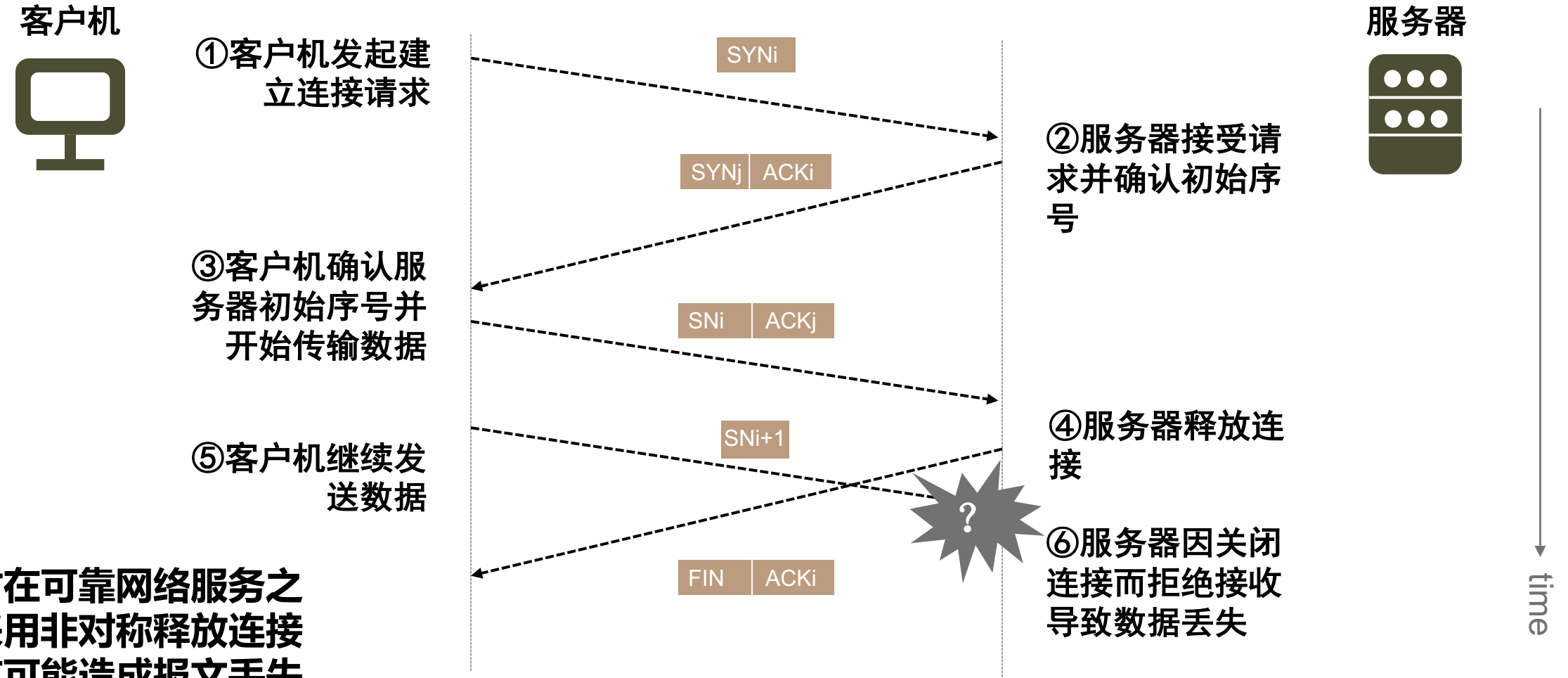
可靠网络服务之上连接终止

非对称方式	对称方式
• 连接的任何一方均可向对方发释放连接请求 • 一旦该请求到达对方连接即告终止	• 终止连接后不能发数据但仍能接收 • 只有在双方均终止连接后连接才算彻底终止



- 通常连接被用于全双工操作，终止应该在两个方向上进行
- 终止连接只是关闭了发送通道，仍然能接收数据（这是对方的发送通道）

可靠网络服务之上连接释放



即时在可靠网络服务之上采用非对称释放连接也有可能造成报文丢失

SYN 连接请求报文　　SNO 数据报文

不可靠网络服务上连接释放问题

两军对垒问题：最后发出信息的蓝军指挥官永远无法确定信息是否安全到达对方。

猜疑链：你会猜疑我是怎么想的，我会猜疑你是怎么想的；就算你知道我是怎么想的，我也知道你是怎么想的，但你还会猜疑我是怎么想你的，我也会猜疑你是怎么想我的...

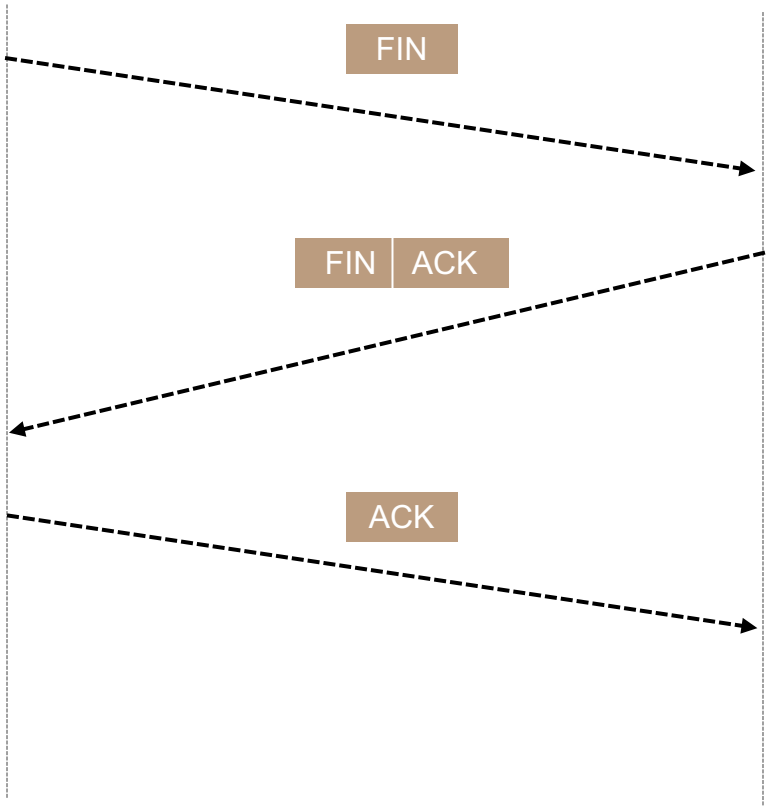
假设：蓝军和绿军是敌对两军，力量强弱如图所示。

现在：左蓝军指挥官决定凌晨5点发起进攻，派了一个通信兵把进攻事宜带给右蓝军指挥官。

“三次握手”方式释放连接

客户机

①客户机发连接终止请求FIN
- 启动定时器



③释放连接并确认对方的FIN

服务器

②服务器接受终止请求，以FIN报文响应
- 启动定时器

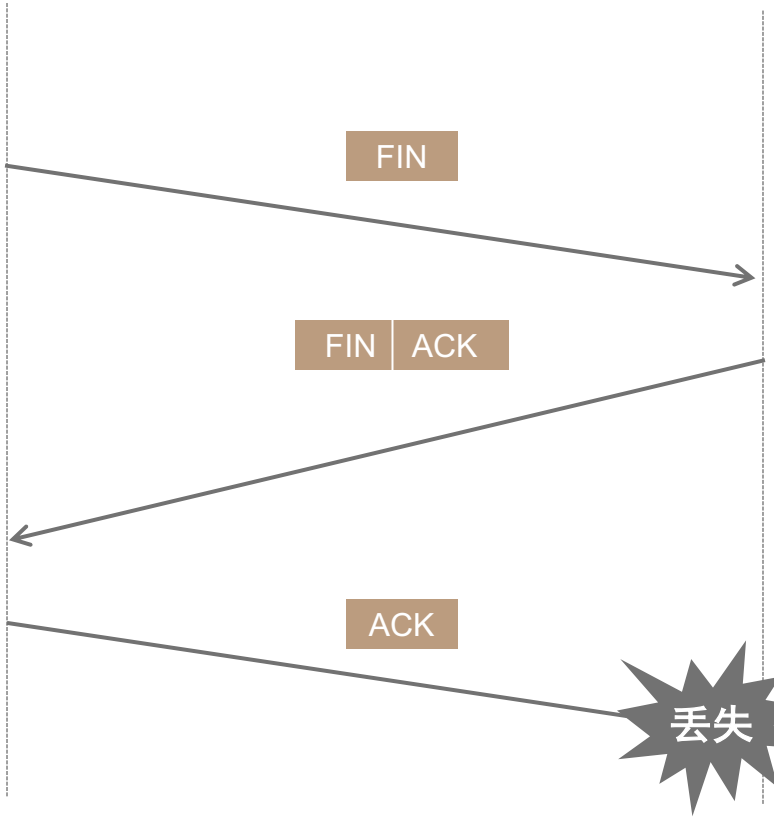
④释放连接

time

“三次握手”释放对最后确认丢失的处理

客户机

①客户机发连接
终止请求FIN
- 启动定时器



服务器

②服务器接受连接
终止
- 启动定时器

（超时）
释放连接

③释放连接
- 确认ACK

time

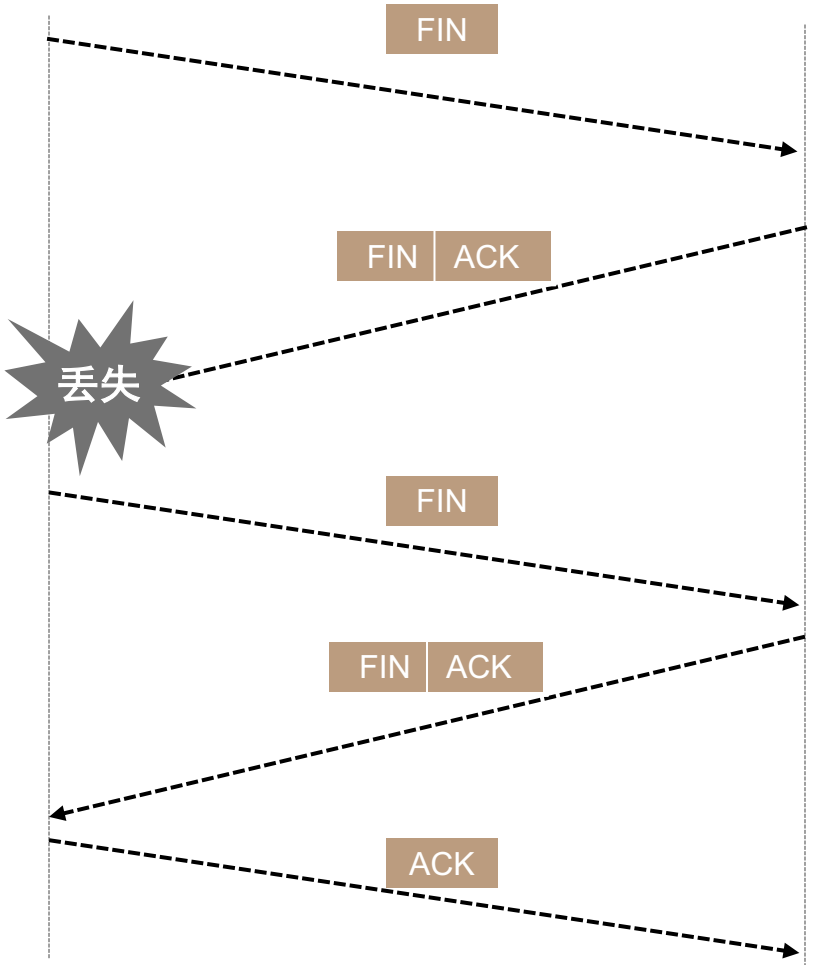
“三次握手”释放对响应丢失的处理

客户机

①客户机发连接终止请求FIN
- 启动定时器

③客户机重发连接终止FIN
- 启动定时器

⑤释放连接
- 确认ACK



服务器

②服务器接受连接终止
- 启动定时器

④服务器发连接终止FIN
- 启动定时器

⑥释放连接

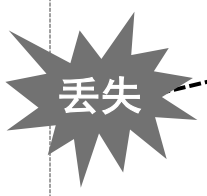
time

“三次握手”释放对响应确认均丢的处理

客户机

①客户机发连接终止请求FIN
- 启动定时器

FIN



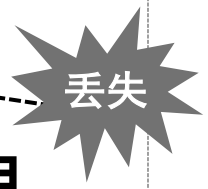
FIN ACK

②服务器接受连接终止请求
- 启动定时器

（超时）释放连接

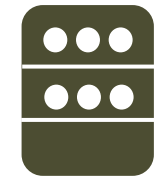
③客户机重发连接终止请求FIN
- 启动定时器

FIN



（N次超时）释放连接

双方设立定时器可用来防止出现物理故障



服务器

time

传输层的可靠数据传输


70/100/140
167/169/172
142/104/78
139/0/18
148/7/9
138/139/133
187/156/127
76/77/50
114/114/114

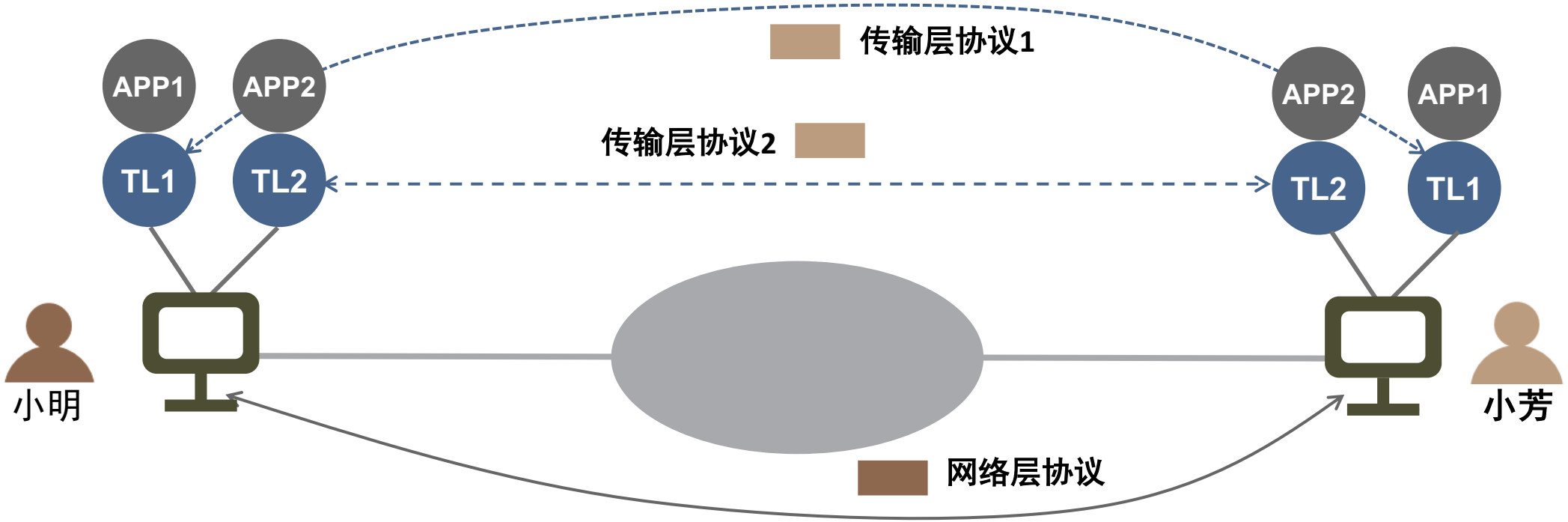
传输层使用网络层服务

可靠网络服务

- 不会丢包
- 不会损坏
- 保证顺序

不可靠网络服务

- 包可能会丢失
- 可能重复损坏
- 可能乱序



基于可靠的网络传递服务

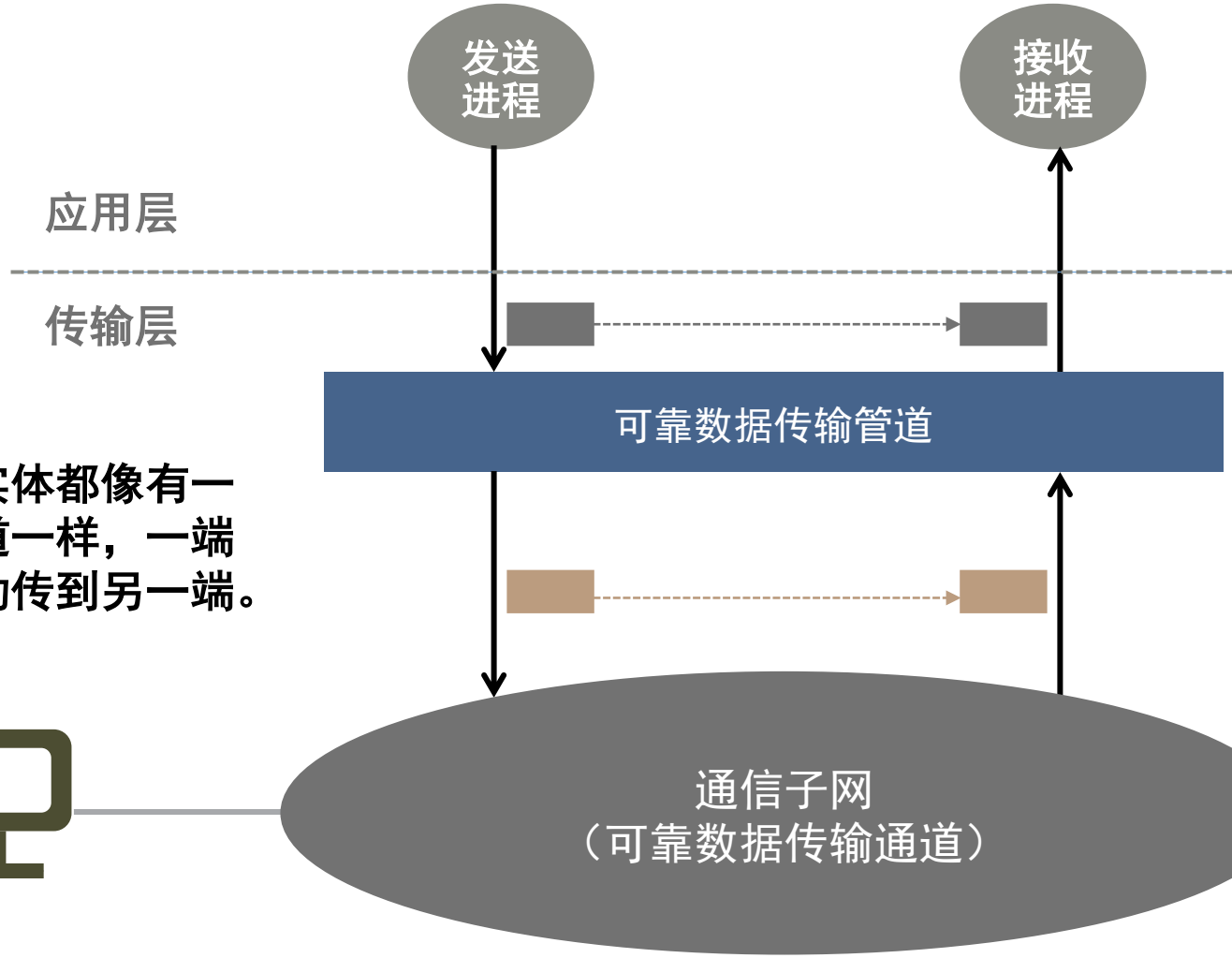
假设：

- 底层网络提供可靠数据传输服务
- 接收端有能力及时接收数据

特性

- 发送端发出的报文按需到达接收端
- 无需反馈机制

可靠数据传输基本概念



应用进程和传输层实体都像有一根可靠数据传输管道一样，一端发出的数据原封不动传到另一端。

？ 如何在可靠网络之上实现可靠数据传输管道？

应用层数据 传输层报文

基于可靠网络数据通道的rdt1.0

假设：

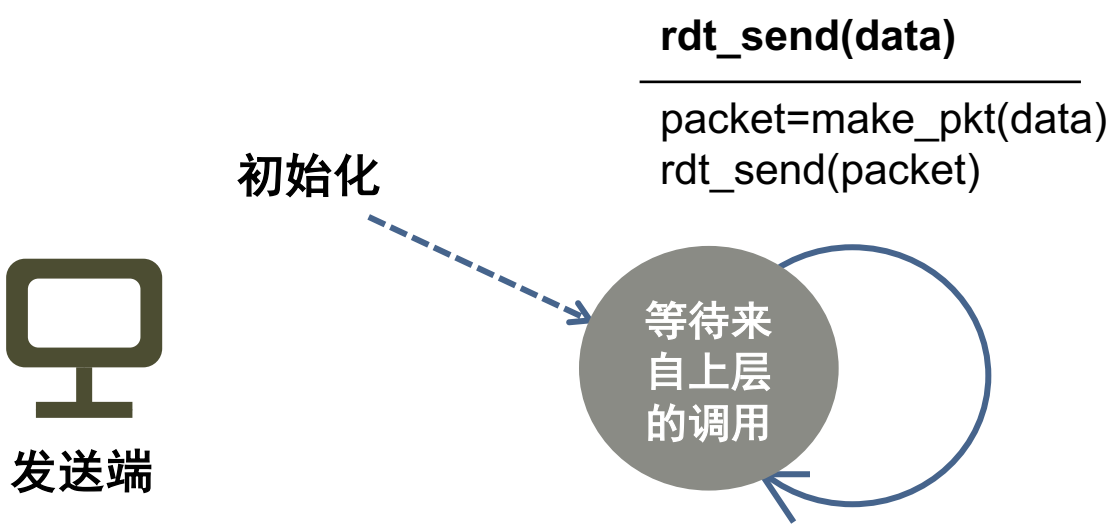
- 底层网络提供可靠数据传输服务
- 接收端有能力及时接收数据

现在：设计一个可靠的数据传输协议

可靠数据传输rdt1.0

- rdt_send(data)：发送data
- rdt_rcv(data)：接收data

可靠数据传输协议rdt1.0——发送端



上层应用进程调用rdt_send()发送数据data触发下列动作：

① 生成一个包含了上层数据data的包packet

② 调用rdt_send()发送包

事件

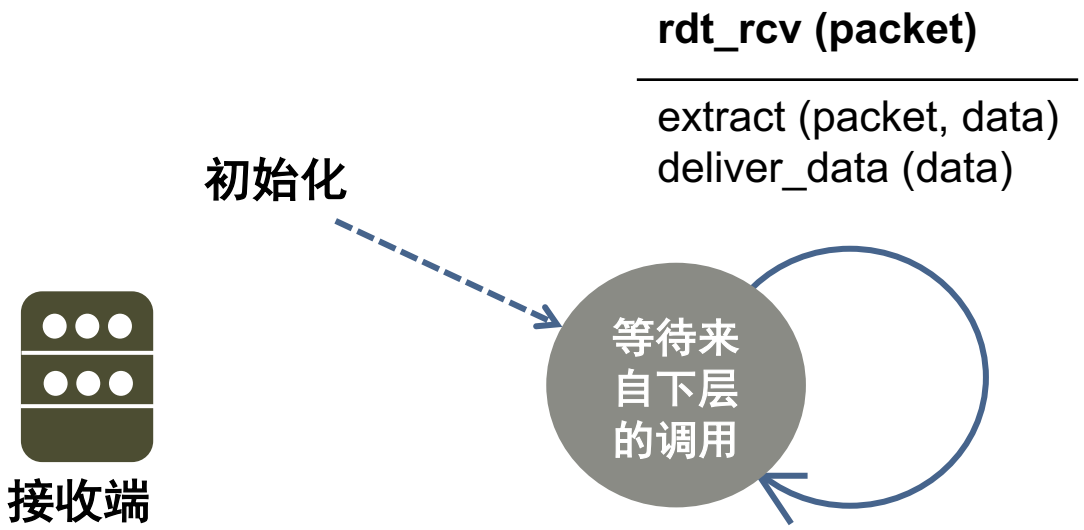
动作

定义了当“事件”发生后，传输层实体采取的“动作”

- 发送端传输层实体每次被动地从应用层获得DATA，调用底层可靠的网络传输服务把封装了DATA的包发送给对等的传输层实体
- 可靠的网络传输通道确保接收端的传输层实体一定收到该包

注意：示例中略去了传输层的端口号分配等事宜

可靠数据传输协议rdt1.0——接收端



底层网络调用rdt_rcv()把报文传到传输层触发下列动作：

① 从报文packet中取出上层数据data

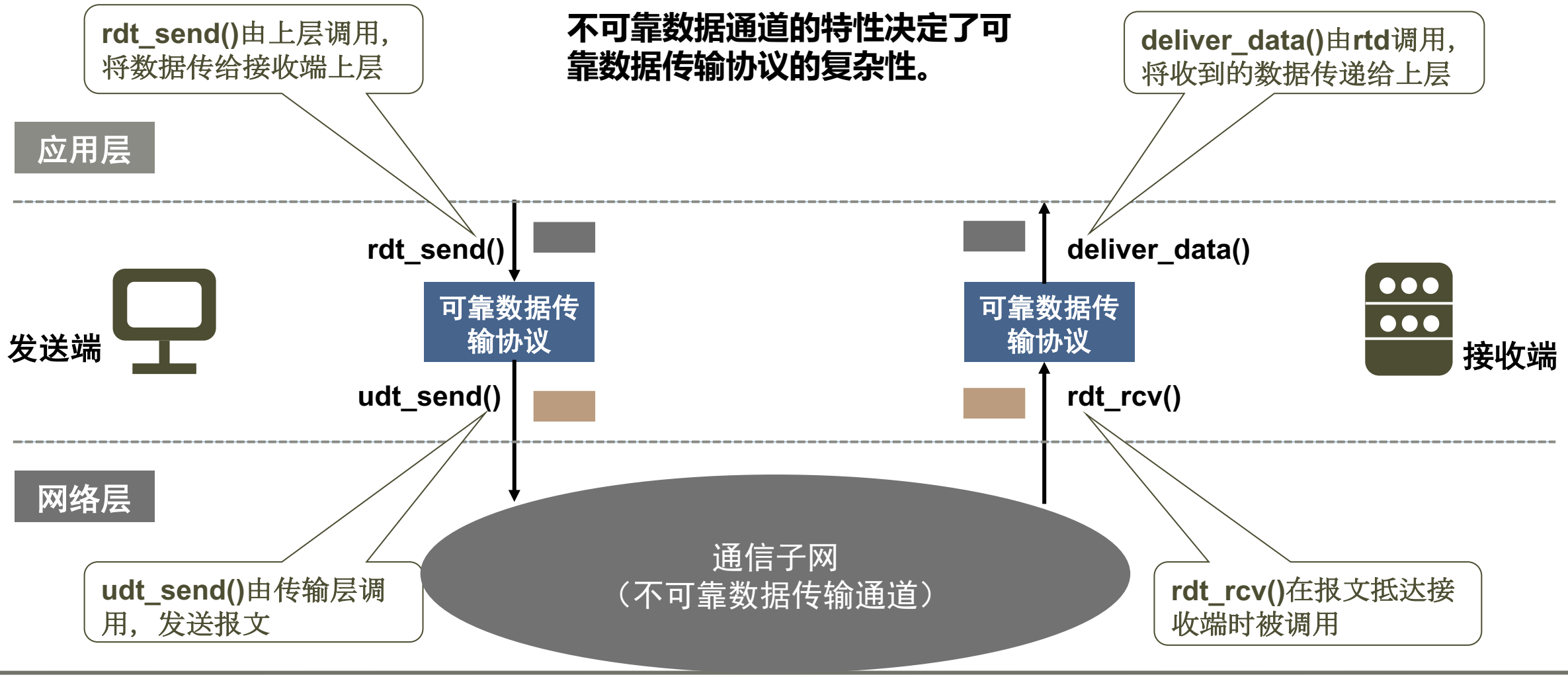
② 调用diliver_data将数据data传给上层应用进程

- 接收端传输层实体等待底层可靠的网络层实体的调用，接收网络层传上来的报文packet
- 传输层实体把报文中的DATA传递给上层应用进程后继续等待下一个报文的到来

事件
动作

定义了当“事件”发生后，传输层实体采取的“动作”

基于不可靠的网络传递服务



rdt: 可靠数据传输　　udt: 不可靠数据传输

传输层的可靠数据传输协议设计示例

基于不可靠的网络数据通道的传输协议

新增三种功能

- 差错检测手段（软件校验和等）
- 接收端的反馈，告知接收情况
- 发送端重发机制，重传输出错数据

可靠数据传输协议rdt2.0

假设

① 报文在传输过程中可能出错
② 报文在传输过程中不会丢失
③ ACK和NAK在传输过程中不会出错
④ ACK和NAK在传输过程中不会丢失

- 携带传输层报文的包可能在传输过程中被损坏，导致收到的传输层报文出现错误
- 传输层报文不会在传输过程中被丢失
- 接收端通过反馈机制向发送端报告接收状况

- rdt_send(data)：发送data
- rdt_rcv(data)：接收data

----------控制功能----------

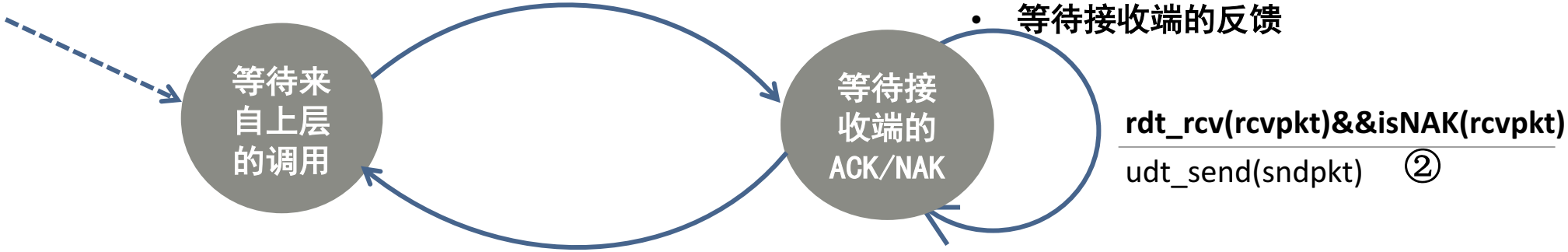
- isACK(rcvpkt)：收到的报文是否为肯定确认(ACK)
- isNAK(rcvpkt)：收到的报文是否为否定确认(NAK)
- corrupt(rcvpkt)：收到的报文在传输过程中是否出错
- notcorrupt(rcvpkt)：收到的报文是否无错

注意：示例中略去了端口号分配等事宜

可靠数据传输协议rdt2.0——发送端

发送端

初始化



rdt_send(data)

sndpkt=make_pkt(data,checksum)

udt_send(sndpkt) ①

①发送端将应用进程数据封装成报文发送

- 为协助接收端校验是否出现错误，报文中要加入校验和
- 调用发送函数发送 报文
- 等待接收端的反馈

rdt_rcv(rcvpkt)&&isNAK(rcvpkt)

udt_send(sndpkt) ②

rdt_rcv(rcvpkt)&&isACK(rcvpkt)

Λ ③

③如果收到接收端反馈的肯定确认，说明包已经被接收端完美接收

- 返回初始状态，等待上层应用进程的调用

②如果收到接收端反馈的否定确认，说明包在传输过程中被破坏需要重发

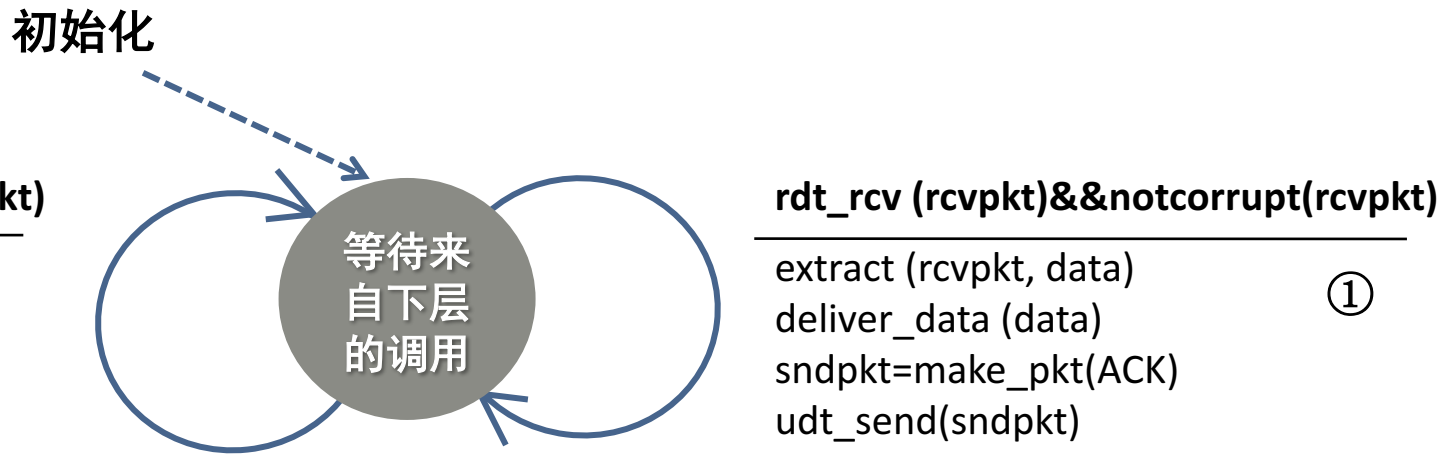
- 调用数据发送函数重发包
- 等待接收单的反馈

事件 / 动作

定义了当“事件”发生后，传输层实体采取的“动作”

可靠数据传输协议rdt2.0

接收端



②收到的报文被校验出有错，立即给发送端反馈否定确认，等待报文的重传。

①从校验正确报文中取出包含的数据交给上层应用进程，并给发送端反馈肯定确认后进入等待接收状态。

可靠数据传输协议rdt2.1

假设

- 报文在传输过程中可能出错
- 报文在传输过程中不会丢失
- ACK和NAK传输可能出错
- ACK和NAK不会被丢失

- 接收端要检查数据报文的正确性
- 发送端要检查确认信息是否正确。

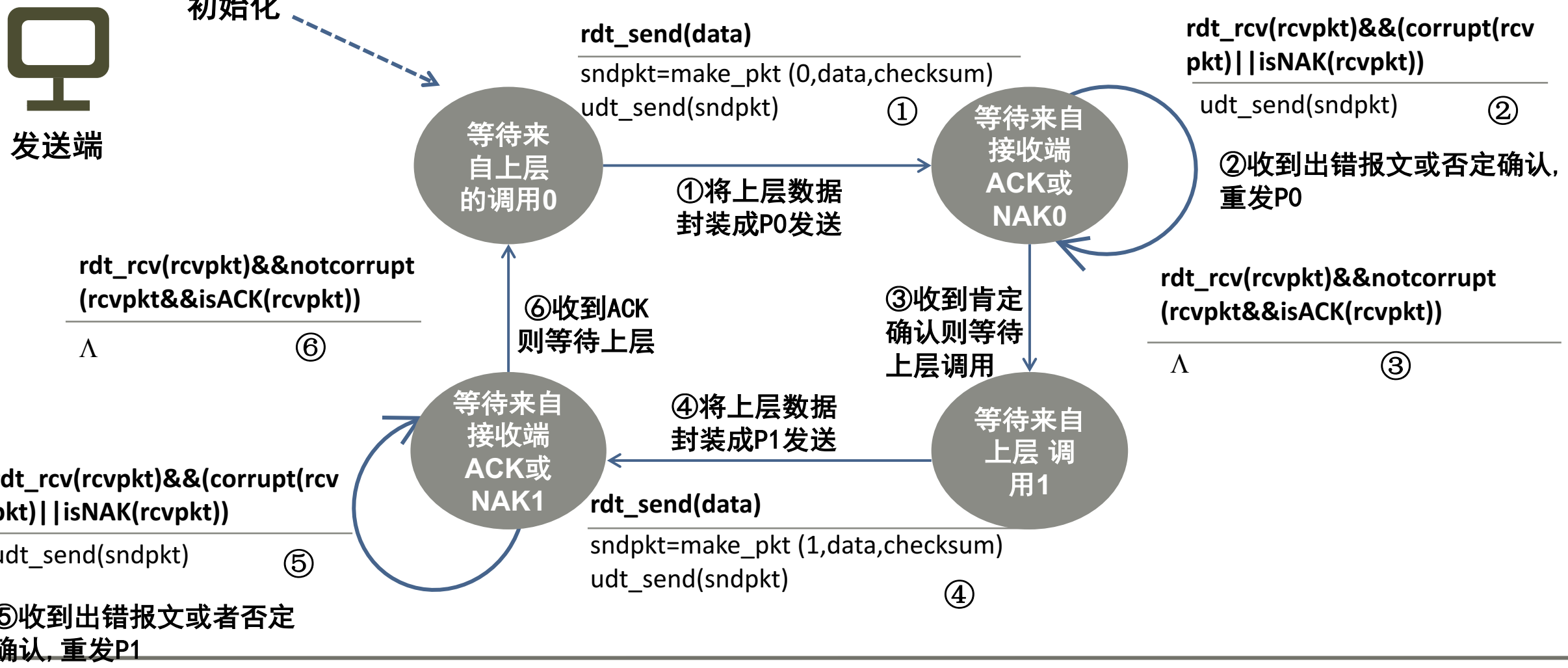
- rdt_send(data)：发送data
- rdt_rcv(data)：接收data

----------控制功能----------

- isACK(rcvpkt)：收到的报文是否为肯定确认(ACK)
- isNAK(rcvpkt)：收到的报文是否为否定确认(NAK)
- corrupt(rcvpkt)：收到的报文在传输过程中是否出错
- notcorrupt(rcvpkt)：收到的报文是否无错

注意：示例中略去了端口号分配等事宜

可靠数据传输协议rdt2.1——发送端



可靠数据传输协议rdt2.1——接收端

rdt_rcv(rcvpkt)&&corrupt(rcvpkt)

sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)

②校验有错反馈NAK

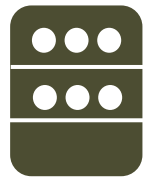
rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq0(rcvpkt))

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pk(ACK, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)&&(corrupt(rcvpkt)

sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)

⑤校验有错则反馈NAK



接收端

等待下层调用0

等待下层调用1

①将P0数据交给上层，反馈ACK

④将P1的数据交给上层，反馈ACK

③对重复包则重发ACK

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq1(rcvpkt))

sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq1(rcvpkt))

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pk(ACK, checksum)
udt_send(sndpkt)

⑥对重复包则重发ACK

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq0(rcvpkt))

sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

可靠数据传输协议rdt2.2

假设

- 报文在传输过程中可能出错和丢失
- 肯定确认ACK传输可能出错和丢失

- 用一种只有肯定确认机制完成可靠传输
- 接收端必须给出ACK号
- 发送端必须检查收到的ACK号

- rdt_send(data)：发送data
- rdt_rcv(data)：接收data

----------控制功能----------

- isACK(rcvpkt)：收到的报文是否为肯定确认(ACK)
- corrupt(rcvpkt)：收到的报文是否在传输过程中出错
- notcorrupt(rcvpkt)：收到的报文是否无措

*为简单起见这里给出的ACK号为接收到的包序号

注意：示例中略去了端口号分配等事宜

可靠数据传输协议rdt2.2——发送端

发送端

初始化

等待来自上层调用0

rdt_send(data)

sndpkt=make_pkt (0,data,checksum)

udt_send(sndpkt)

①将上层数据封装成P0发送

等待来自接收端ACK 0

rdt_rcv(rcvpkt)&&(corrupt (rcvpkt)||isACK(rcvpkt,1))

udt_send(sndpkt)

②收到出错报文或确认P1，重发P0

rdt_rcv(rcvpkt)&&notcorrupt (rcvpkt&&isACK(rcvpkt， 0)

Λ

③收到确认P0，无需动作，等上层调用

等待来自上层调用1

④将上层数据封装成P1发送

等待来自接收端ACK1

rdt_send(data)

sndpkt=make_pkt (1,data,checksum)

udt_send(sndpkt)

rdt_rcv(rcvpkt)&&(corrupt (rcvpkt)||isACK(rcvpkt,0))

udt_send(sndpkt)

⑤收到出错报文或确认P0，重发P1

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt&&isACK(rcvpkt,1)

Λ

⑥收到确认P1，无需动作，等上层调用

可靠数据传输协议rdt2.2——接收端

接收端

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq0(rcvpkt))

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)

①将P0数据交给上层,反馈确认P0

rdt_rcv(rcvpkt)&&(corrupt(rcvpkt)||has_seq0(rcvpkt))

udt_send(sndpkt)

②对出错报文或乱序报文，则重发确认P0

等待下层调用0

等待下层调用1

③将P1数据交给上层,反馈确认P1

④对出错/乱序报文重发确认P1

rdt_rcv(rcvpkt)&&(corrupt(rcvpkt)||has_seq1(rcvpkt))

udt_send(andpkt)

rdt_rcv(rcvpkt)&&notcorrupt(rcvpkt)&&has_seq1(rcvpkt))

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1, checksum)
udt_send(sndpkt)

拥塞控制之

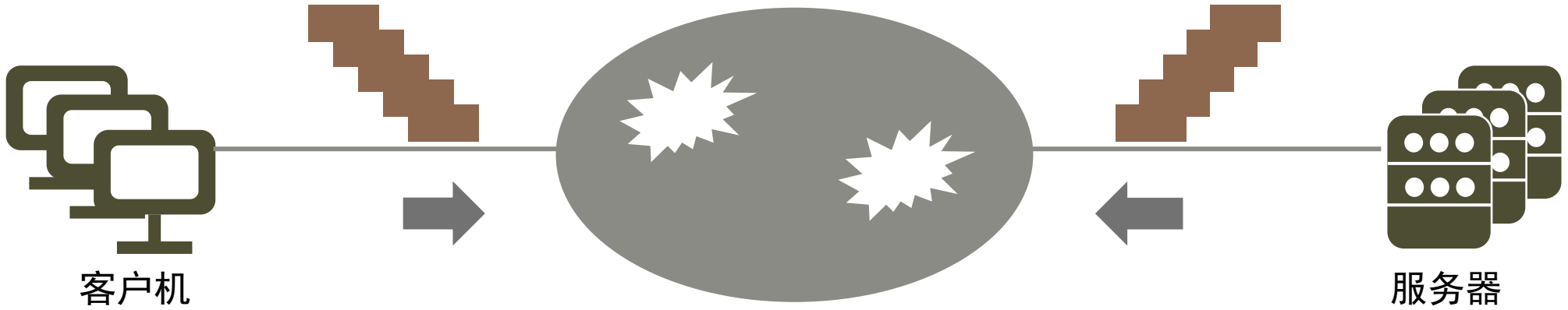
拥塞的形成以及危害

什么是拥塞?

拥塞：太多的发送端给网络发送了太快太多的数据，导致网络来不及处理而出现堆积在某个区域。

网络拥塞后果

- 队列延迟加大
- 路由器的缓冲区溢出(丢失包)



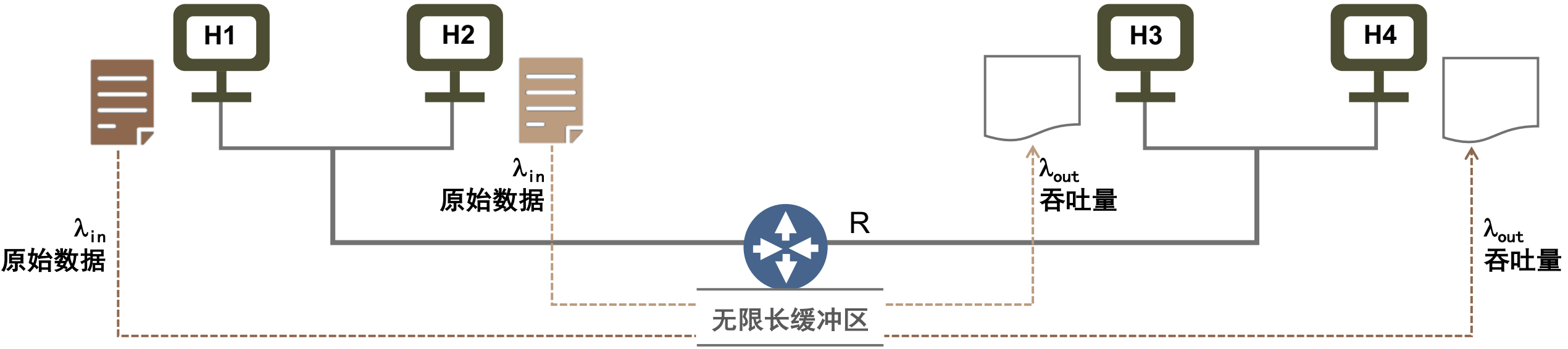
如果路由器具有无限长队列

假设：如图所示网络中

- 路由器有无限大缓冲区，出境链路容量为R
- 无错误控制、无流量控制、无壅塞控制

特点

- 无需重发
- 带宽利用率最大
- 队列延迟增大



λ_{in} ： 发送主机应用程序发出的原始数据（率）

λ_{out} ：连接的吞吐量（接收端的每秒字节数）

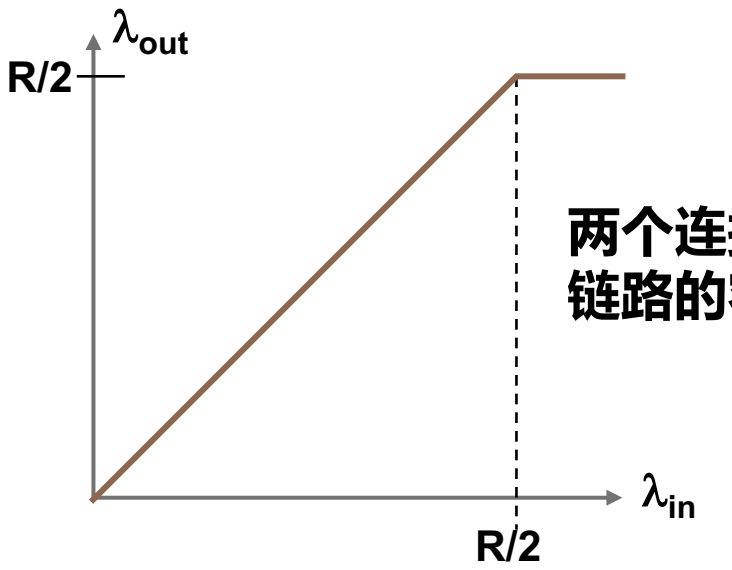
吞吐量、包延迟与发送速率

发送端速率在0~R/2之间

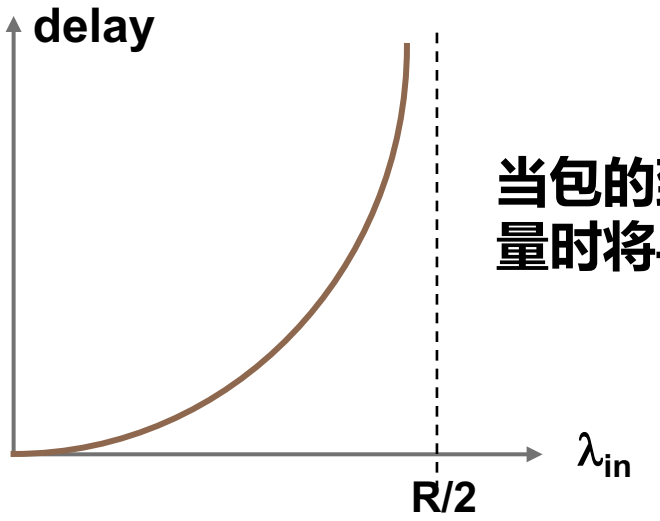
- 每个连接的$\lambda_{out} = \lambda_{in}$

发送端速率大于R/2

- 每个连接的$\lambda_{out} = R/2$
- 路由器的无限长队列将吸收来不及发出去报文



两个连接平分链路的容量R



当包的到达率接近链路容量时将导致长的排队延迟

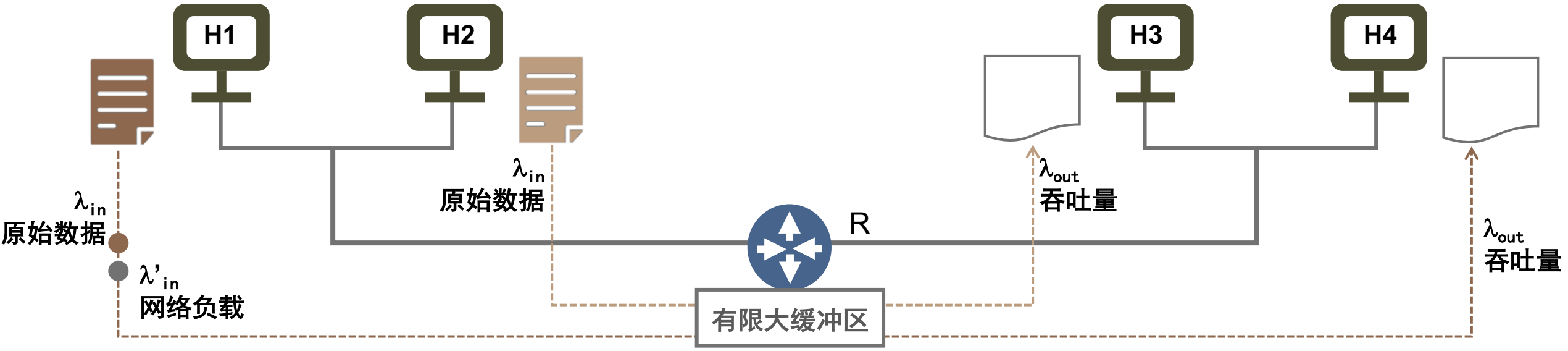
路由器只有有限长队列

假设：如图所示网络中

- 路由器只有有限大缓冲区，出境链路容量为R
- 无错误控制、无流量控制、无壅塞控制

特点

- 带宽利用率最大
- 缓冲区可能溢出造成丢包
- 包排队延迟过大造成超时



λ'_{in} （传输层发到网络的数据）= 原始数据 λ_{in} + 重发的数据（率）

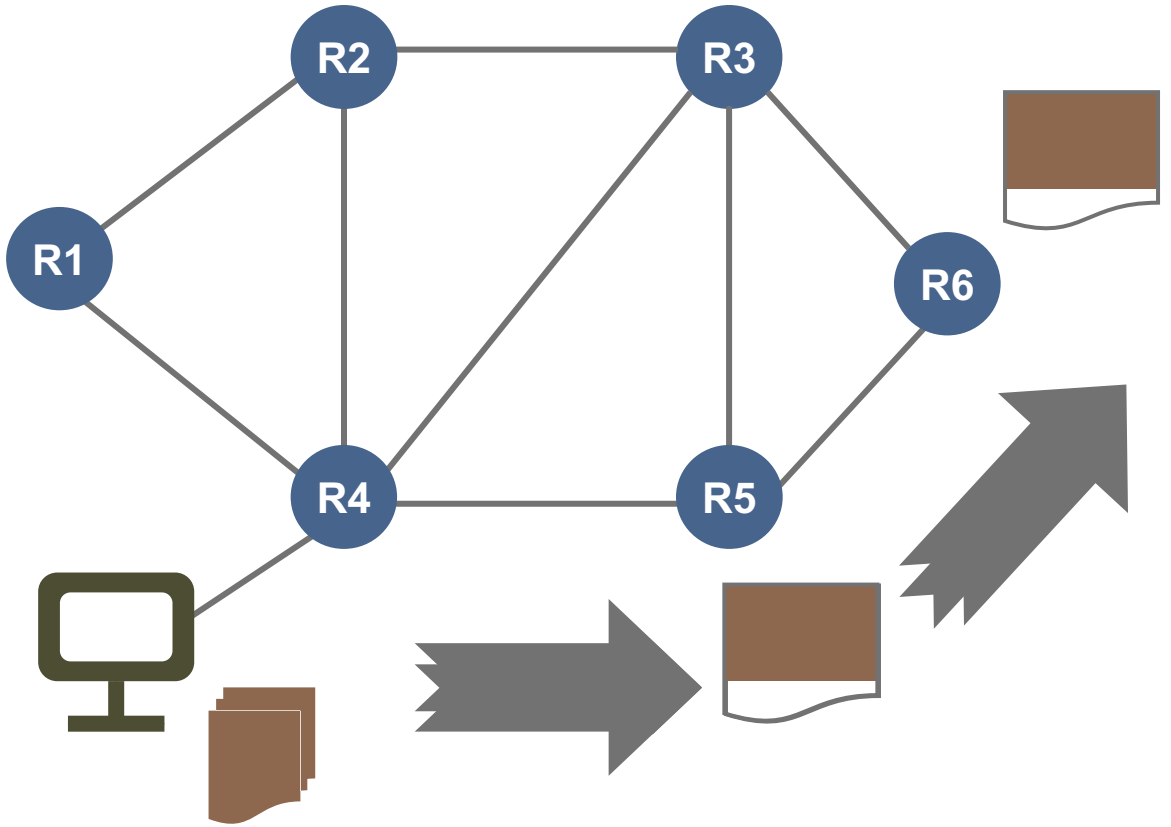
λ_{out}： 连接的吞吐量（接收端的每秒字节数）

局部拥塞的蔓延

假设：如图所示的互联网中,路由器R6上队列将满，要求路由器5放慢发送速度

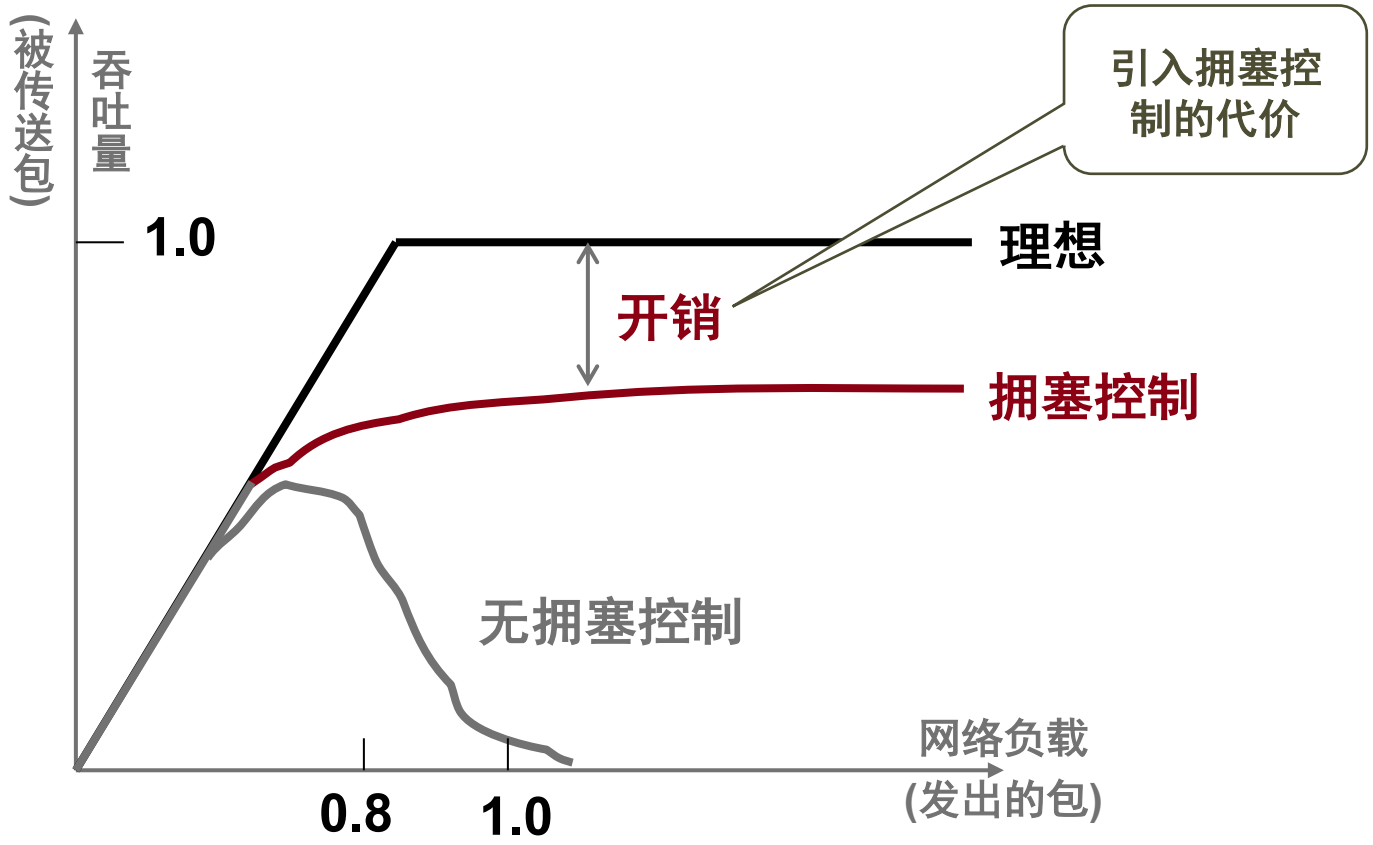
- 网络中某一点(R6)的拥塞将很快波及到一个区域，甚至整个网络。

- 必须以控制整个网络流量的方式来使用流量控制工具。



拥塞控制对网络吞吐量的影响

拥塞控制技术无法达到理论上的理想值。



理想状态

要求所有的站点都能知道提交给网络的包的时间和速率

不加任何控制

当不同节点的队列长度增加时实际吞吐量呈下降趋势

施以拥塞控制

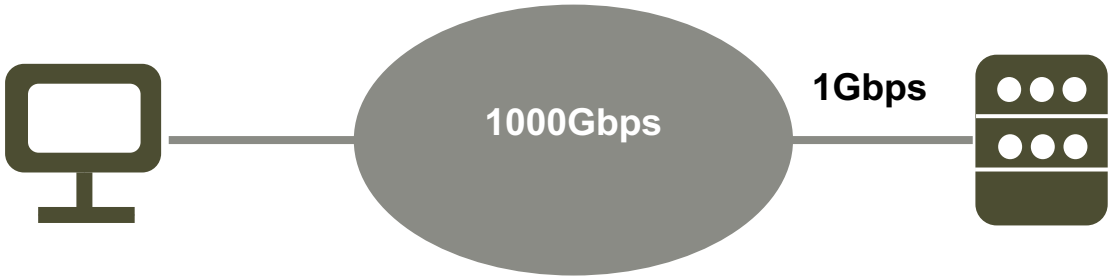
限制每个节点的队列长度以避免吞吐量崩溃。

流量控制与拥塞控制

流控只与发送方和接收方之间的端-端通信有关。

拥塞控制是全局问题：涉及所有主机、路由器及路由器的存储-转发能力。

假设：个人PC机和大型服务器通信



假设：1000台主机同时发送

网络流量通畅，但两端处理能力不一致必须进行流量控制！

收发双方能力相当无需流量控制，但网络容量小，必须进行拥塞控制

拥塞控制之

拥塞控制基本方法

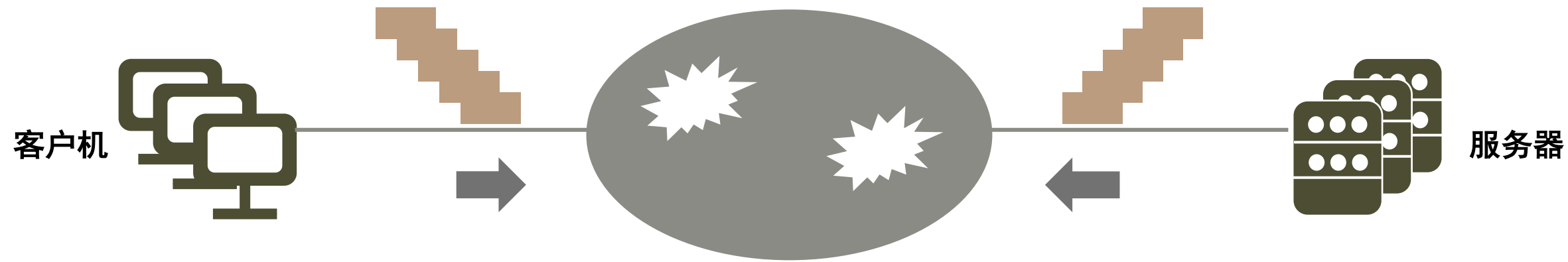
拥塞控制机制分类

端 - 端拥塞控制

- 网络层不提供对拥塞控制显式支持
- 主机必须由网络行为推断拥塞的发生
  - ✓ 包丢失
  - ✓ 延迟增大

网络协助拥塞控制

- 路由器检测拥塞并向发送端反馈信息
  - ✓ 抑制包
  - ✓ 逐跳后压
  - ✓ 显式拥塞通知



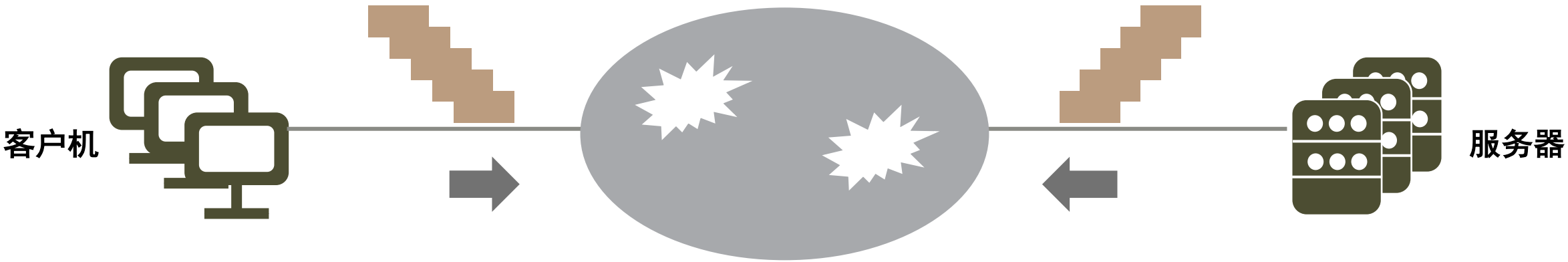
拥塞控制机制的途径

拥塞避免

- 资源预留
- 通信量整形
- 随机早期丢包

拥塞检测

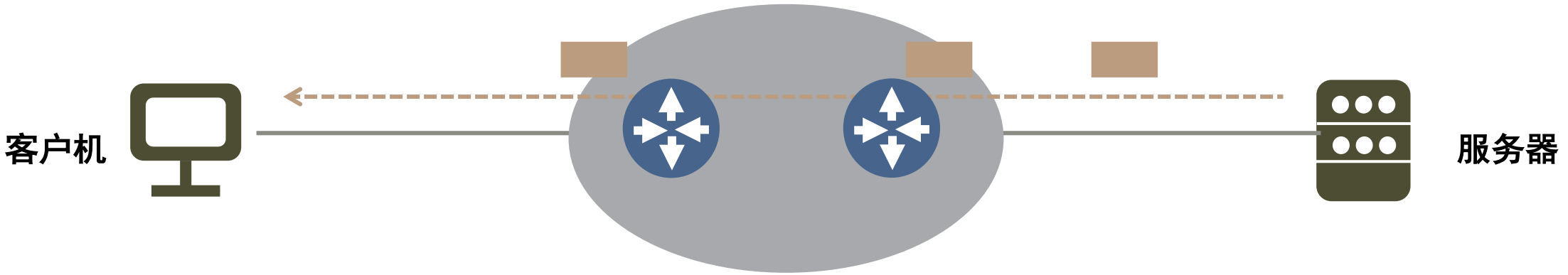
- 显式拥塞通知
- 抑制包
- 卸载



拥塞避免基本思想

拥塞避免的基本思想是通过良好的（协议）设计，尽可能减少网络拥塞发生的概率。

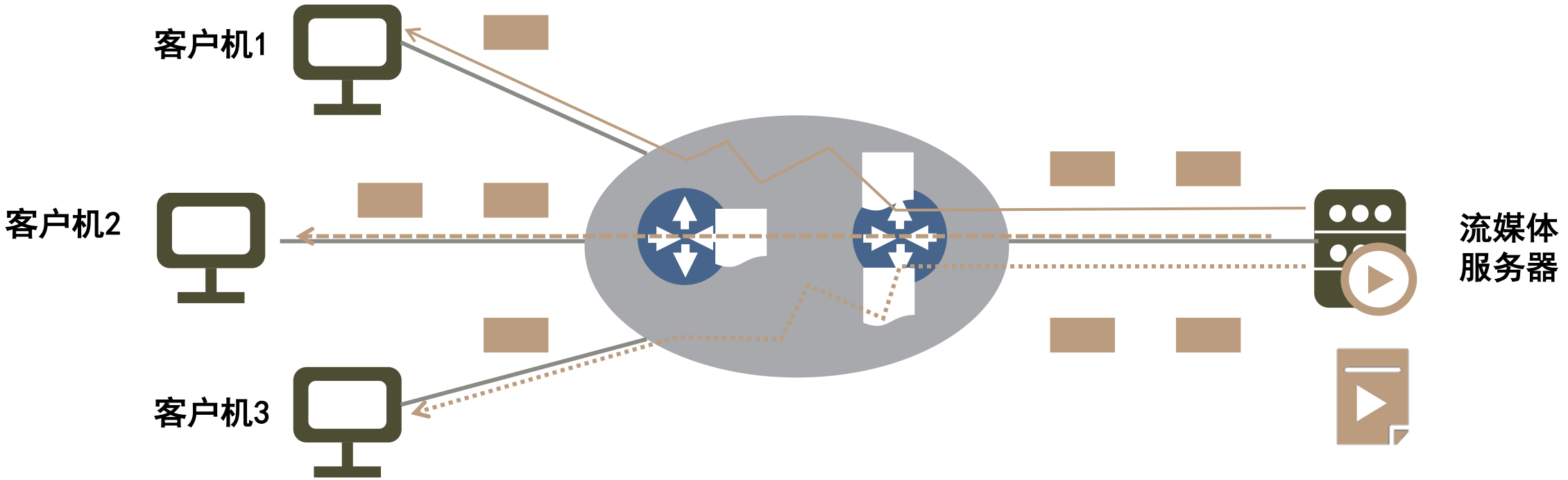
- 发送方调整发送速率尽量不发送多于网络能实际传输的数据
- 面临拥塞即将发生时通过反馈机制通知发送方降低发送速率



拥塞避免之资源预留

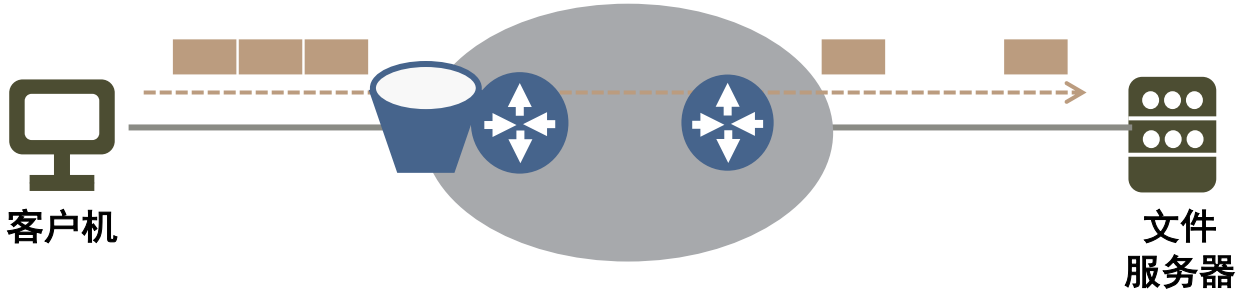
资源预留：数据通信之前网络把通信路径上的资源（存储、带宽）根据应用需要都预留好。

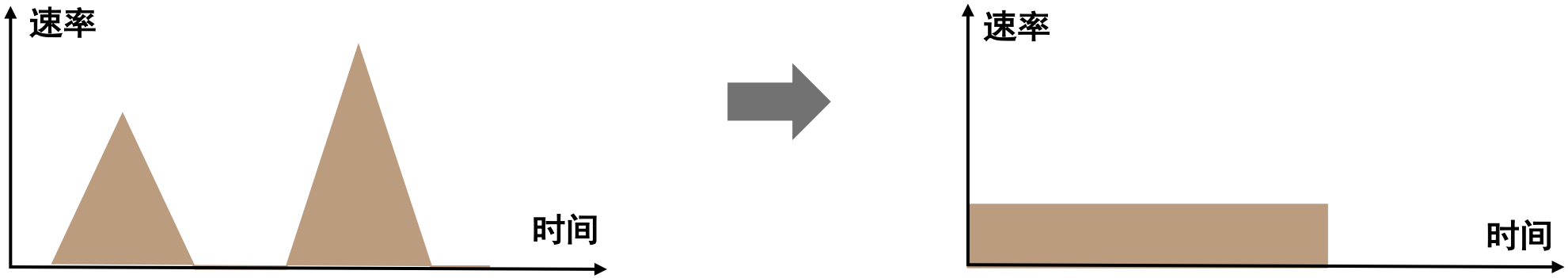
- 数据包在路由器不会堆积
- 数据包在路由器不会被丢弃
- 包的存储-转发时间有限



拥塞避免之通信量整形

通信量整形：通过调节注入网络的数据流的平均速率和突发速率，使得数据流呈现出平稳状态。





拥塞避免之早期丢包

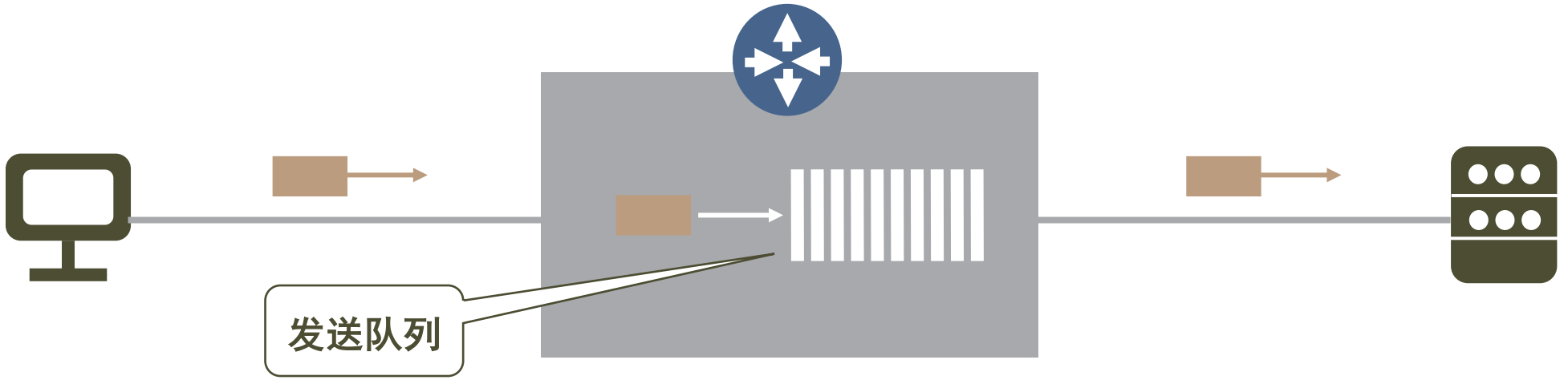
早期丢包：路由器在耗尽缓冲区之前开始丢弃一个或多个数据包。

?

- 何时开始丢包？
- 丢哪些包？

随机早期检测（RED）：当某条链路的队列长度超过某个阈值时随机丢弃一些包。

- 如果平均队列长度低于某个低阈值，将该包排入队列
- 如果平均队列长度大于某个高阈值，则丢弃该包
- 如果平均队列长度介于两个阈值之间，计算拥塞发生的概率

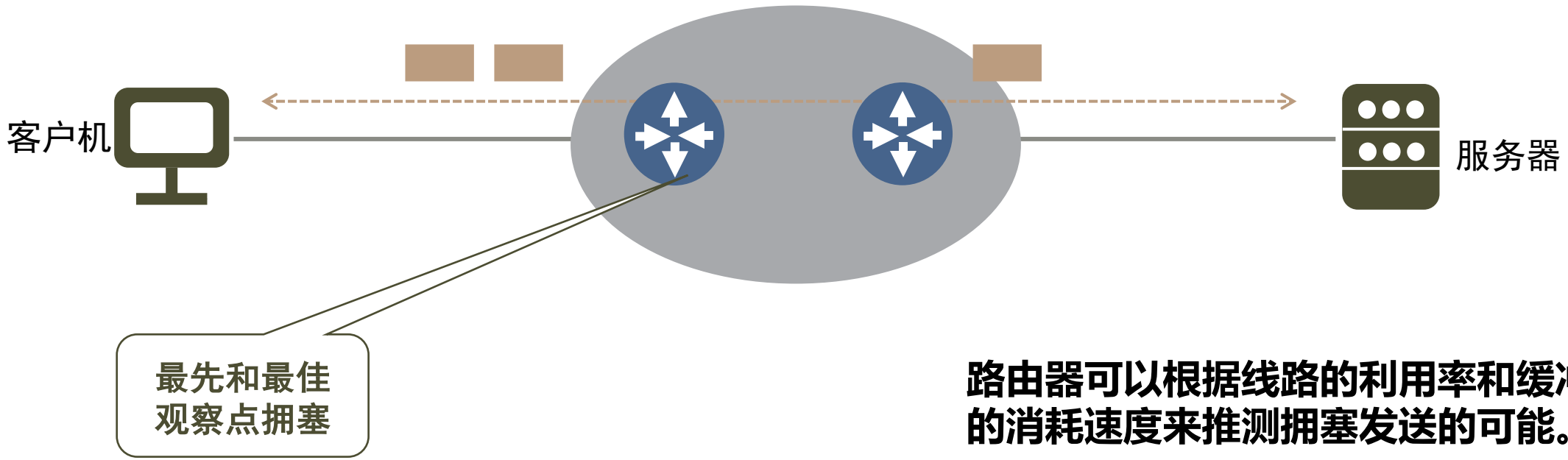


拥塞检测基本思想

拥塞检测：动态监测网络状态，一旦发生拥塞立即采取措施，以便防止拥塞蔓延到网络其他区域。

?

- 谁来检测拥塞？
- 检测到后如何处理？

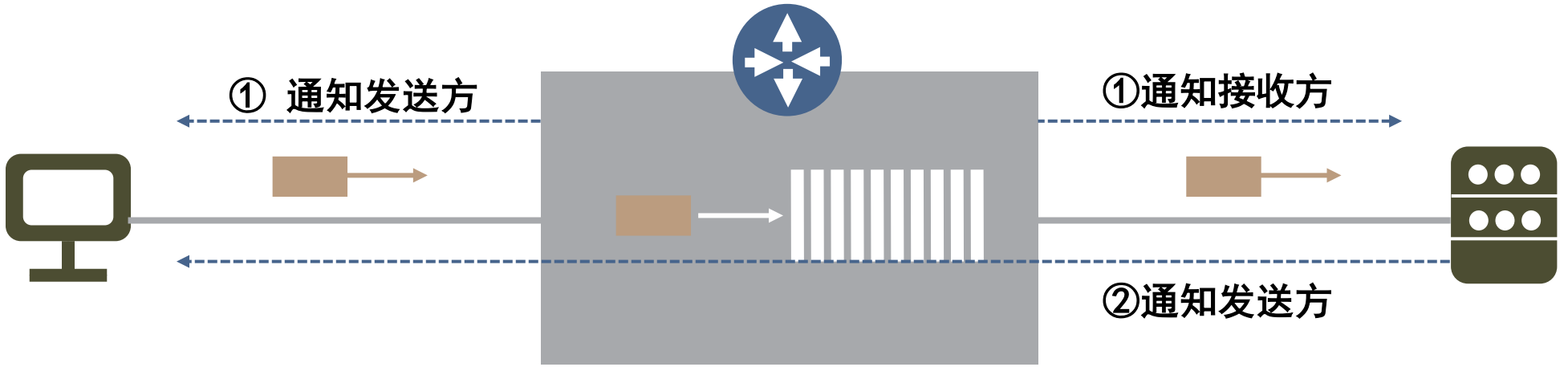


路由器可以根据线路的利用率和缓冲区的消耗速度来推测拥塞发送的可能。

拥塞检测之拥塞通知

拥塞通知：当路由器检测到拥塞即将形成，便通知连接两端，以便发送方降低发送速率减少注入网络流量，达到缓解拥塞的目的。

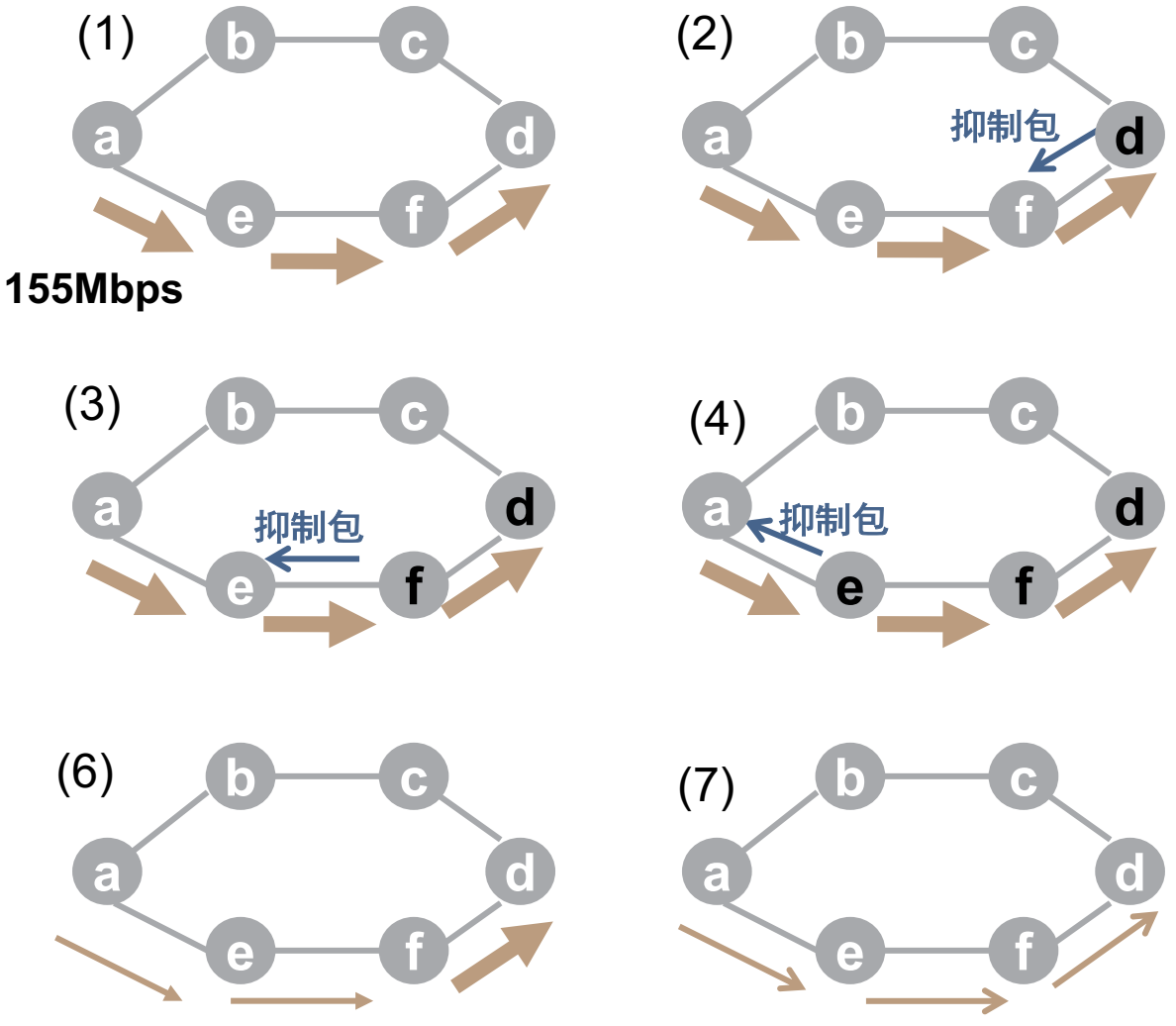
- 路由器随机选择一个包，给包的发送方发送一个拥塞警告包
- 路由器随机选择一个包，在该包打上标记通知接收方，接收方通过确认通知发送方



拥塞检测之抑制包

抑制包：路由器检测到拥塞后给包的发送方返回的具有抑制其发送速率作用的包。

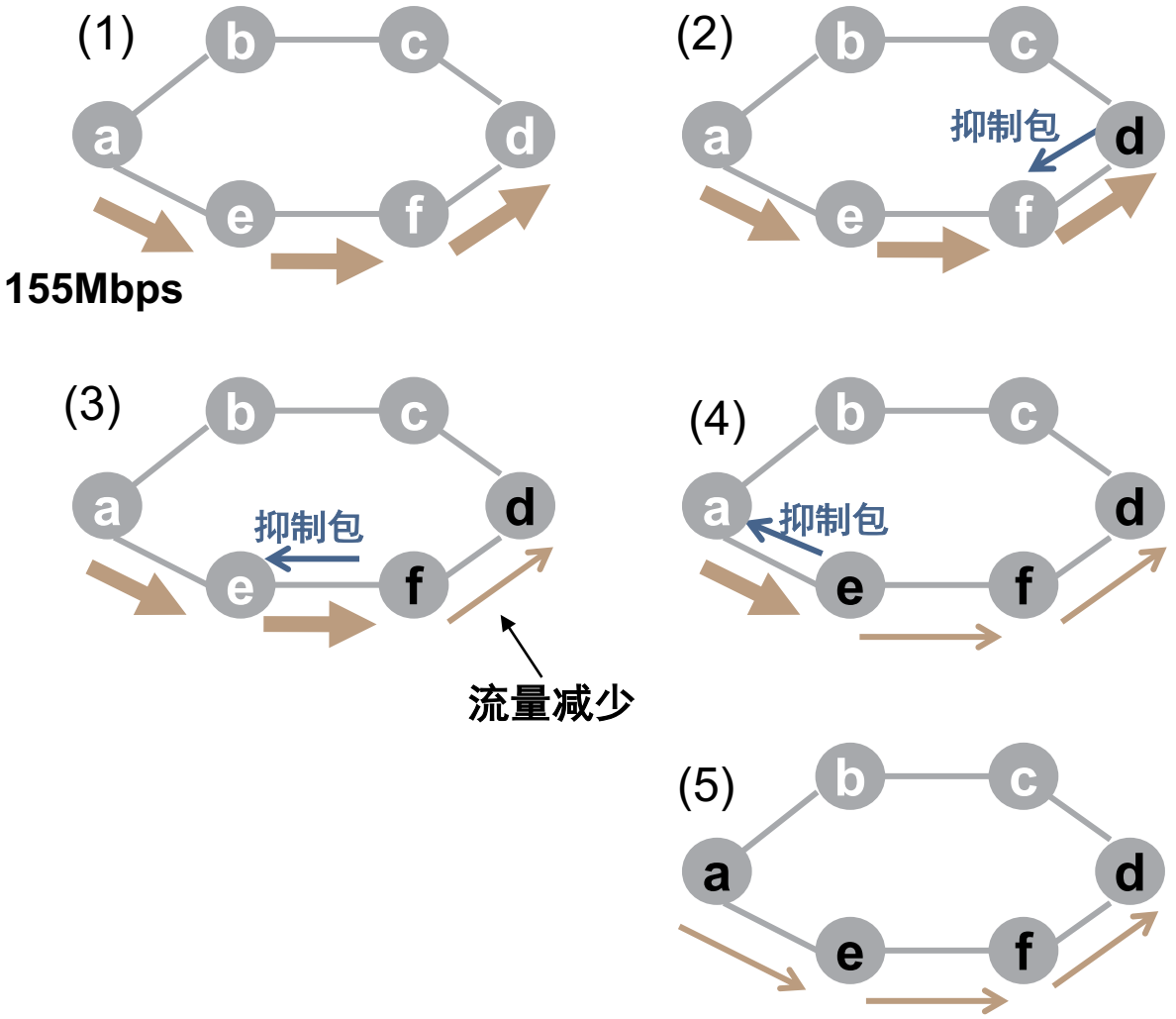
在长距离或者高速率的环境下抑制包对源端作用太慢。



拥塞控制之逐跳抑制包

逐跳（HOP-HOP）抑制包：每一跳都降低传输速率（即使抑制包未到达源端之前）。

转发抑制包的路由器及时做出速率调整对缓解拥塞有很好的作用.



拥塞控制之卸载

卸载：当路由器被包所淹没时只能将包丢弃

- 丢弃哪个包取决于应用
- 应用程序打上丢包优先级标记，在发生拥塞的节点作为是否被丢弃的依据

“葡萄酒”策略

- 陈的比新的香
- 文件传输不能丢弃老的包，否则将造成接收数据不完整

“牛奶”策略

- 新的比旧的鲜
- 实时语音或者视频应用丢弃老的包比丢弃新的更适合