传输层

- 审查了TCP / IP协议参考焦点现在转移到TCP层。

  - 这层是整个协议层次的心脏。

- 它位于应用层是希望通过网络进行通信的应用程序提供了一个“交通”服务之下。

传输层

- TCP提供了可靠的，高性价比的端到端数据传输服务 *独立地* 物理网络（多个）。

- 要认识到，这一点很重要 *服务* 是一个非常不同的 *协议*，虽然它们经常被混淆。

服务

- 参考模型中的每一层提供一组功能，以上面刚刚所述层：
  - 这组功能被称为“服务”。
- 下面被称为“服务提供者”和上面的层的层被称为“服务用户”
- 该服务通过层之间的接口进行访问。

协议

- 在另一方面协议是服务是如何实现的。

- 通常，协议指定了一个框架结构，其中将包括许多包含的控制数据字段组成：

  - 一般这种帧结构被称为协议数据单元（PDU）

  - 例子包括数据链路帧，IP数据报等。

协议

- 该议定书也将通常指定用于解释和响应该PDU中的控制数据的过程。

- 例如，如果一个服务提供数据的可靠传输，则必须存在跟踪的一些装置和从数据丢失中恢复。

协议

- 在PDU控制字段包括编号的这种情况下部分：
  - 例如字节数，帧编号等
- 该协议还将指定在控制字段中的数据是如何被解释，如果有必要回应。
  - 例如，对于丢失帧返回一个REJ消息。

TCP传输服务产品

- TCP传输服务具有以下特点：

  - ***连接方向：*** 两个应用程序之前，实体可以沟通，他们必须先建立连接。

  - ***点至点的通讯方式：*** 每个TCP连接具有完全相同 二 端点。

  - ***完整的可靠性：*** TCP保证数据将被传递准确的发送，即无数据丢失或失序的

  - ***全双工通信：*** TCP连接允许数据在任一方向流动

    - TCP缓冲器传出和传入数据
    - 这允许应用程序继续执行其他代码而数据正在被传输

TCP传输服务产品

- ***流接口：*** 该 资源 应用程序发送 *连续* 跨越连接的八位位组序列
  - 数据传递 *整块* 到TCP交付
  - TCP并不能保证在同样大小的块，它是由源应用程序提供传输数据。

- ***可靠的连接启动：*** TCP两种应用 同意 任何新的连接

- ***优美的连接关闭：*** 这两者都可以请求连接被关闭
  - TCP保证关闭连接之前，可靠地交付所有数据

传输服务

- TCP传输服务是提供给一个 *用户进程* 存在的应用层内：

  - 这个 *用户进程* 被认为是“交通运输服务用户”，

  - 这项服务是通过一组通常提供
    原语 跨层之间的界面，

  - 调用这些原语使运输服务提供商执行一些动作。

传输实体

- 为了实现该服务将作为被称为传输层内有助于未来的讨论TCP软件 *传输实体：*

  - 这是“交通运输服务提供商”。

传输实体

- 该 *传输实体* 可以位于任何数量的地方，包括：
  - 在操作系统（OS）内核，
  - 作为一个单独的用户处理中，
  - 在绑定到网络应用程序库软件包，
  - 在网络接口卡。

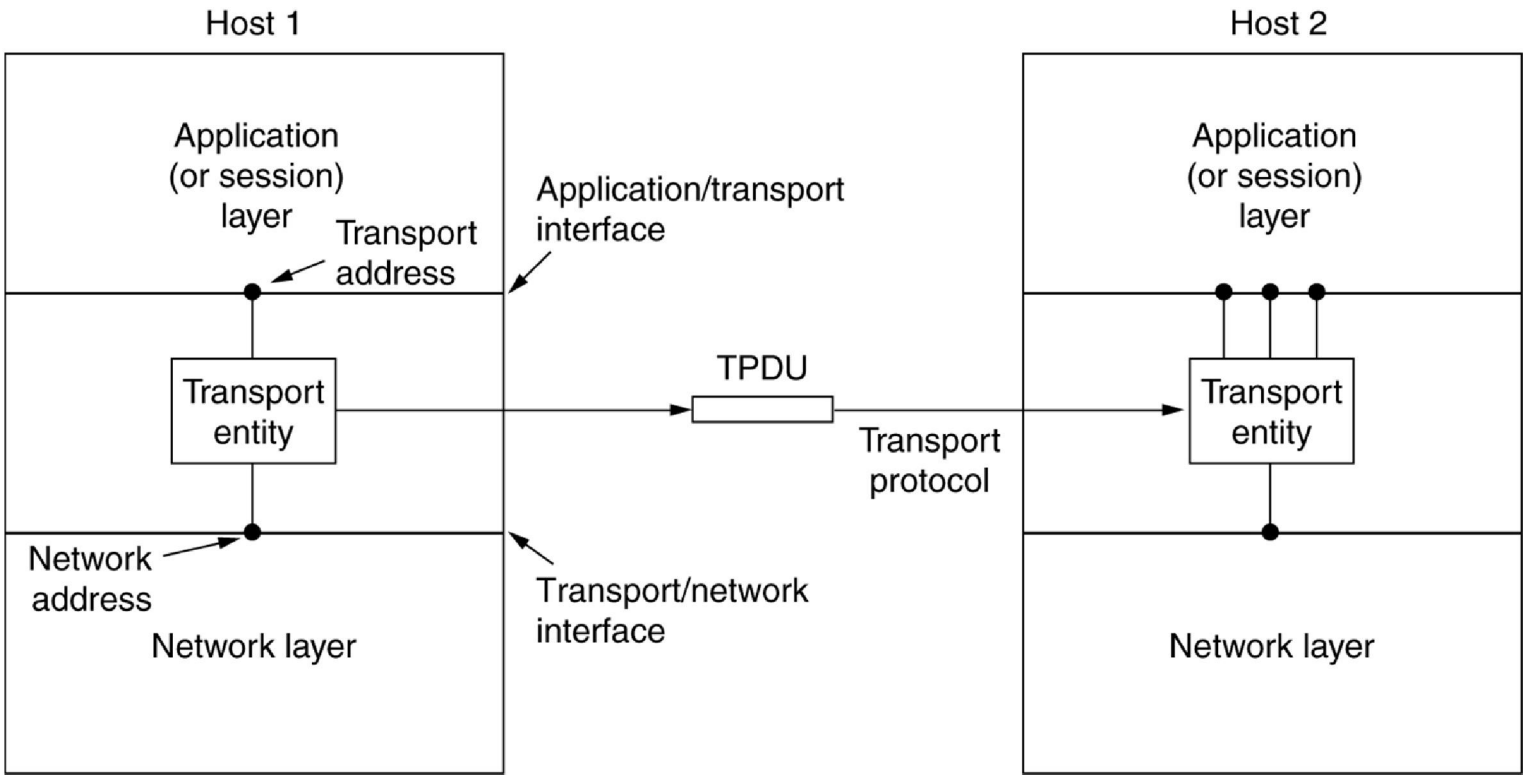
传输实体

- 如果 *协议栈* 位于OS中的 **原语** 被实现为 *系统调用：*
  - 这些调用打开机器的控制权交给OS发送和接受必要的PDU。
- 接下来的图显示了在上下文中的运输企业。
- 可以看出它显示了传输层应用层和网络层之间坐着。

传输实体

- 它具有每一个之上和之下的层中的一个连接：
  - 这是“服务提供者”向应用层，
  - A“服务用户”的网络层。
- 该TPDU表示被之间交换的帧结构 *对等实体：*
  - 该PDU不动 水平 传输层之间，
  - 相反，它移动 上和下 通过协议栈。

网络，传输层和应用层

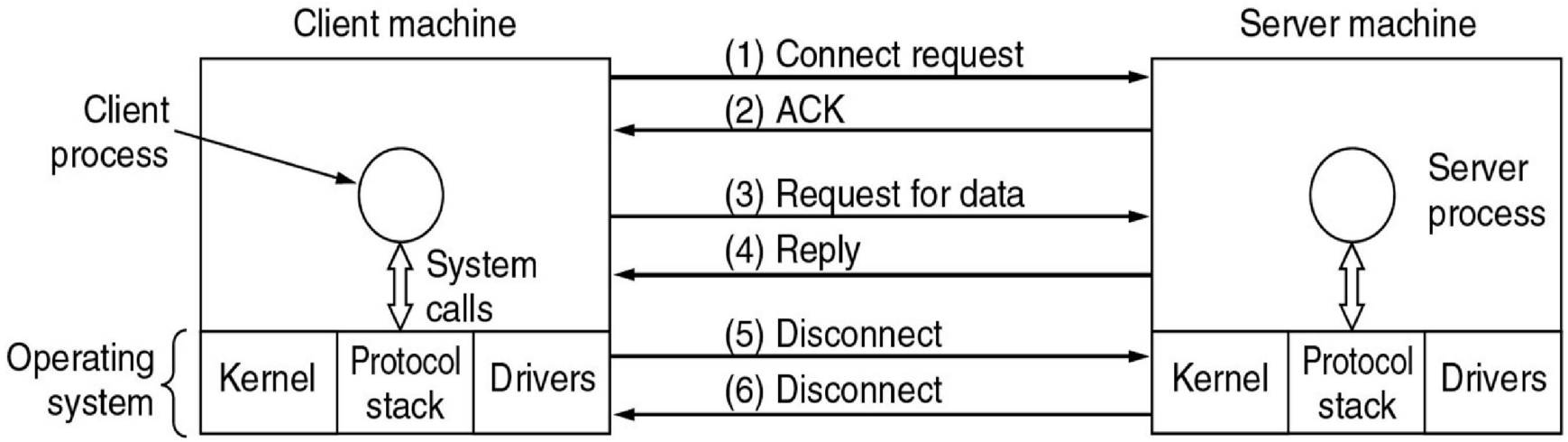

Host 1
Host 2
Application
(or session)
layer
Transport
address
Application/transport
interface
Application
(or session)
layer
TPDU
Transport
entity
Transport
protocol
Transport
entity
Network
address
Network layer
Transport/network
interface
Network layer

运输服务原语

- 做这些原语是什么样子？
- *考虑 **通用** 传*输接口如所示的下一个图中：
  - 在这里可以看到基元的列表。
  - 这些原语允许应用程序的建立，使用和释放连接。
- 通常有调用这些原语的一个非常精确的顺序：
  - 确切的序列不同为 *客户* 和 *服务器。*

一组基本的运输服务原语

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection



管理连接

- 在解释原始的调用序列它了解是很重要的 *沟通的阶段。*
- 从HDLC通常有一个面向连接的服务相关的通信的三个阶段以前的讨论回想一下：
  - 连接建立，
  - 数据传输，并且，
  - 连接释放。
- 在每个阶段的各种PDU的是本之间交换 *客户* 和 *服务器*：
  - 每个PDU包含用于“对方”的消息。

每相连接的交互

- *连接建立阶段：*
  - 该 *服务器* 首先 执行LISTEN原语：
    - 服务器的 *传输实体* 响应由该原始通话 闭塞 服务器，直到客户端请求到达时，
  - 客户端的 然后 执行CONNECT电话：
    - 这将导致连接请求TPDU被发送到服务器，
  - 服务器传输实体放开从服务器返回接受TPDU到客户端的连接，
  - 客户端传输实体然后放开从客户端和连接被认为是 **既定** 。

每相连接的交互

- *数据传输阶段：*
  - 与 活性 连接数据现在可以在客户端和服务器之间使用SEND和交换基元RECEIVE，
  - 每一侧必须使用（粘连）接收和发送轮流。
- *发布阶段：*
  - 无论是客户端或服务器可以调用断开原始：
    - 此发送一个DISCONNECT TPDU到远程传输实体
    - 抵达后，该连接被认为是 **释放。**

伯克利套接字

- 上面讨论的基本传输原语是 不 标准化。

- 相反，大多数操作系统的设计师都采用了

  插座元

  - 这些源自加州的UNIX操作系统的伯克利大学其中载 *TCP / IP*

    的网际协议套件

- 因此套接字API已经成为

  事实上的 标准用于连接到TCP / IP

套接字通信和UNIX

- 从UNIX背景的 *插座* 使用UNIX中的许多概念
- 应用程序通过通信 *插座* 以同样的方式，它传输数据或从文件
- 对于这个UNIX使用的 *开放式读写关闭* 范例
  - 应用程序调用：
    - *打开* 准备为读/写文件
    - *读* 要么 *写* 检索或从文件发送数据/
    - *关* 完成使用文件
- 什么时候 *打开* 首先叫 *描述* 返回
  - 今后所有与文件交互需要此 *描述*
  - socket通信也使用了这种描述法

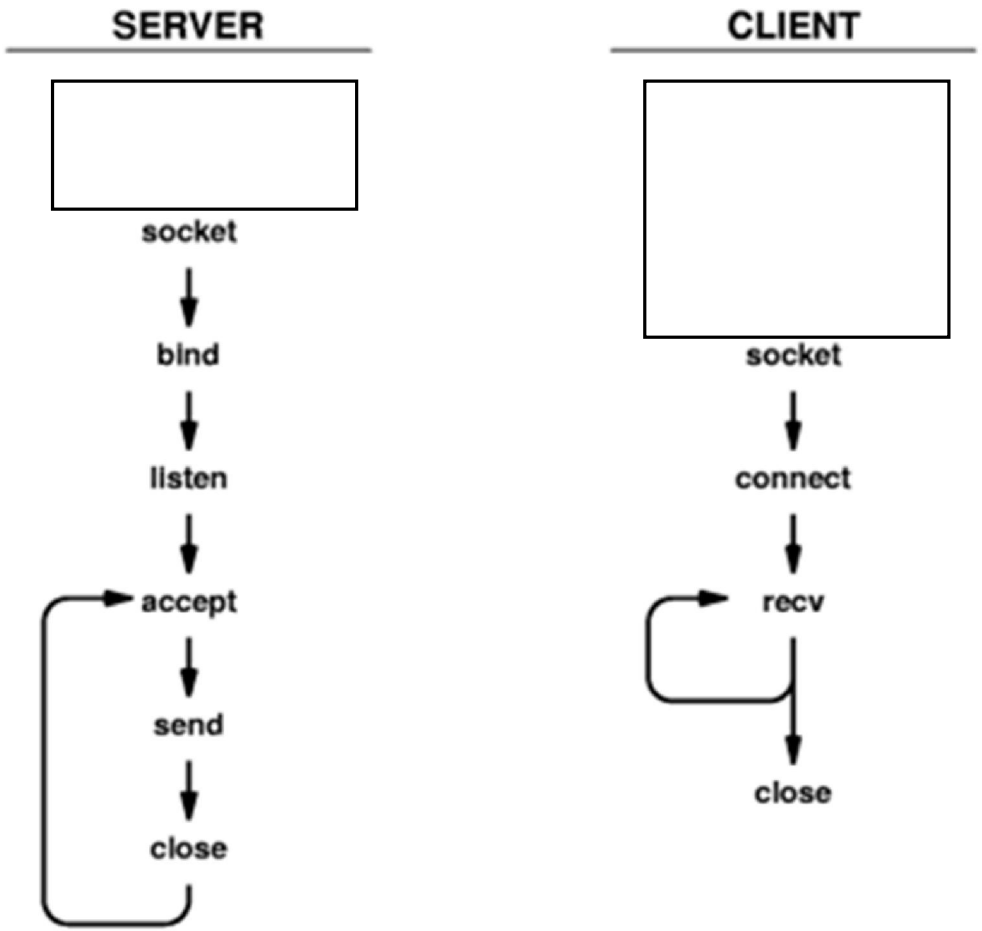
套接字通信和UNIX

- 使用TCP / IP协议进行通信的应用程序必须请求OS以创建 ***插座*** ：
  - 这是一个抽象的概念，其将在后面详细进行说明。
- 操作系统返回一个描述符 *识别* 插座：
  - 现在只是把它作为一个 *文件句柄。*
- 与文件I / O描述符必须在所有未来的交互使用的插座
- 下面的幻灯片列出的Socket API原语和示出了如何将这些原语由客户端和服务器应用程序使用的示例

插座传输基元

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
D	Send some data over the connection
READ /接收写入/ SEN	Receive some data from the connection
CLOSE	Release the connection

示例客户端 - 服务器交互使用 *套接字*



插座基元 - 解释

- 下面的图元被执行

  服务器：

  - SOCKET：该原语创建一个新的 *终点*

    传输实体内：

    - 表空间传输实体中分配，
    - 文件描述符返回它在未来的所有呼叫使用
  - BIND：这个原始套接字绑定到一个网络地址：

    - 这允许远程客户端连接到它

插座基元 - 解释

- LISTEN：这个原始分配的传入呼叫请求传输实体内的排队空间。

- ACCEPT：服务器等待传入连接该原语块：

  - 一旦接收到连接请求的 *传输实体*

    创建 新 插座等同于原来的一个，并返回一个文件描述符到服务器。

  - 服务器 *叉关闭* 一个新的进程或 *服务线程* 处理 连接 在新的socket

  - 服务器 也 继续等待原始套接字的更多的连接

插座基元 - 解释

- 下面的图元被执行 *客户：*
  - SOCKET：如前
  - CONNECT：这个原始块的客户端，并积极启动连接过程

- 下面的图元被执行 *客户* 和

  服务器：
  - SEND和RECV：这些原语被用于发送，并通过全双工连接接收数据
  - CLOSE：这个原始版本的传输连接

插座和插座库

- 在大多数系统中，*插座* 功能是操作系统的一部分。
- 然而，有些系统，需要一个 *套接字库* 提供接口给 *传输实体：*
  - 这些不同的操作到 *本地人* 套接字API，
  - 该库插槽程序代码被链接到应用程序，并驻留在它的地址空间，
  - 调用一个套接字库通控制，而不是操作系统的库例程。
- 这两种实现提供从程序员的角度相同的语义
- 使用任何一种实现可应用 **移植**
  到其他计算机系统。

实施例使用的插座

- 下一幻灯片显示了使用套接字API的示例客户端应用程序。
- 它是连接到一个daytime服务器的当前日期和时间上的服务器主机简单的daytime客户端。
- 这个应用程序是用“C”。
- 线24，41，44和57示出了插座基元4被调用。
- 这一计划将在课堂上解释，你将有机会编译并运行它针对预编译的服务器。

一个例子客户端使用 *套接字*

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "Practical.h"

int main(int argc, char *argv[]) {
    char recvbuffer[BUFSIZE]; // I/O buffer
    int numBytes = 0;

    if (argc < 3) // Test for correct number of arguments
    DieWithUserMessage("Parameter(s)",
        "<Server Address> <Server Port>");

    char *servIP = argv[1];     // First arg: server IP address (dotted quad)

    in_port_t servPort = atoi(argv[2]);

    // Create a reliable, stream socket using TCP
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock < 0)
        DieWithSystemMessage("socket() failed");

    // Construct the server address structure
    struct sockaddr_in servAddr;            // Server address
    memset(&servAddr, 0, sizeof(servAddr)); // Zero out structure
    servAddr.sin_family = AF_INET;          // IPv4 address family
    // Convert address
    int rtnVal = inet_pton(AF_INET, servIP, &servAddr.sin_addr.s_addr);
```

一个例子客户端使用 *套接字*

```
34      if (rtnVal == 0)
35          DieWithUserMessage("inet_pton() failed", "invalid address string");
36      else if (rtnVal < 0)
37          DieWithSystemMessage("inet_pton() failed");
38      servAddr.sin_port = htons(servPort);     // Server port
39
40    // Establish the connection to the echo server
41      if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
42          DieWithSystemMessage("connect() failed");
43
44      while ((numBytes = recv(sock, recvbuffer, BUFSIZE - 1, 0)) > 0) {
45          recvbuffer[numBytes] = '\0';     // Terminate the string!
46          fputs(recvbuffer, stdout);       // Print the echo buffer
47              /* Receive up to the buffer size (minus 1 to leave space for
48                  a null terminator) bytes from the sender */
49          }
50      if (numBytes < 0)
51        DieWithSystemMessage("recv() failed");
52          //    else if (numBytes == 0)
53          //    DieWithUserMessage("recv()", "connection closed prematurely");
54
55      fputc('\n', stdout); // Print a final linefeed
56
57      close(sock);
58      exit(0);
59  }
60
```