火龙果软件学院

TCP IP高级编程

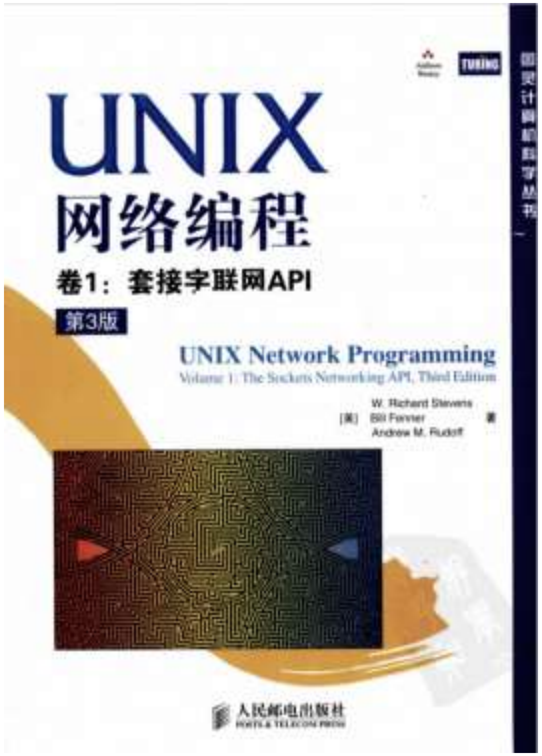
Evolve by case

培训目标

- 支持更好地Teamwork
  - 整个团队象一个人一样协调，一样灵活

课程内容安排概览

- 0.5天-TCP/IP与套接字编程基础
- 0.5天-深入套接字编程
- 0.5天-UDP与SCTP编程
- 0.5天-域名相关编程



环境准备

操作系统

- Unix/Linux操作系统
- 建议使用Ubuntu 16.04 LTS
- 可以使用服务器或虚拟机
  - 如果使用虚拟机，需要增加一个Host-only网卡，以便传递文件 https://my.oschina.net/wiselyming/blog/177153

环境安装

- 安装配置OpenSSH，以便从本机传递文件
  - https://blog.ansheng.me/article/ubuntu-install-configuration-ssh.html
- 一个文本编辑器（VIM、EMACS或其他）
- 配置安装镜像（国外镜像比较慢）
  - Ubuntu 16.04配置方法：http://blog.csdn.net/Hehailiang_Dream/article/details/54094634
- 安装gcc等编译环境
  - ubuntu可以使用如下命令：
    sudo apt install build-essential

测试环境配置

- 将代码hello-echo.c用sctp/ssh传递到linux环境
- 执行编译：

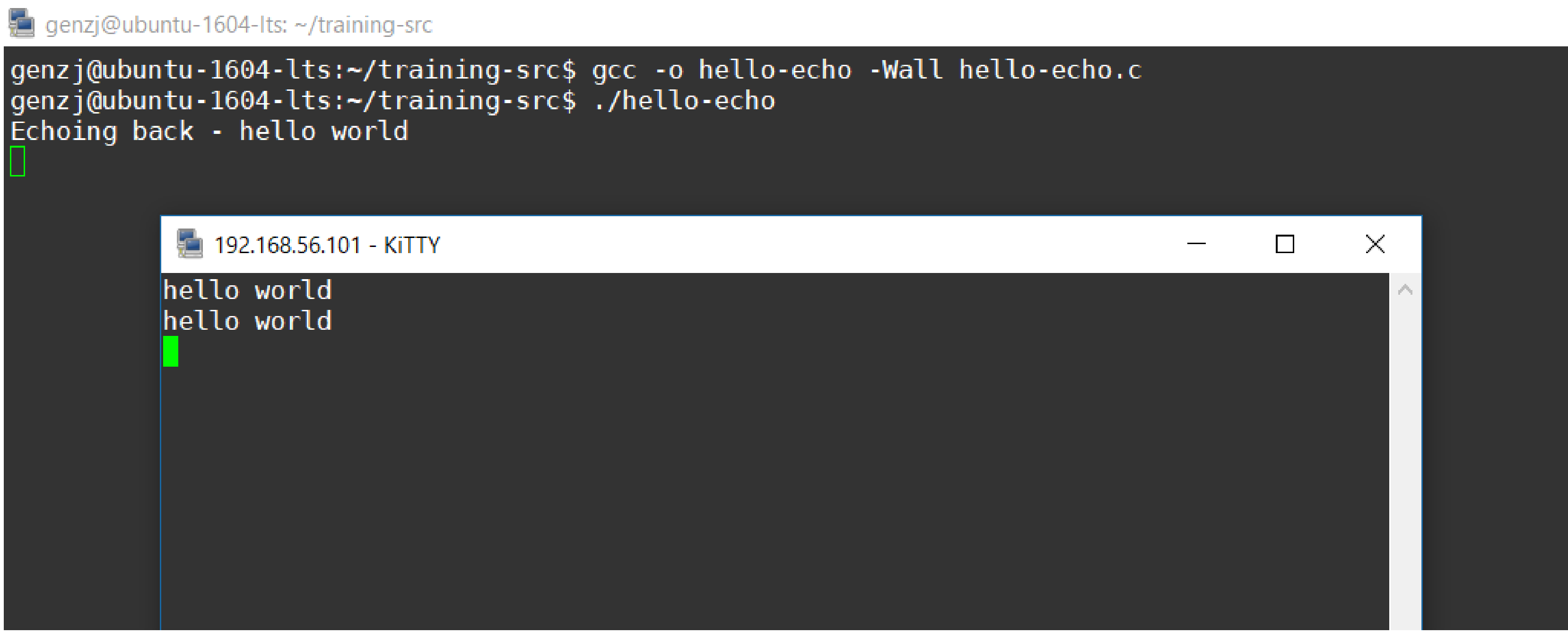
  gcc -o hello-echo hello-echo.c

  如果正确运行，不会有任何输出
- 运行：

  ./hello-echo

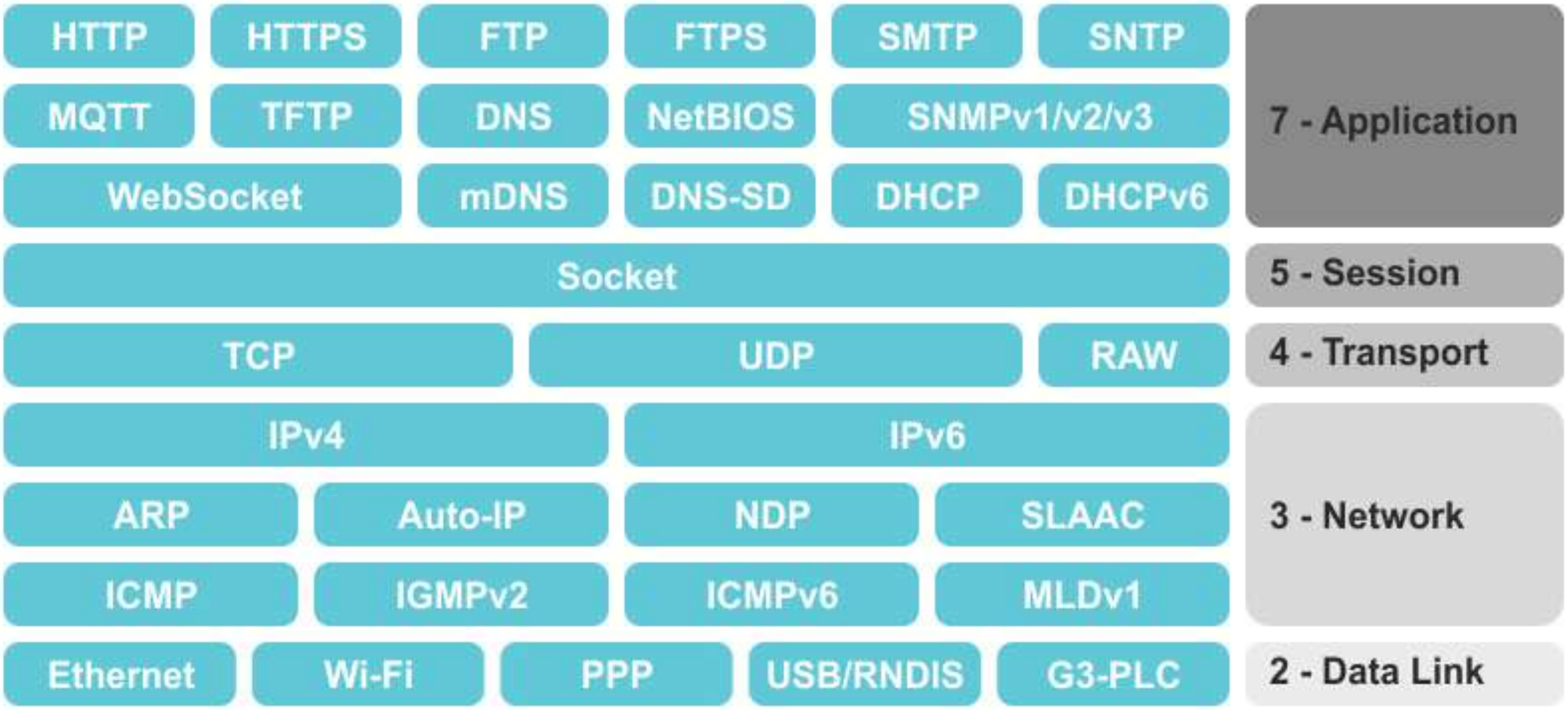
  如果正确运行，不会有任何输出，但也不会出现Shell提示符
- 此时用telnet等程序连接到Linux环境22000端口，输入任意字符，回车后可以看到回复，即为环境配置成功

环境测试截图


genzj@ubuntu-1604-lts: ~/training-src
genzj@ubuntu-1604-lts:~/training-src$ gcc -o hello-echo -Wall hello-echo.c
genzj@ubuntu-1604-lts:~/training-src$ ./hello-echo
Echoing back - hello world
192.168.56.101 - KiTTY
hello world
hello world

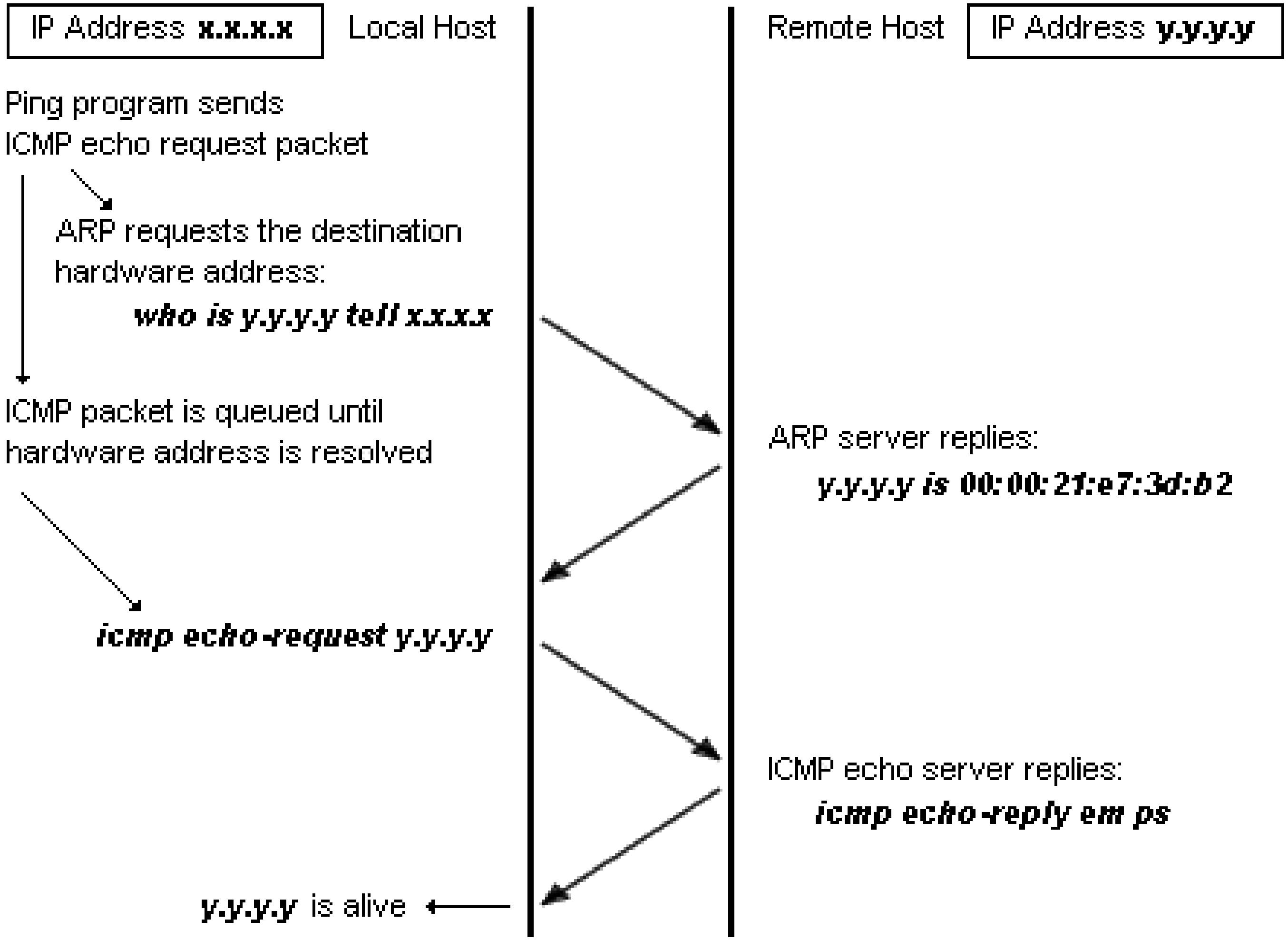
TCP/IP协议族简介

纵观TCP/IP协议栈

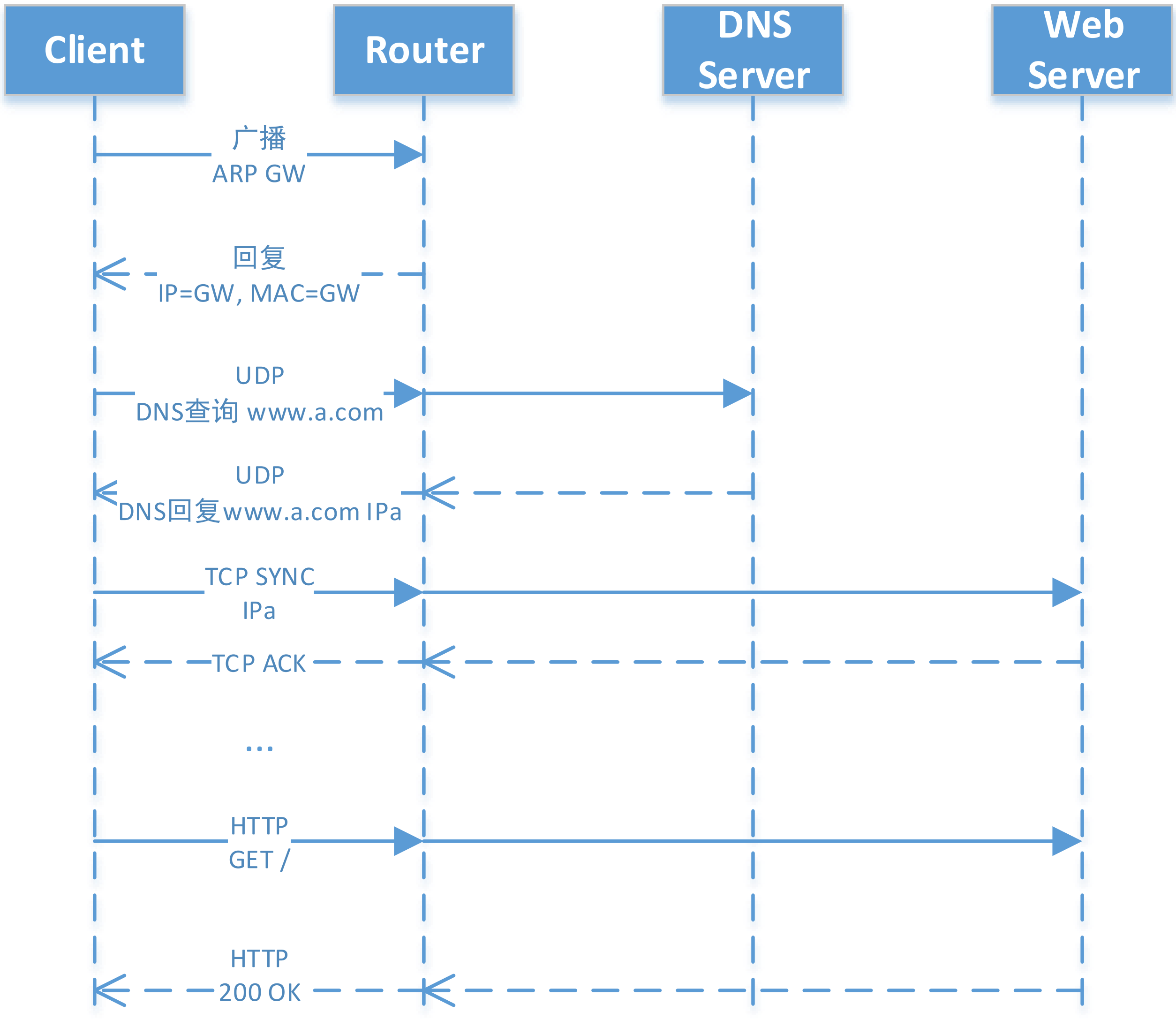


HTTP
HTTPS
FTP
FTPS
SMTP
SNTP
MQTT
TFTP
DNS
NetBIOS
SNMPv1/v2/v3
WebSocket
mDNS
DNS-SD
DHCP
DHCPv6
7 - Application
Socket
5 - Session
TCP
UDP
RAW
4 - Transport
IPv4
IPv6
ARP
Auto-IP
NDP
SLAAC
ICMP
IGMPv2
ICMPv6
MLDv1
3 - Network
Ethernet
Wi-Fi
PPP
USB/RNDIS
G3-PLC
2 - Data Link

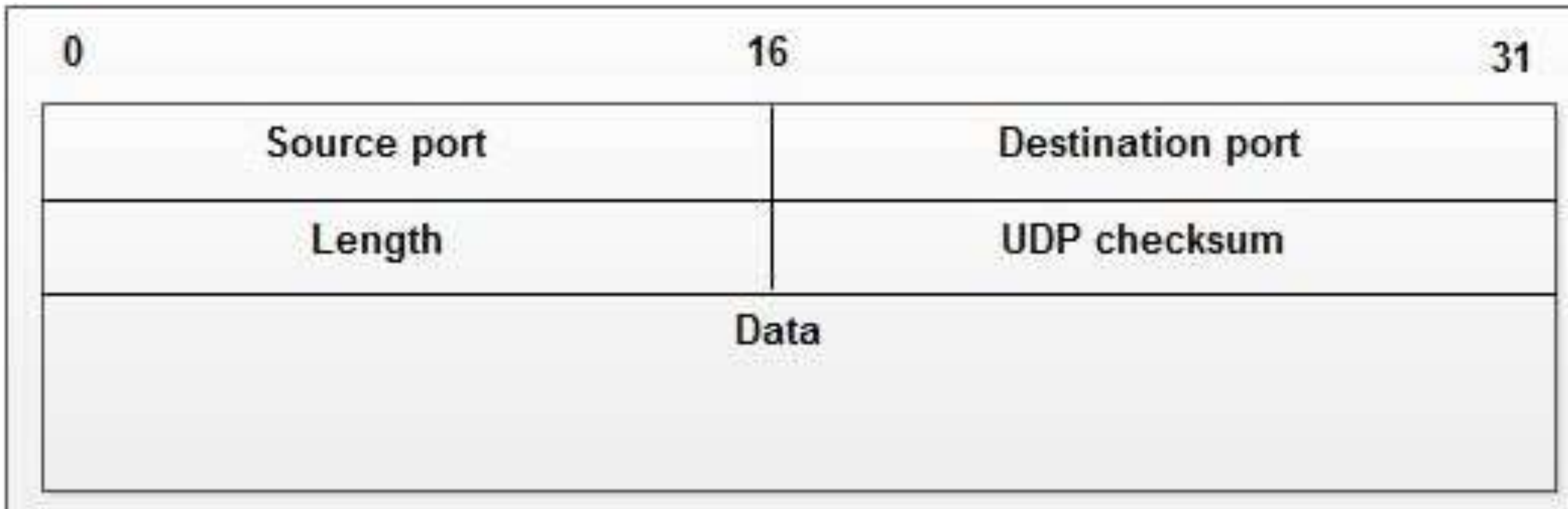
实例分析——ping局域网主机



实例分析——访问网页

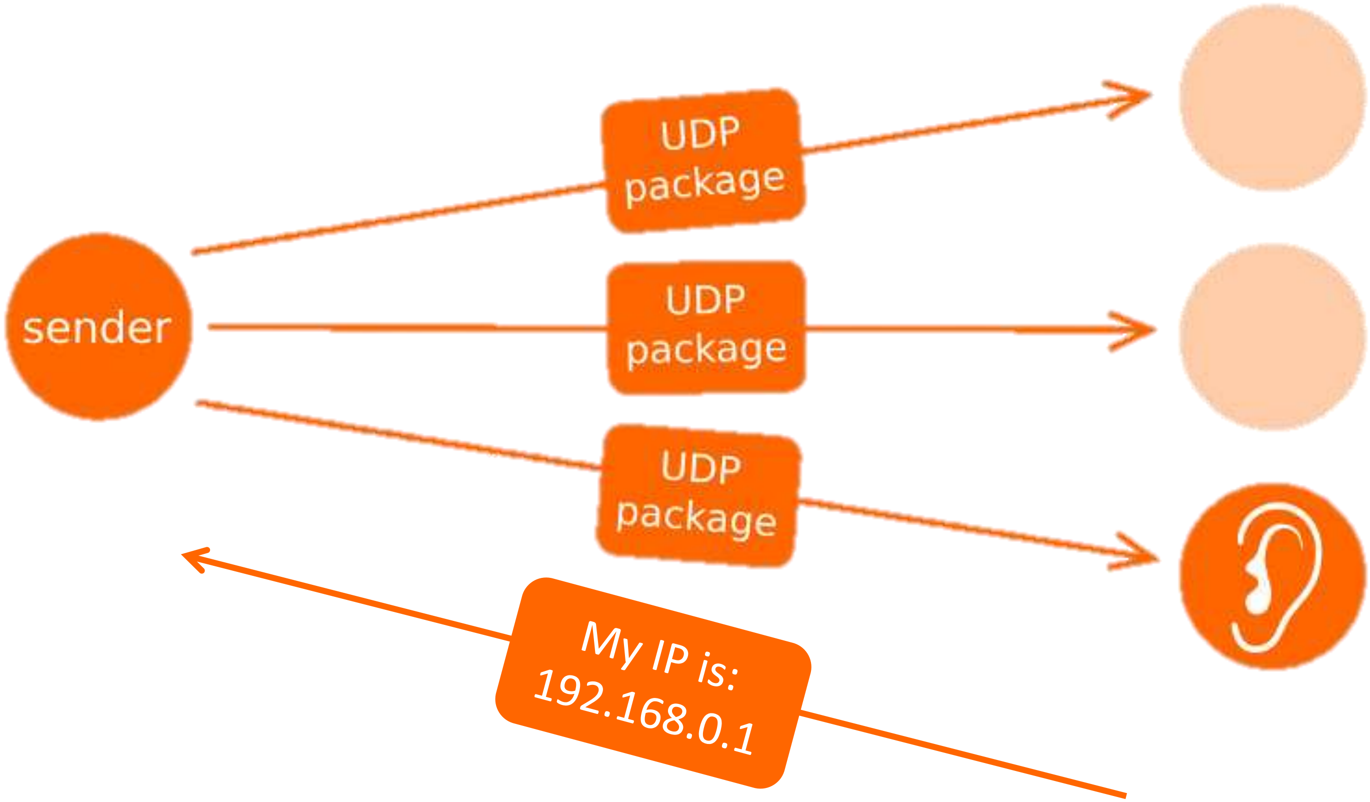

Client
Router
DNS Server
Web Server
广播
ARP GW
回复
IP=GW, MAC=GW
UDP
DNS查询 www.a.com
UDP
DNS回复www.a.com IPa
TCP SYNC
IPa
TCP ACK
...
HTTP
GET /
HTTP
200 OK
....

用户数据报协议（UDP）简介

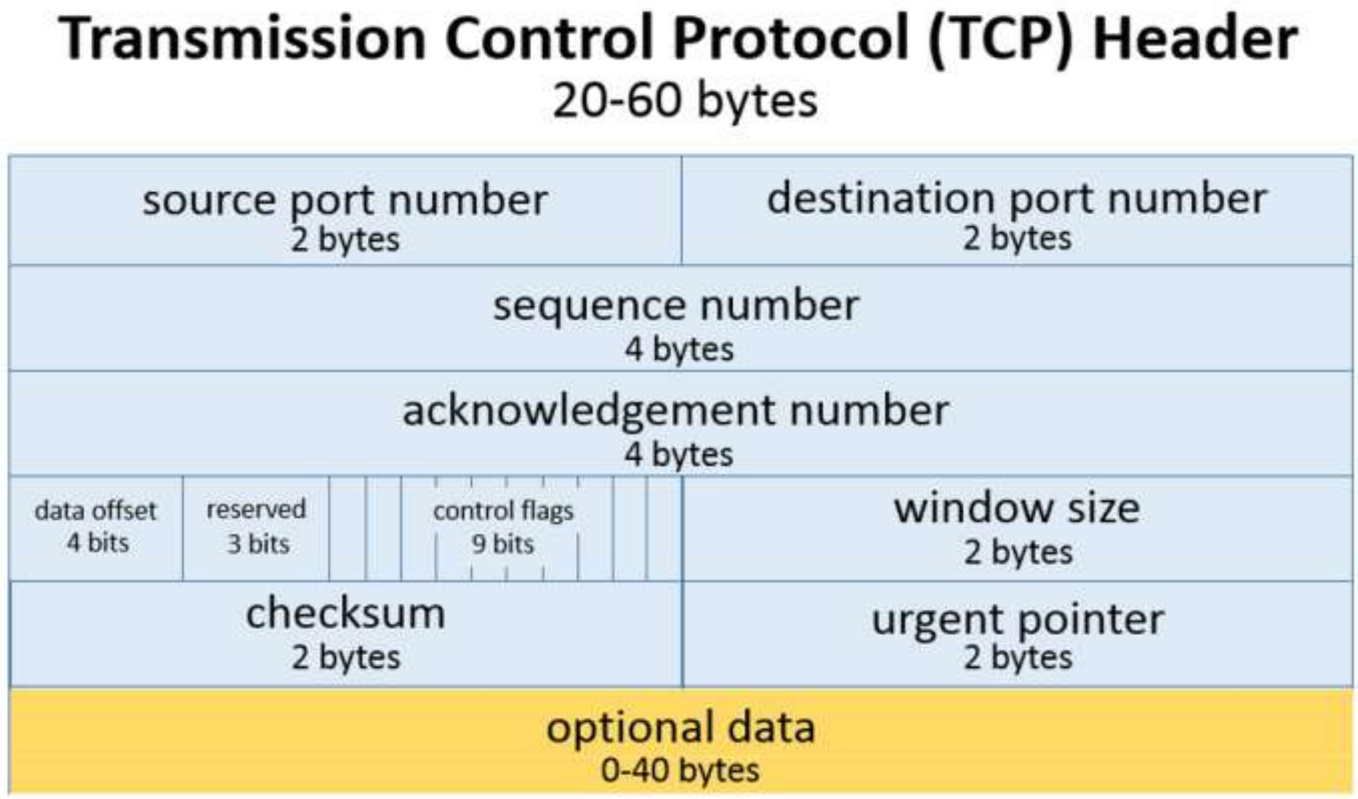


Format of the UDP datagram

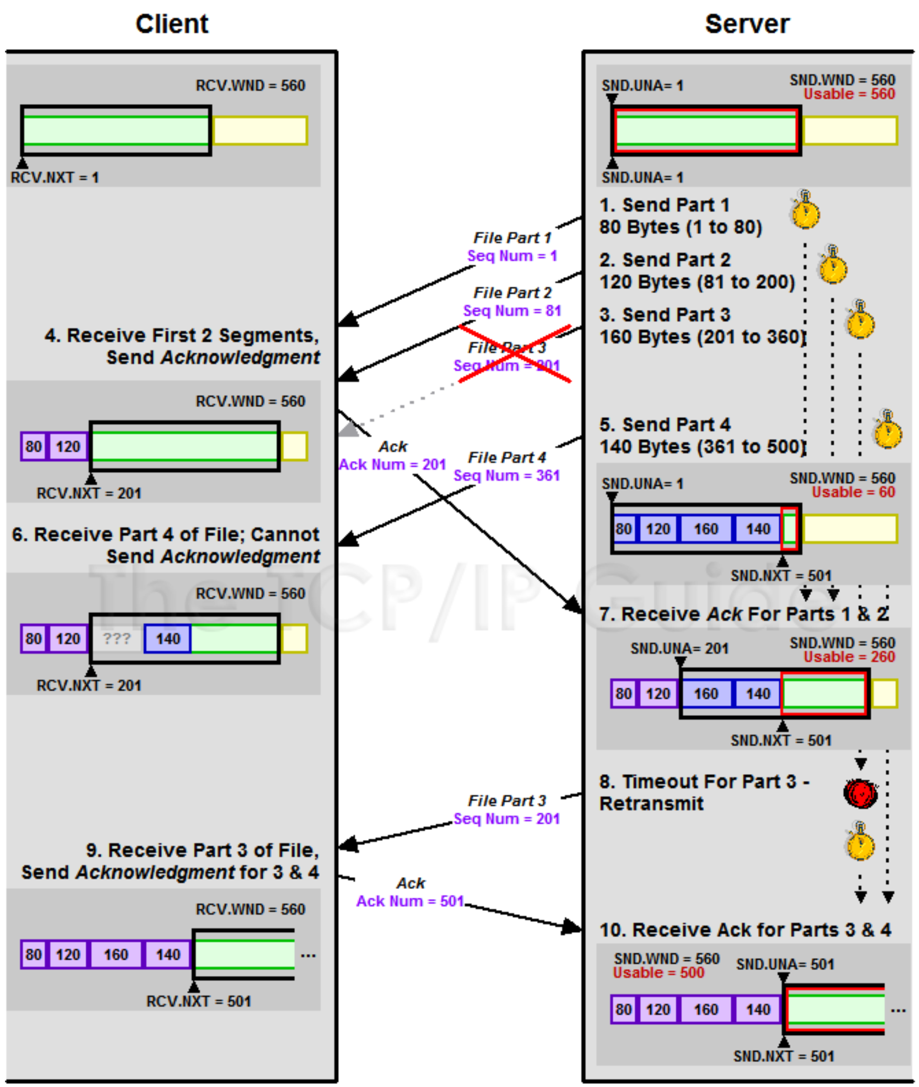
实例分析——设备发现


UDP
package
sender
UDP
package
UDP
package
My IP is:
192.168.0.1

传输控制协议（TCP）简介



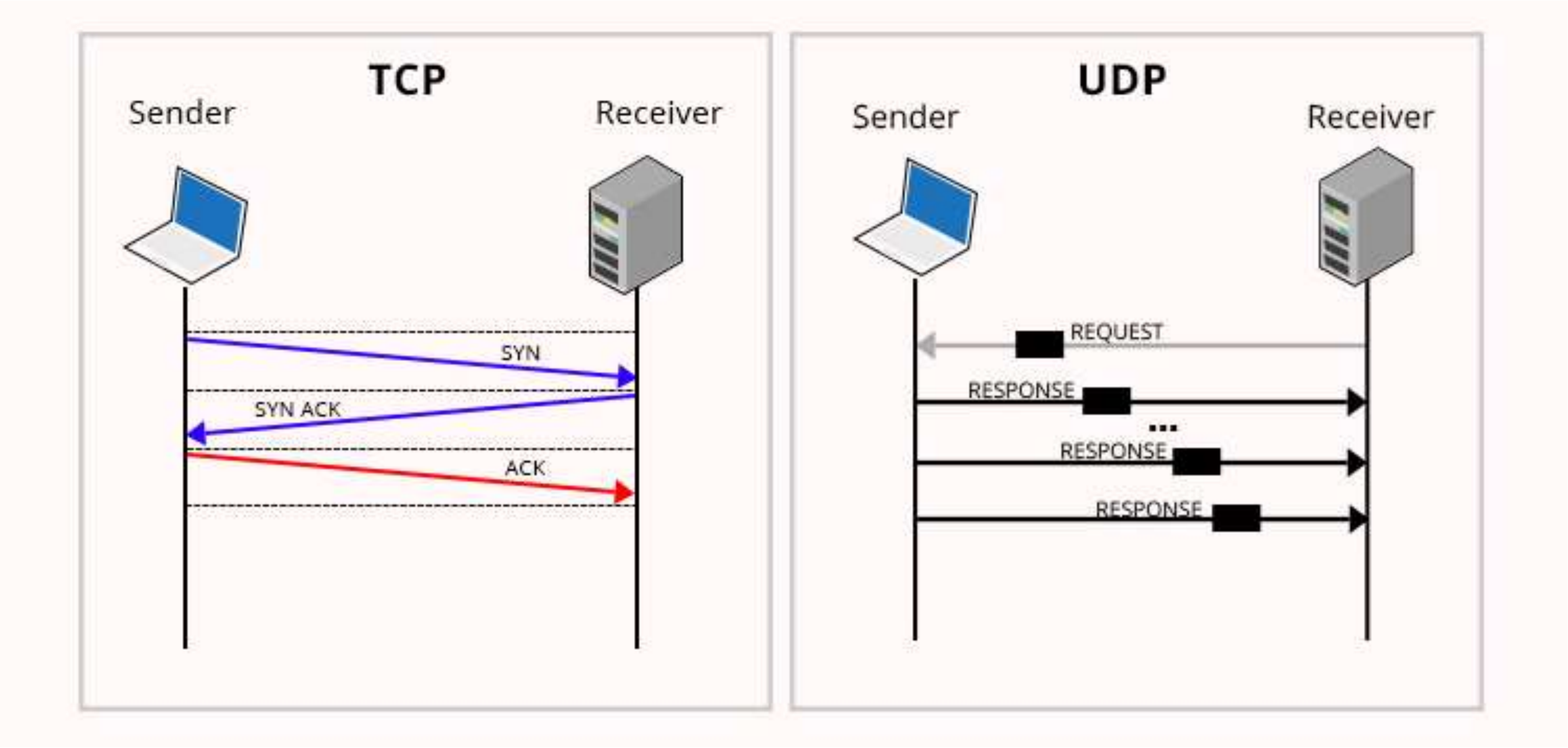
传输控制协议（TCP）简介



TCP与UDP对比

Properties	TCP	UDP
Header	Dynamic header (20 – 60 B)	Static header of 8 Bytes
Max segment	any size or 2^30 B	short message 65536 Bytes
Flow Control	Yes, Window and seq. no.	NO
Checksum	Compulsory	Optional
Connection nature	TCP+ IP = connection oriented	UDP+ IP= connection less
Error control	Own mechanism	Depends on ICMP (No self feature)
Support multicast	NO	YES
Support broadcast	NO	Yes
Examples service	HTTP,SMTP,FTP,TELNET	TFTP,DNS,SNMP

TCP与UDP对比


TCP
Sender
Receiver
SYN
SYN ACK
ACK
UDP
Sender
Receiver
REQUEST
RESPONSE
...
RESPONSE
RESPONSE

究竟用TCP还是UDP

- 客户端主动发起间歇性的无状态的查询——HTTP/HTTPS over TCP
- 客户端和服务器都可以独立主动发包，但是偶尔发生延迟可以容忍——TCP
- 客户端和服务器都可以独立发包，而且无法忍受延迟——UDP
- 其他情况——relUDP或KCP等自定协议

端口号

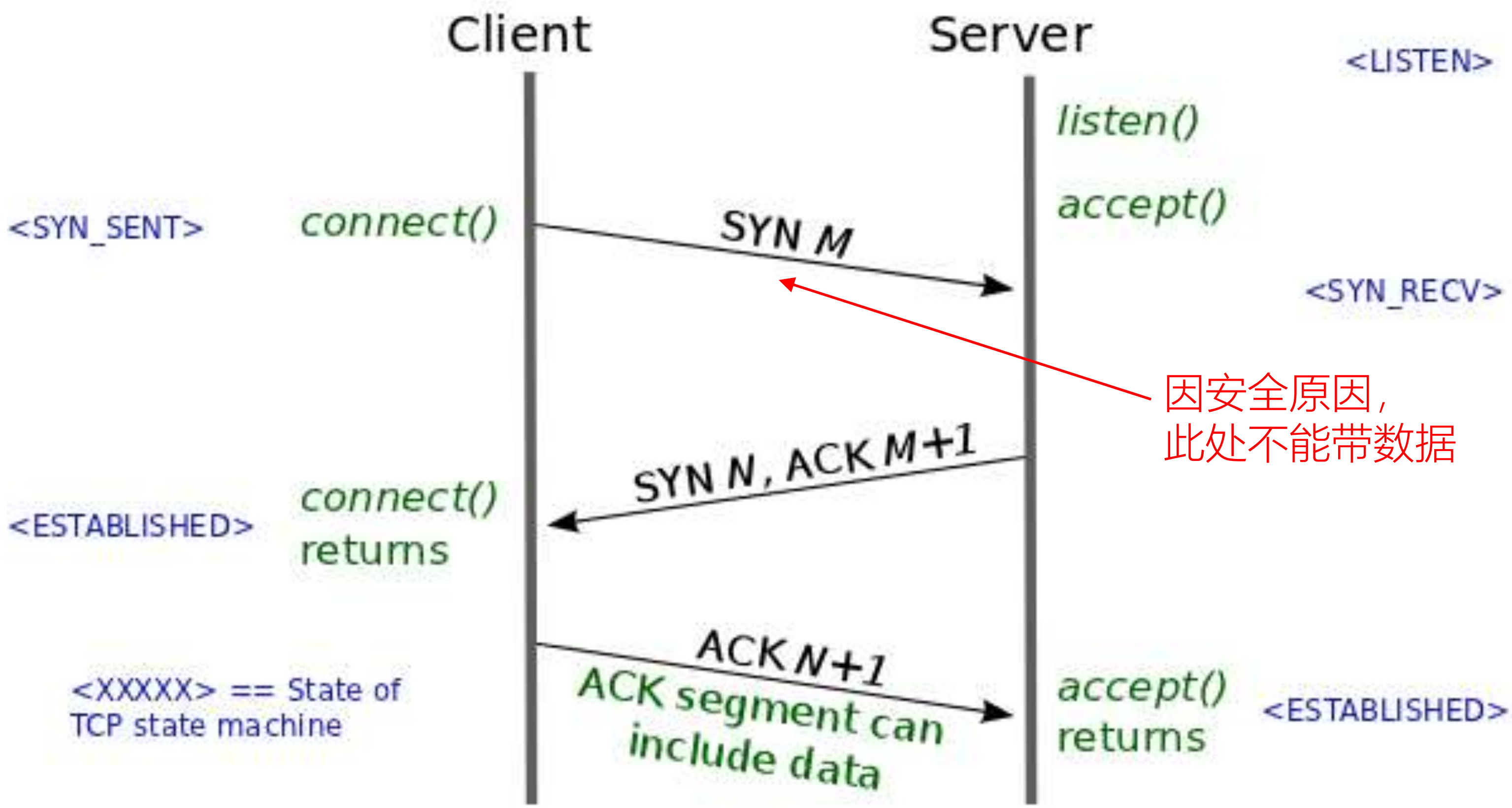
- 经由软件创建的服务
- 通信的端点
- 提供多路复用能力
- 由本机地址、本机端口号、目标机地址、目标机端口号、通信协议组成的五元组唯一确定一个网络连接
- 低于1024的是公认连接端口（Well-known port）

常见网络服务及其端口号

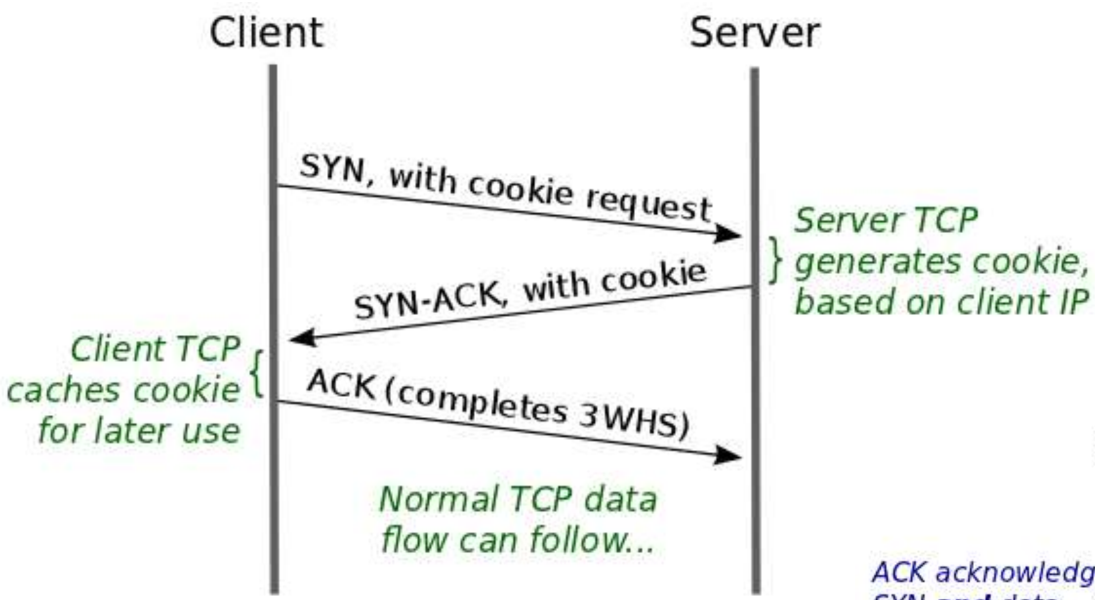
- 21: File Transfer Protocol (FTP)
- 22: Secure Shell (SSH)
- 23: Telnet remote login service
- 25: Simple Mail Transfer Protocol (SMTP)
- 53: Domain Name System (DNS) service
- 80: Hypertext Transfer Protocol (HTTP)
- 110: Post Office Protocol (POP3)
- 123: Network Time Protocol (NTP)
- 143: Internet Message Access Protocol (IMAP)
- 161: Simple Network Management Protocol (SNMP)
- 194: Internet Relay Chat (IRC)
- 443: HTTP Secure (HTTPS)

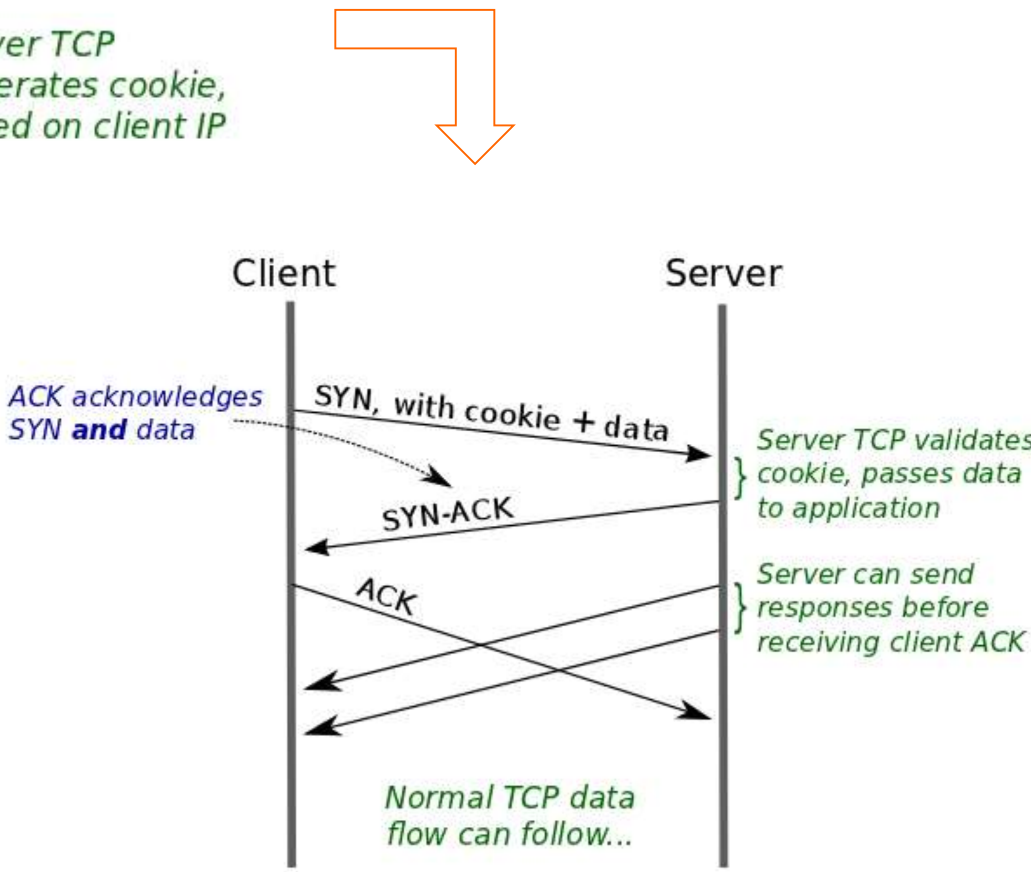
深入TCP协议

三次握手——TCP连接的建立

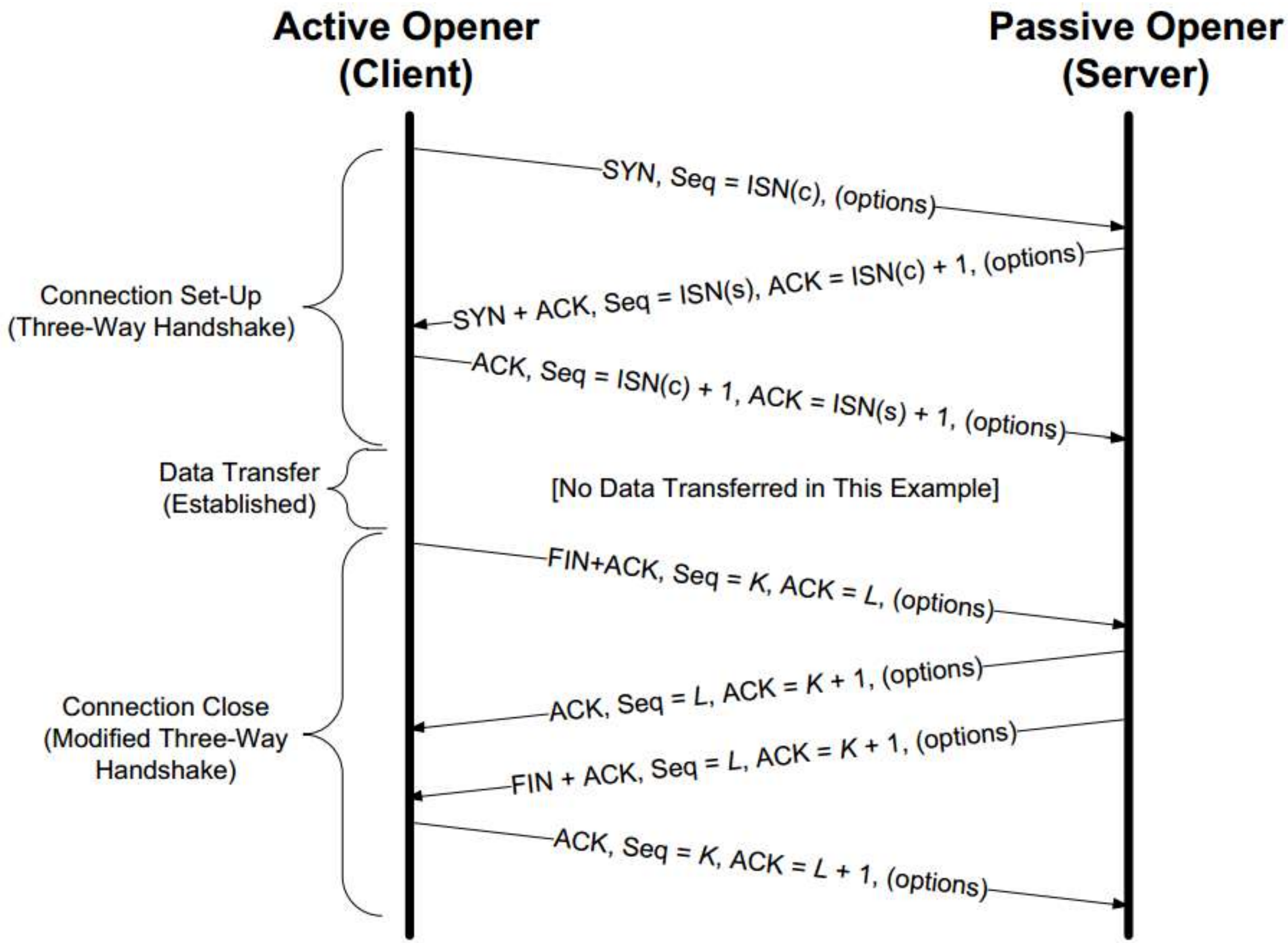


能更快吗——TCP快速打开（Fast Open）



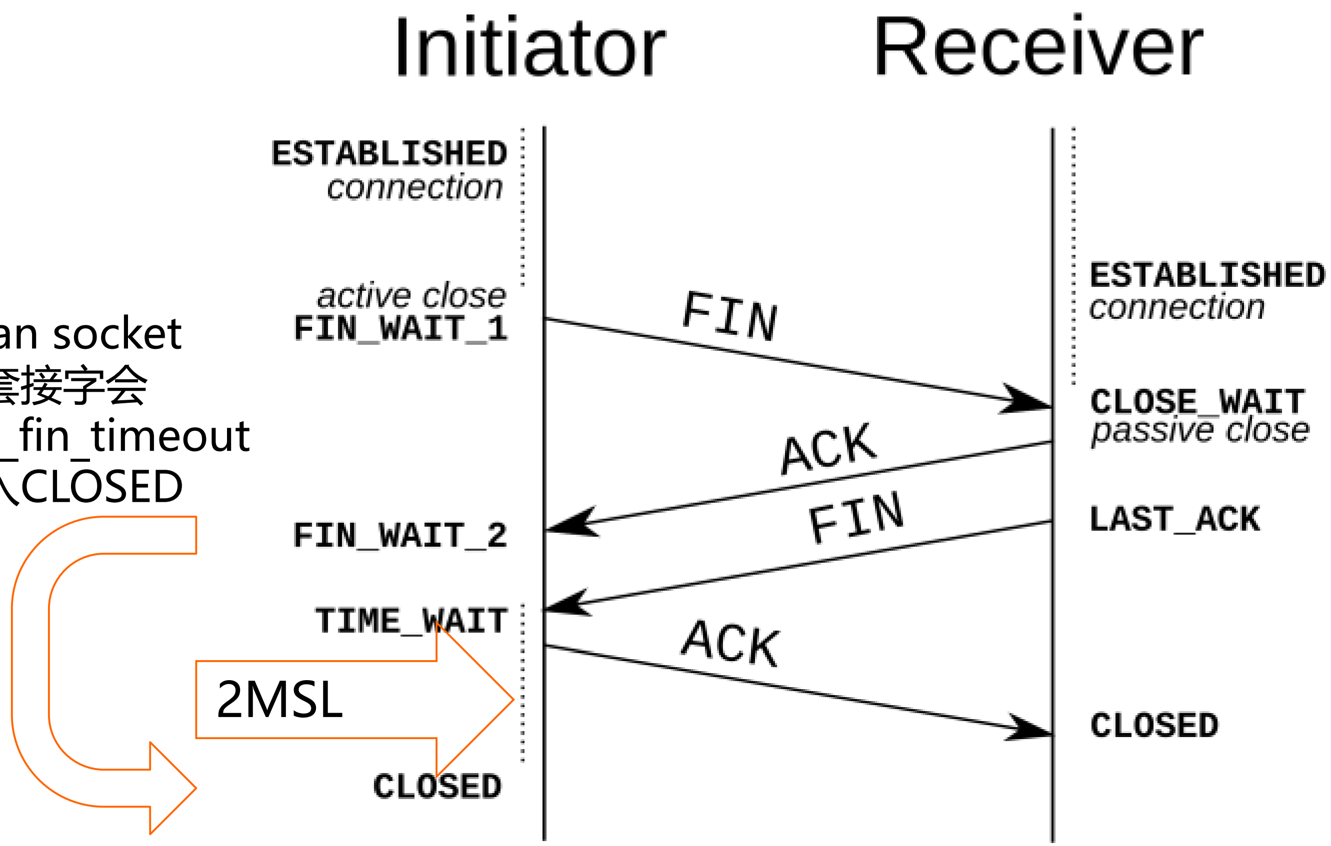


四次挥手——TCP连接的释放

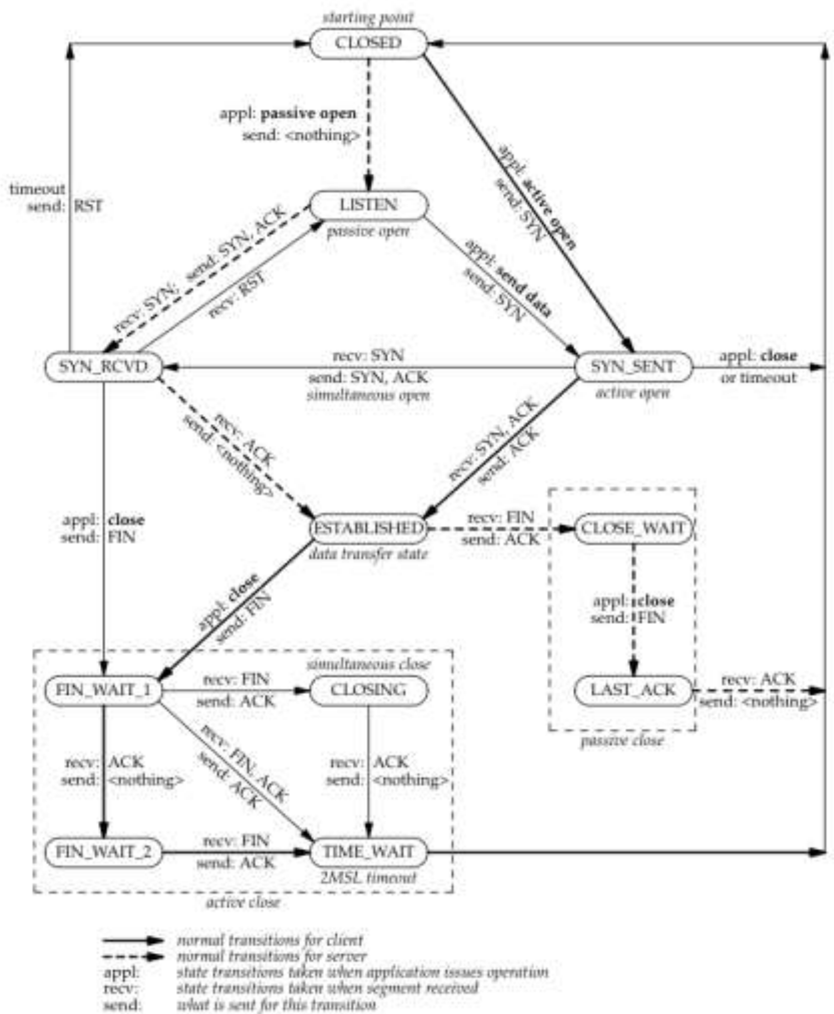


释放的难题——TIME_WAIT状态

orphan socket
孤立套接字会
在tcp_fin_timeout
后进入CLOSED

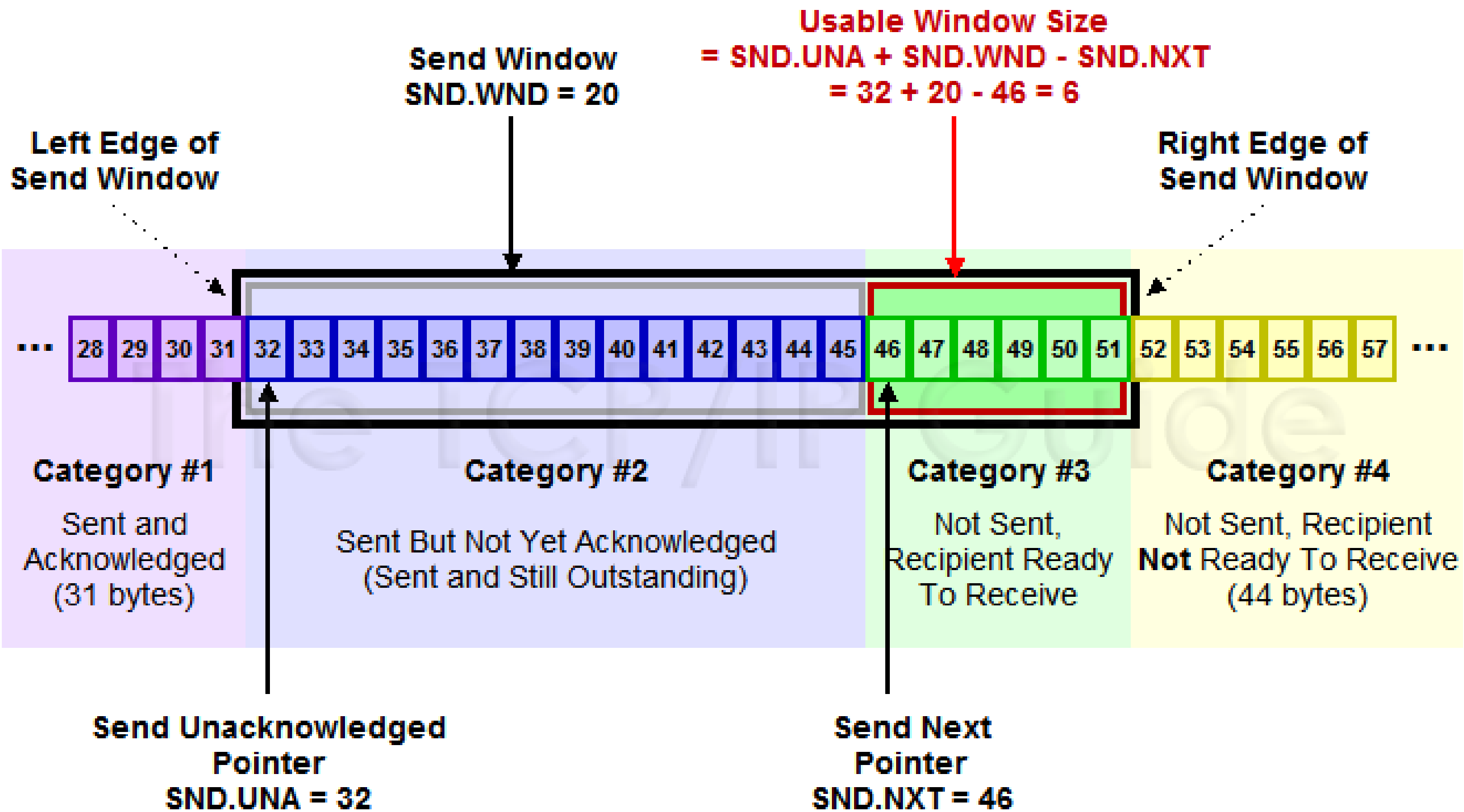


TCP完整状态机



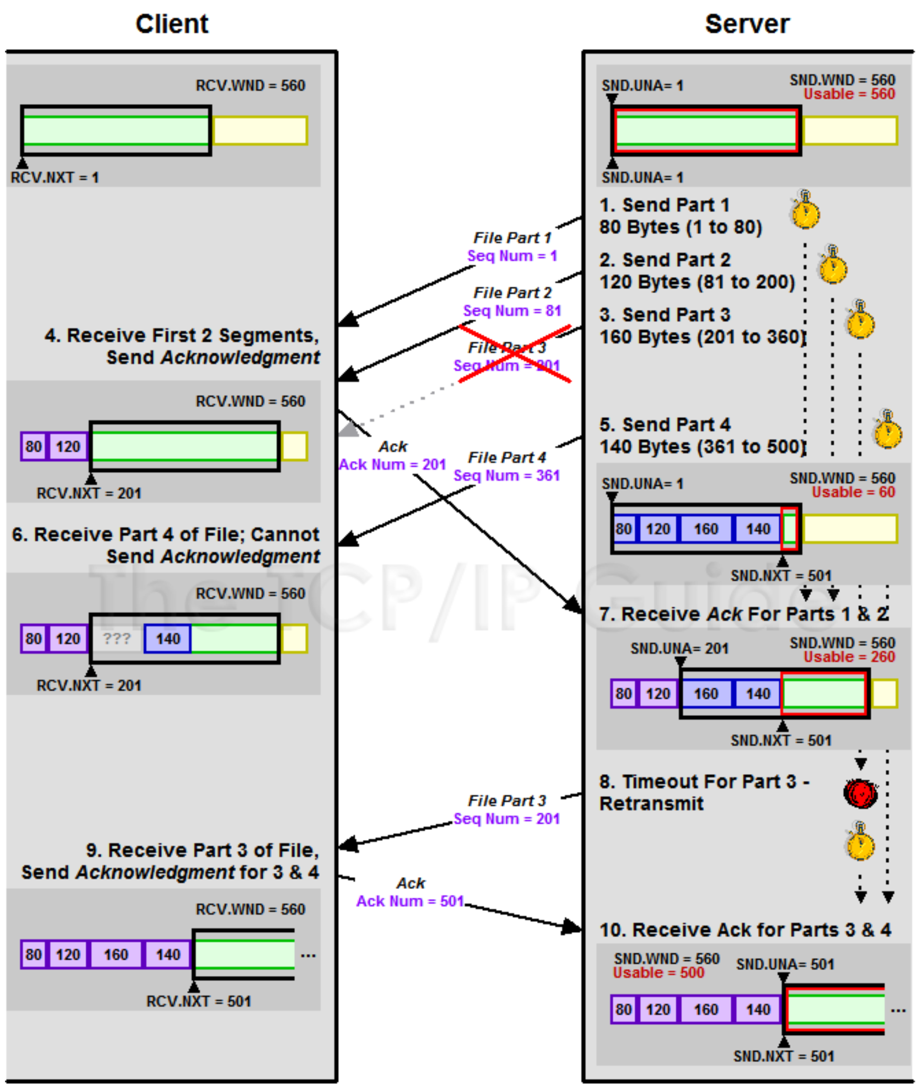
TCP state transition diagram.

ACK与滑动窗口



Usable Window Size
= SND.UNA + SND.WND - SND.NXT
= 32 + 20 - 46 = 6
Send Window
SND.WND = 20
Left Edge of Send Window
Right Edge of Send Window
... 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 ...
Category #1
Sent and Acknowledged (31 bytes)
Category #2
Sent But Not Yet Acknowledged (Sent and Still Outstanding)
Category #3
Not Sent, Recipient Ready To Receive
Category #4
Not Sent, Recipient Not Ready To Receive (44 bytes)
Send Unacknowledged Pointer
SND.UNA = 32
Send Next Pointer
SND.NXT = 46

ACK与滑动窗口

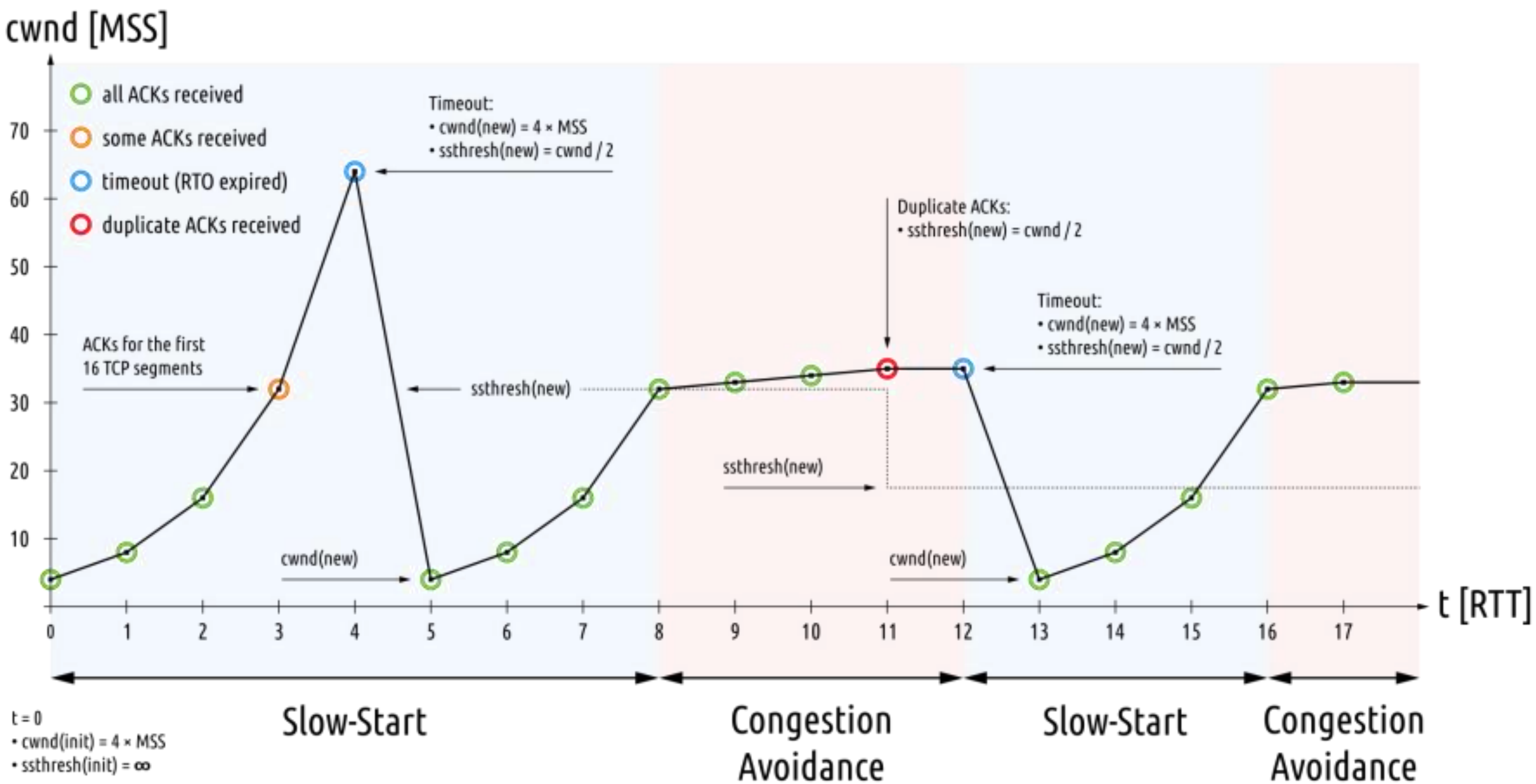


Client
Server
RCV.WND = 560
RCV.NXT = 1
SND.UNA= 1
SND.WND = 560
Usable = 560
1. Send Part 1
80 Bytes (1 to 80)
2. Send Part 2
120 Bytes (81 to 200)
3. Send Part 3
160 Bytes (201 to 360)
File Part 1
Seq Num = 1
File Part 2
Seq Num = 81
File Part 3
Seq Num = 201
4. Receive First 2 Segments,
Send Acknowledgment
RCV.NXT = 201
Ack
Ack Num = 201
5. Send Part 4
140 Bytes (361 to 500)
File Part 4
Seq Num = 361
Usable = 60
SND.NXT = 501
6. Receive Part 4 of File; Cannot
Send Acknowledgment
???
7. Receive Ack For Parts 1 & 2
SND.UNA= 201
Usable = 260
8. Timeout For Part 3 -
Retransmit
9. Receive Part 3 of File,
Send Acknowledgment for 3 & 4
Ack Num = 501
RCV.NXT = 501
10. Receive Ack for Parts 3 & 4
Usable = 500
SND.UNA= 501
80
120
160
140

另一个窗口——拥塞窗口（CWND）

- CWND开始时很小
- 随传输慢慢扩增到RWND
- 网络状况发生变化（RTT增加、ACK重传）时减小
- 再逐步恢复RWND

拥塞控制


cwnd [MSS]
all ACKs received
some ACKs received
timeout (RTO expired)
duplicate ACKs received
70
60
50
40
30
20
10
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
Duplicate ACKs:
• ssthresh(new) = cwnd / 2
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
ACKs for the first
16 TCP segments
ssthresh(new)
ssthresh(new)
cwnd(new)
cwnd(new)
t [RTT]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
t = 0
• cwnd(init) = 4 × MSS
• ssthresh(init) = ∞
Slow-Start
Congestion Avoidance
Slow-Start
Congestion Avoidance

优秀拥塞控制算法简介

- 指数递增拥塞控制（BIC）
- 平滑的指数递增拥塞控制（CUBIC）

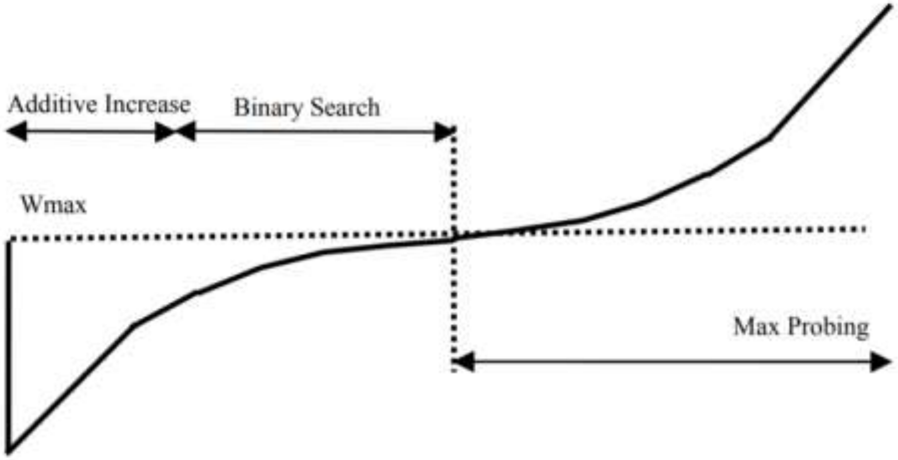


Fig. 1: The Window Growth Function of BIC

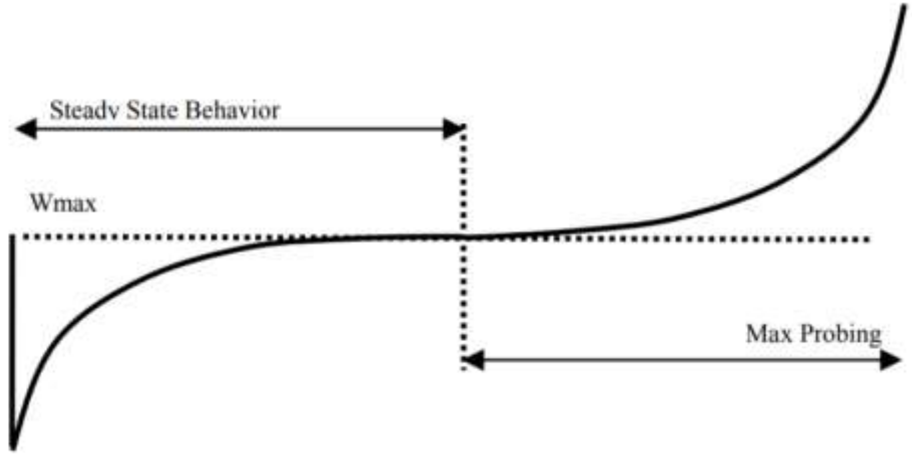
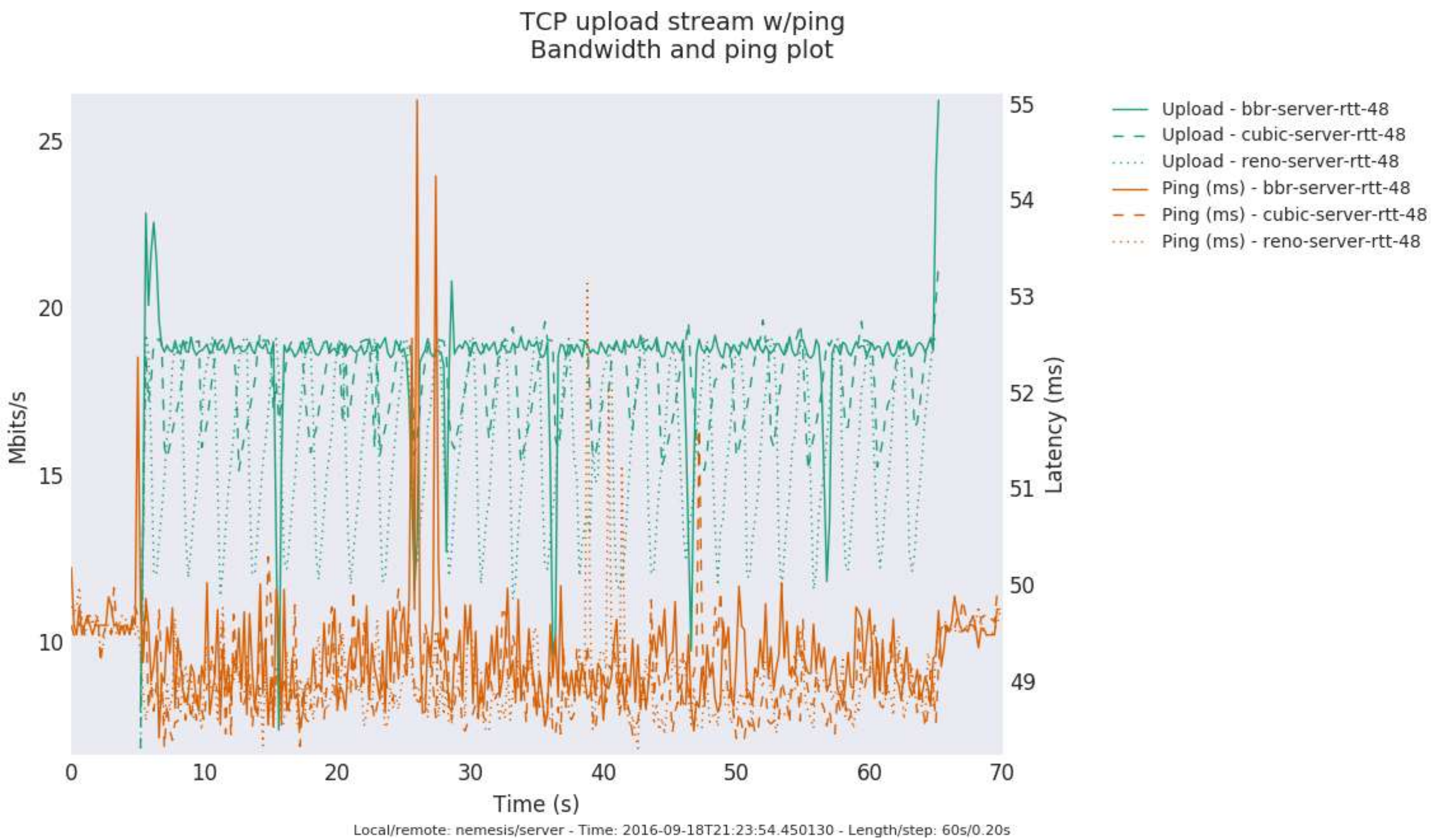


Fig. 2: The Window Growth Function of CUBIC

优秀拥塞控制算法简介

- TCP-BBR



套接字编程

套接字地址结构

- 套接字地址结构均以sockaddr_开头，并以对应每个协议簇的唯一后缀结尾。
- 当作为一个参数传递进任何套接字函数时，套接字地址总是以引用形式来传递。
- 类似于void*代表通用指针类型，通用套接字地址结构可以便于参数传递，使得套接字函数能够处理来自所支持的任何协议簇的套接字地址结构。

套接字地址类型

```
//ipv4的套接字地址结构
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;//32位的ipv4地址
};

//网络套接字地址结构
struct sockaddr_in
{
    uint8_t sin_len //套接字地址结构的长度
    sa_family_t sin_family // AF_INET
    in_port_t sin_port; //端口号
    struct in_addr sin_addr; //网络地址
    unsigned char sin_zero[8]; //不使用，该字段必须为0
};
```

套接字地址使用方法

```
struct sockaddr_in servaddr;//声明一个套接字地址
bzero(&servaddr, sizeof(servaddr));//套接字结构体清0
servaddr.sin_family = AF_INET;//使用ipv4的协议簇
servaddr.sin_port = htons(13);//端口号，13为获取日期和时间的端口

Bind(
    listenfd,
    (struct sockaddr_in *) &servaddr,
    sizeof(servaddr)
);//必须对servaddr进行强制转换
```

地址变换函数

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

- int inet_aton(const char *cp, struct in_addr *inp);
- char *inet_ntoa(struct in_addr in);
- in_addr_t inet_addr(const char *cp);
- in_addr_t inet_network(const char *cp);
- struct in_addr inet_makeaddr(int net, int host);
- in_addr_t inet_lnaof(struct in_addr in);
- in_addr_t inet_netof(struct in_addr in);

```
#include <arpa/inet.h>
```

- const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
- int inet_pton(int af, const char *src, void *dst);

字节序函数

- #include <arpa/inet.h>
- uint32_t htonl(uint32_t hostlong);
- uint16_t htons(uint16_t hostshort);
- uint32_t ntohl(uint32_t netlong);
- uint16_t ntohs(uint16_t netshort);

字节操纵函数

- #include <string.h>
- //b开头（表示字节）的一组函数，POSIX 2001-2008
- void bzero(void* dest,size_t nbytes);
- void bcopy(const void* src,void *dest , size_t nbytes);
- int bcmp(const void *ptr1, const void* ptr2, size_t nbytes);

- //mem开头（表示内存）的一组函数，C规范
- void *memset(void *dest , int c , size_t len);
- void *memcpy(void *dest , const void* src , size_t nbytes);
- int memcmp(const void *ptr1 , const void *ptr2, size_t nbytes);

输入+输出参数——值-结果函数

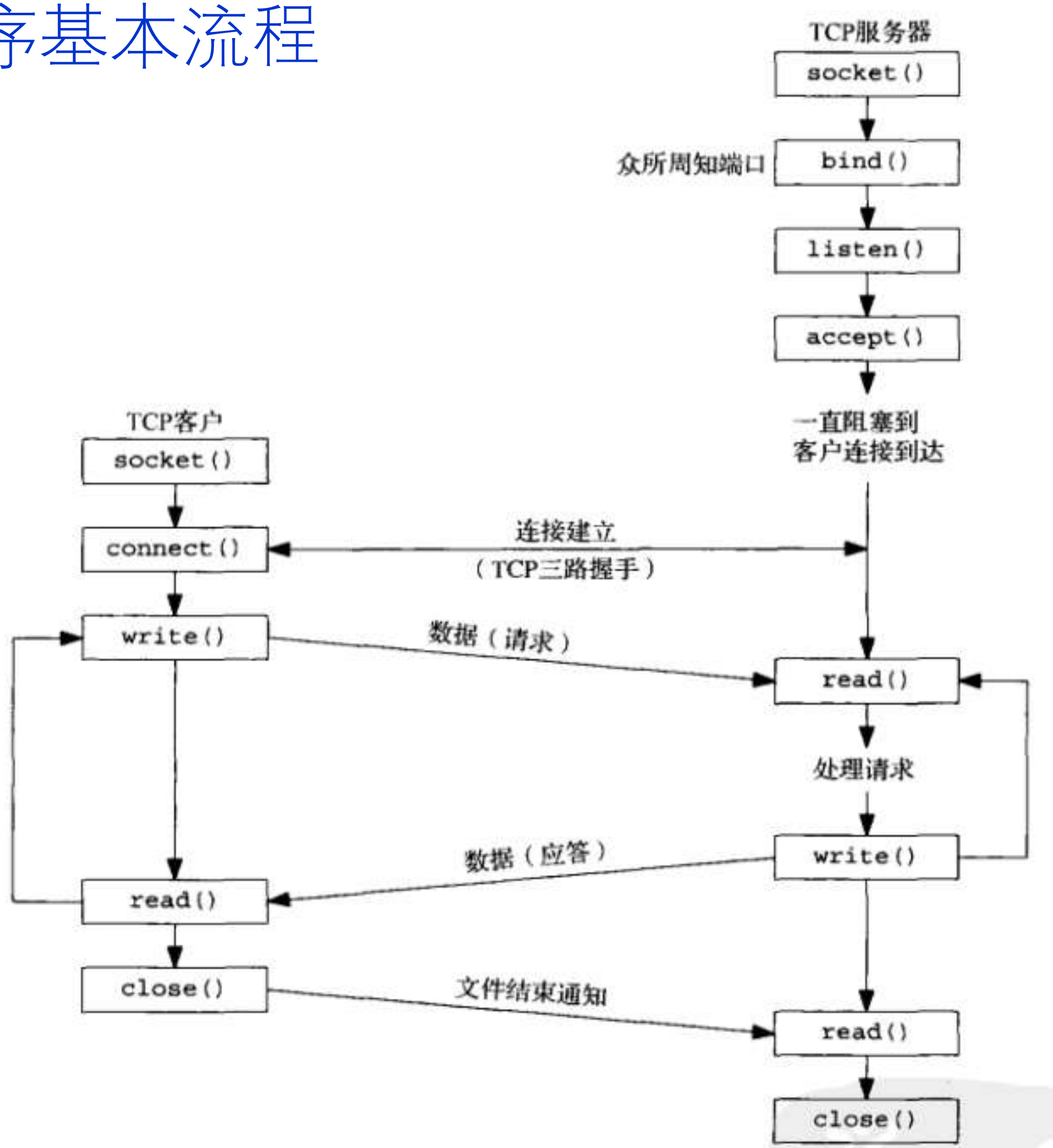
- #include <sys/socket.h>
- int bind(int socket, const struct sockaddr *address, socklen_t address_len);
- int connect(int socket, const struct sockaddr *address, socklen_t address_len);
- ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);

输入+输出参数——值-结果函数

- #include <sys/socket.h>
- int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);
- ssize_t recvfrom(int socket, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict address, socklen_t *restrict address_len);
- int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);
- int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);

TCP套接字编程

TCP套接字程序基本流程



创建Socket——socket

int socket(int *family*, **int** *type*, **int** *protocol***);**

family	**说明**	**type**	**说明2**	**protocol**	**说明3**
AF_INET	IPv4协议	SOCK_STREAM	字节流套接字	IPPROTO_TCP	TCP传输协议
AF_INET6	IPv6协议	SOCK_DGRAM	数据报套接字	IPPROTO_UDP	UDP传输协议
AF_LOCAL	Unix域协议	SOCK_SEQPACKET	有序数组套接字	IPPROTO_SCTP	SCTP传输协议
AF_ROUTE	路由套接字	SOCK_RAW	原始套接字		
AF_KEY	密钥套接字				

创建Socket——socket

int socket(int *family*, **int** *type*, **int** *protocol***);**

Protocol	**AF_INET**	**AF_INET6**	**AF_LOCAL**	**AF_ROUTE**	**AF_KEY**
SOCK_STREAM	TCP/SCTP	TCP/SCTP	是		
SOCK_DGRAM	UDP	UDP	是		
SOCK_SEQPACKET	SCTP	SCTP	是		
SOCK_RAW	IPv4	IPv6		是	是

连接到服务器——connect

int connect(int *socket*, **const struct sockaddr** ***address*, socklen_t *address_len*);

- 成功则返回0
- 如果返回-1，则需要检查errno变量(errno.h)获知具体原因
- 如果返回-1，则应该关闭（close）该socket

思考——客户端所用IP地址/端口号是如何确定的?

- 查看客户端地址/端口号
  - netstat -npt
- 是否能修改端口号?
  - cat /proc/sys/net/ipv4/ip_local_port_range
  - sudo bash -c "echo 32769 61000 > /proc/sys/net/ipv4/ip_local_port_range "
- 是否能指定所用地址?

发送到服务器——write

```
ssize_t                                         /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
        size_t          nleft;
        ssize_t         nwritten;
        const char      *ptr;
        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
                        if (nwritten < 0 && errno == EINTR)
                                nwritten = 0;           /* and call write() again */
                        else
                                return(-1);                     /* error */
                }
                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(n - nleft);
}
```

从服务器接收——read

```
ssize_t                                         /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
        size_t  nleft;
        ssize_t nread;
        char    *ptr;
        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nread = read(fd, ptr, nleft)) < 0) {//循环读取
                        if (errno == EINTR)
                                nread = 0;              /* and call read() again */
                        else
                                return(-1);
                } else if (nread == 0)
                        break;                          /* EOF */
                nleft -= nread;
                ptr   += nread;
        }
        return(n - nleft);              /* return >= 0 */
}
/* end readn */
```

关闭连接——close/shutdown

- int close(int *fildes*);
- int shutdown(int *socket*, int *how*);
  - SHUT_RD - Disables further receive operations.
  - SHUT_WR - Disables further send operations.
  - SHUT_RDWR - Disables further send and receive operations.

函 数	说 明
shutdown, SHUT_RD	在套接字上不能再发出接收请求；进程仍可往套接字发送数据；套接字接收缓冲区中所有数据被丢弃；再接收到的任何数据由TCP丢弃（习题6.5）；对套接字发送缓冲区没有任何影响。
shutdown, SHUT_WR	在套接字上不能再发出发送请求；进程仍可从套接字接收数据；套接字发送缓冲区中的内容被发送到对端，后跟正常的TCP连接终止序列（即发送FIN）；对套接字接收缓冲区无任何影响。
close, l_onoff = 0（默认情况）	在套接字上不能再发出发送或接收请求；套接字发送缓冲区中的内容被发送到对端。如果描述符引用计数变为0：在发送完发送缓冲区中的数据后，跟以正常的TCP连接终止序列（即发送FIN）；套接字接收缓冲区中内容被丢弃。

综合实例——hello-client

- 参见simple-hello/hello-client.c

指定套接字地址——bind

- int bind(int *socket*, **const struct sockaddr** **address*, socklen_t *address_len*);

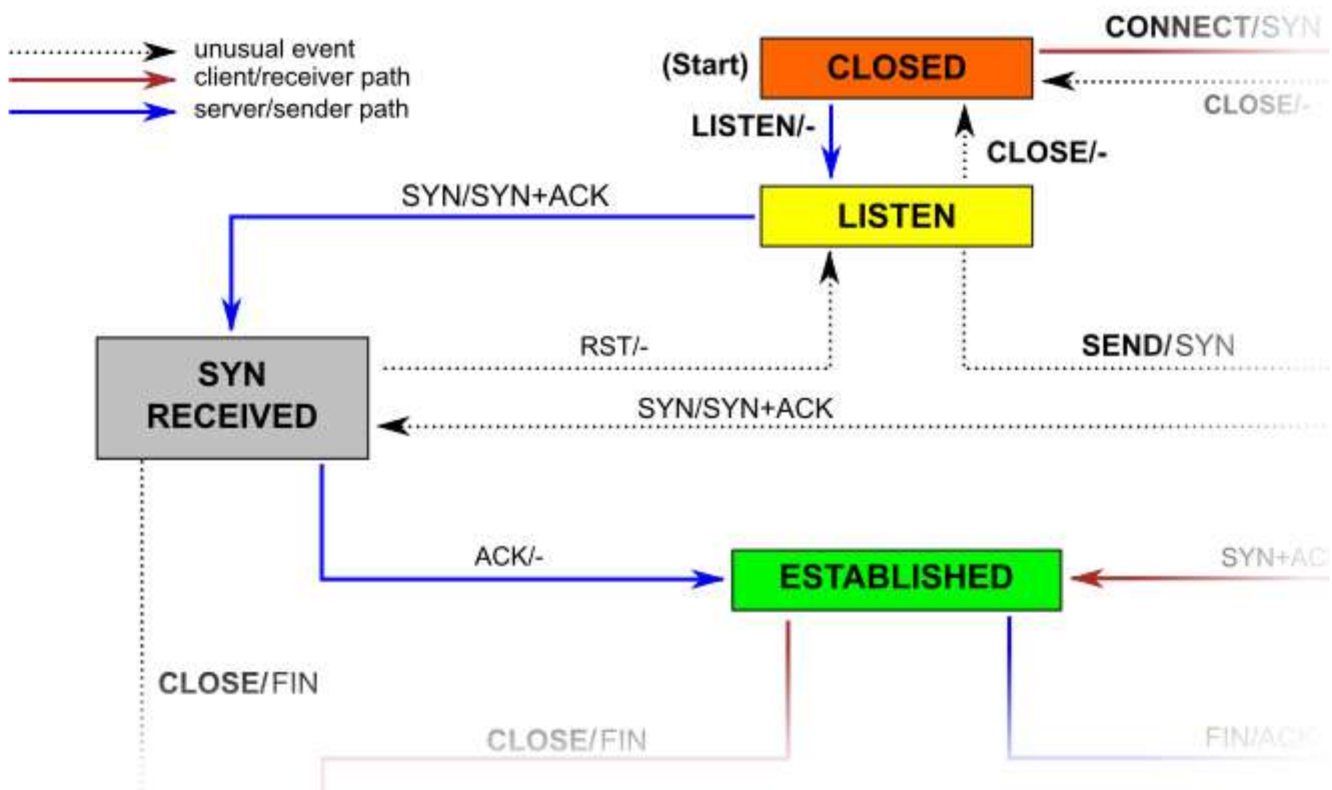
思考：确定端口号

- 系统能帮我们挑选合适的端口号吗？
  - addr.sin_port = 0;
- 如何在所有网口都监听？
  - addr.sin_addr.s_addr = INADDR_ANY;
- 如何显示系统挑选的端口号？

```
addrLen = sizeof(addr);
if (getsockname(fd, (struct sockaddr *)&addr, &addrLen) == -1) {
    printf("getsockname() failed");
}
printf("port=%d \n", addr.sin_port);
```

开始监听——listen

- int listen(int *socket*, **int** *backlog***);**



接受新连接——accept

- int accept(int *socket*, **struct sockaddr *restrict** *address*, socklen_t *restrict *address_len*);
- **将阻塞直至能够返回一个已经建立（三次握手成功）的新连接**

综合实例——echo-server

- 参见simple-hello/echo-server.c

TCP服务器/客户端实际程序

实用的服务器程序——并发服务器

- 使用fork
  - pid_t fork(void);
  - 对子进程返回0，对父进程返回子进程PID
  - 父子进程均从fork返回处继续执行
- 每个accept到的客户端传入套接字放到新的进程里处理
- 主进程持续监听和纳入新客户端
- 参考程序fork-server/echo-server.c

处理终止

- 正常终止，即对方套接字关闭，本方收到0字节返回
- 此时父进程应该清理结束的子进程，否则会导致僵死进程的出现

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND          WCHAN
22038 22036 pts/6     S   -bash            read_chan
17870 22038 pts/6     S   ./tcpserv01      wait_for_connect
19315 17870 pts/6     Z   [tcpserv01 <defu do_exit
```

清理结束的子进程

- 捕捉系统通知子进程结束的信号SIGCHLD——注册回调
- 在回调函数中调用wait/waitpid处理已经终止的子进程
  - 每次调用wait/waitpid只处理一个终止子进程
  - 没有子进程终止时，wait会阻塞，waitpid可以不阻塞
- 主函数中处理系统调用产生EINTR错误的情况——重启系统调用

处理终止（续）

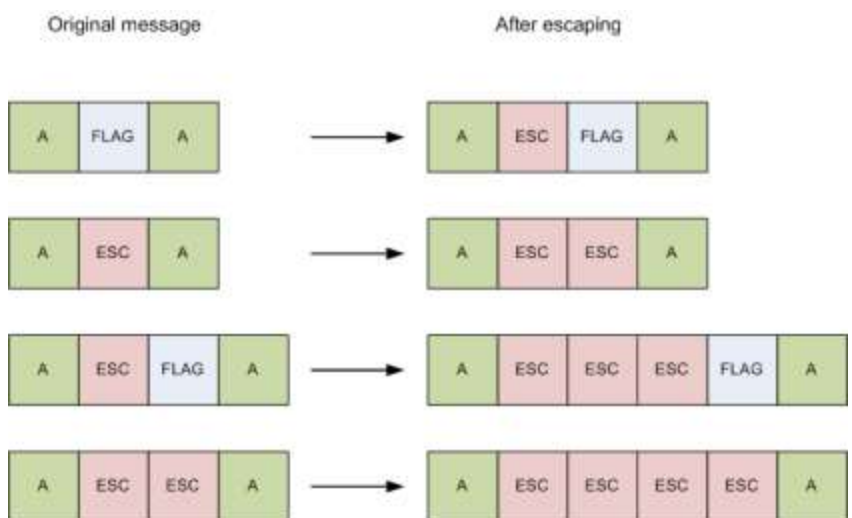
- 父进程先于子进程结束——子进程被init （PID1）收集
- 程序意外关闭——系统负责关闭套接字
- 系统意外关闭，后未重启——/proc/sys/net/netfilter/ip_conntrack_tcp_timeout_established （需要模块nf_conntrack_ipv4）
- 系统意外关闭，后又重启——RESET

数据的编码与解析——序列化与反序列化

- 解决跨平台数据表示问题
- 解决大小端问题
- 常用方法：
  - XML
  - JSON
  - YAML
  - Google Protocol Buffer

流式数据的分割——分帧

- 定长模式
- Type-length-value 类型-长度-值模式，如ASN.1
- Length-Data 长度-数据模式
- 帧定界符模式，如ndjson、MAC帧、各种串口协议等



IO复用

从fork到prefork

- 动态调整子进程数（限制数）
- 隔离进程资源
- 参考Apache prefork模式设计

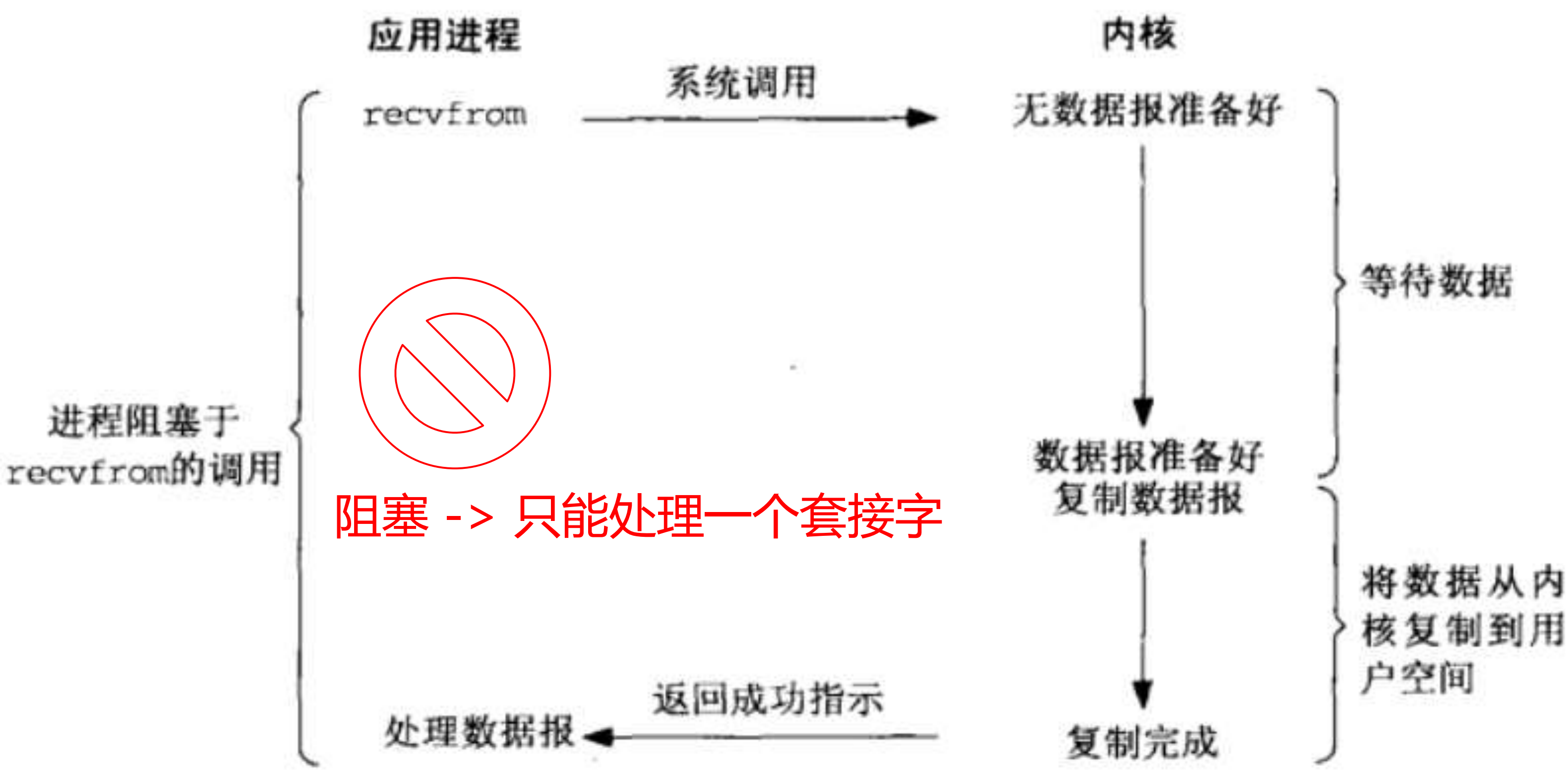
Prefork模型实例

- 参考程序prefork-server/echo-server.c

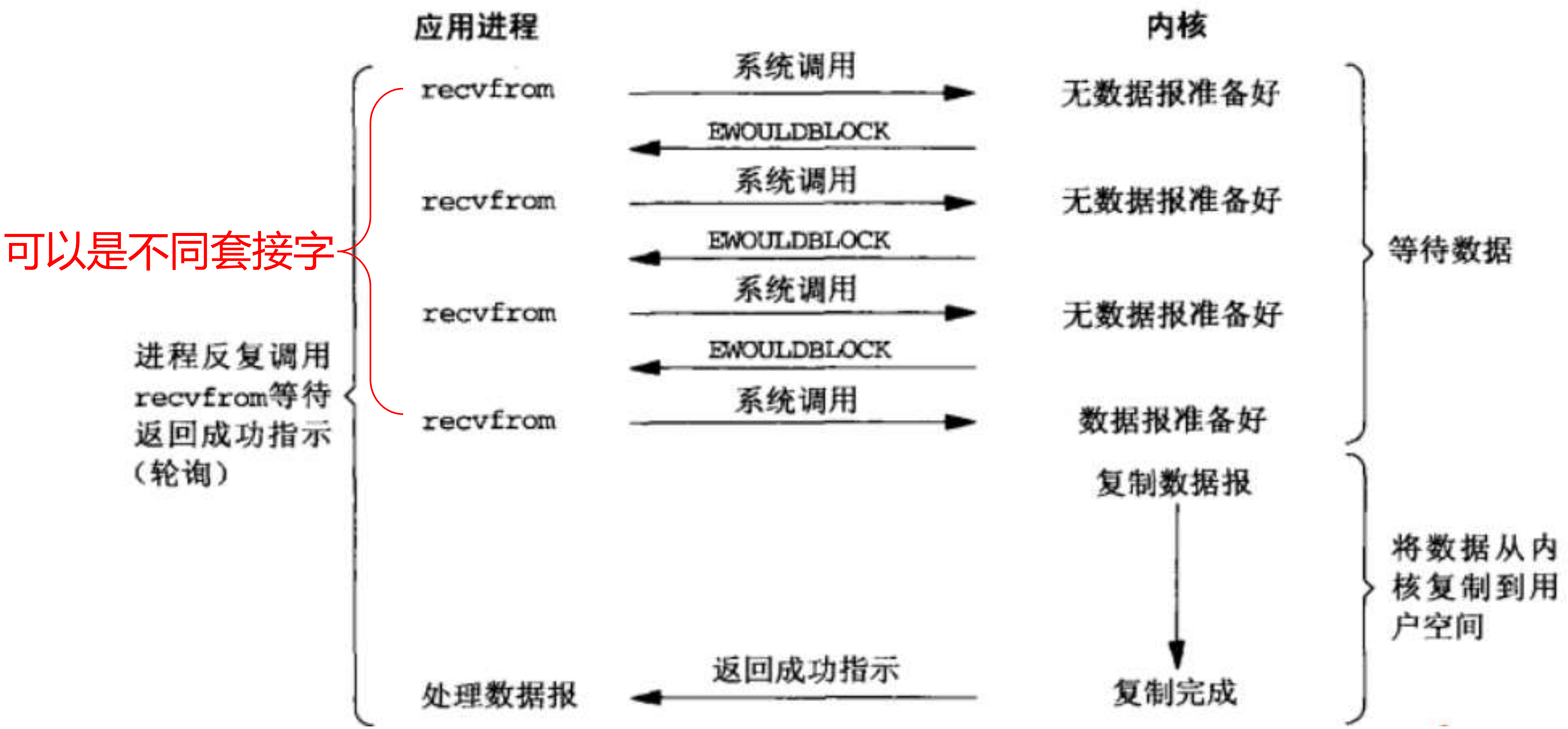
Fork、Prefork无法解决下列问题

- 处理多个描述符（一般是交互式输入或套接字）
- 处理多个套接字（代理/转发）
- TCP服务器既要处理监听套接字，又要处理已连接套接字
- 既要适用/接收TCP，又要适用/接收UDP
- 如果一个服务器要处理多个服务或者多个协议

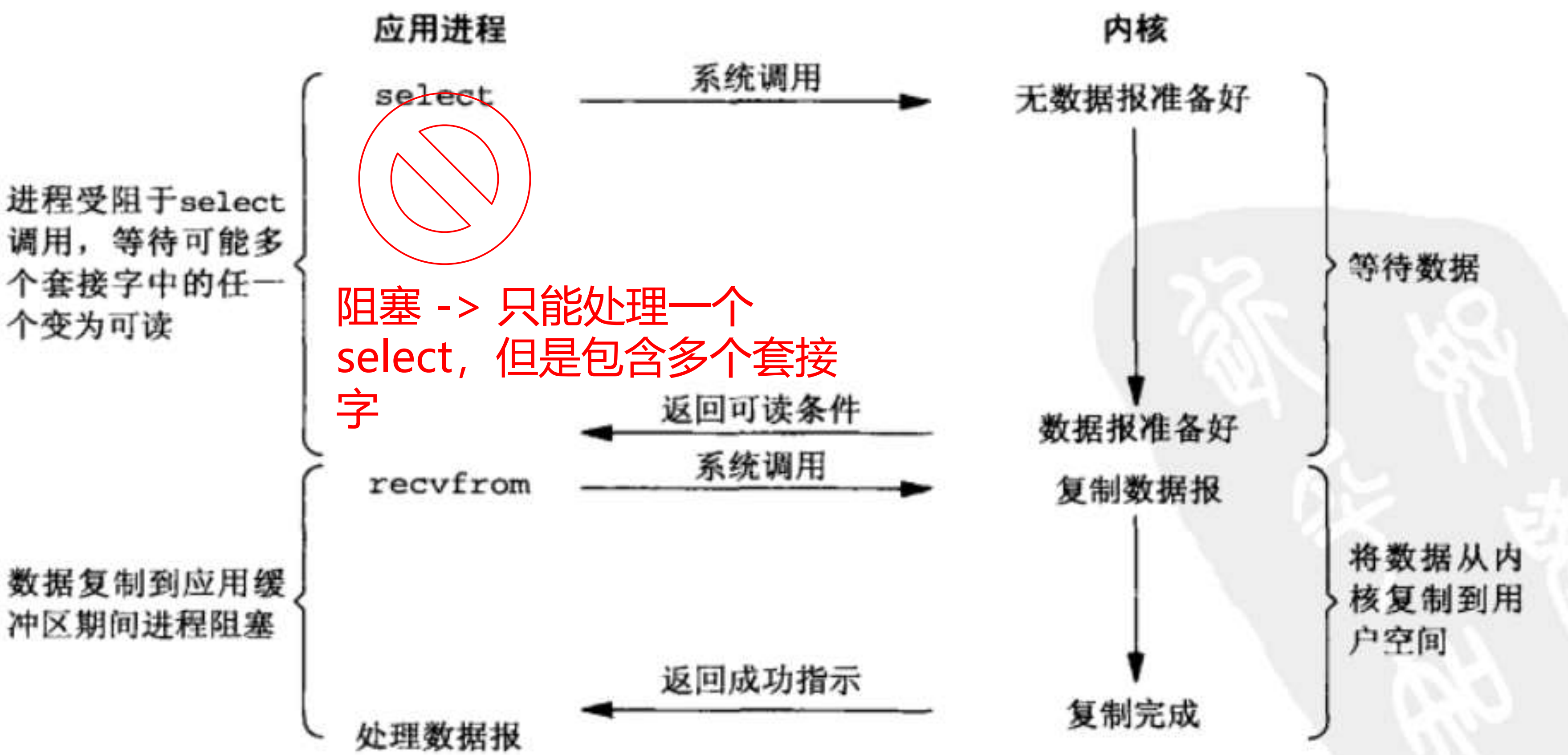
阻塞模型


应用进程
内核
recvfrom
系统调用
无数据报准备好
等待数据
进程阻塞于
recvfrom的调用
阻塞 -> 只能处理一个套接字
数据报准备好
复制数据报
将数据从内
核复制到用
户空间
返回成功指示
处理数据报
复制完成

非阻塞（轮询）

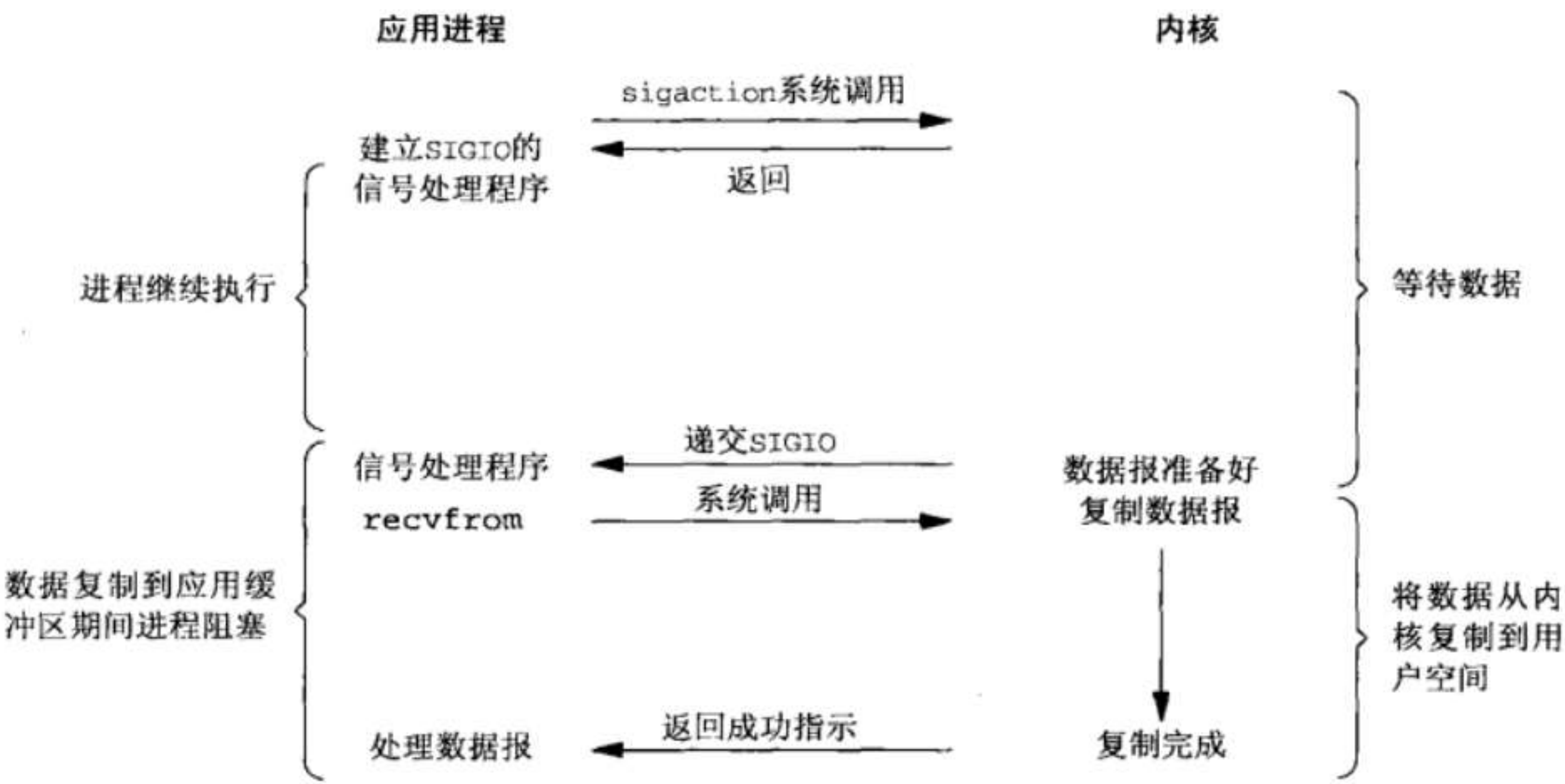

应用进程
内核
系统调用
recvfrom
无数据报准备好
EWOULDBLOCK
系统调用
recvfrom
无数据报准备好
EWOULDBLOCK
可以是不同套接字
等待数据
系统调用
recvfrom
无数据报准备好
EWOULDBLOCK
进程反复调用
recvfrom等待
返回成功指示
（轮询）
系统调用
recvfrom
数据报准备好
复制数据报
将数据从内
核复制到用
户空间
返回成功指示
处理数据报
复制完成

IO复用

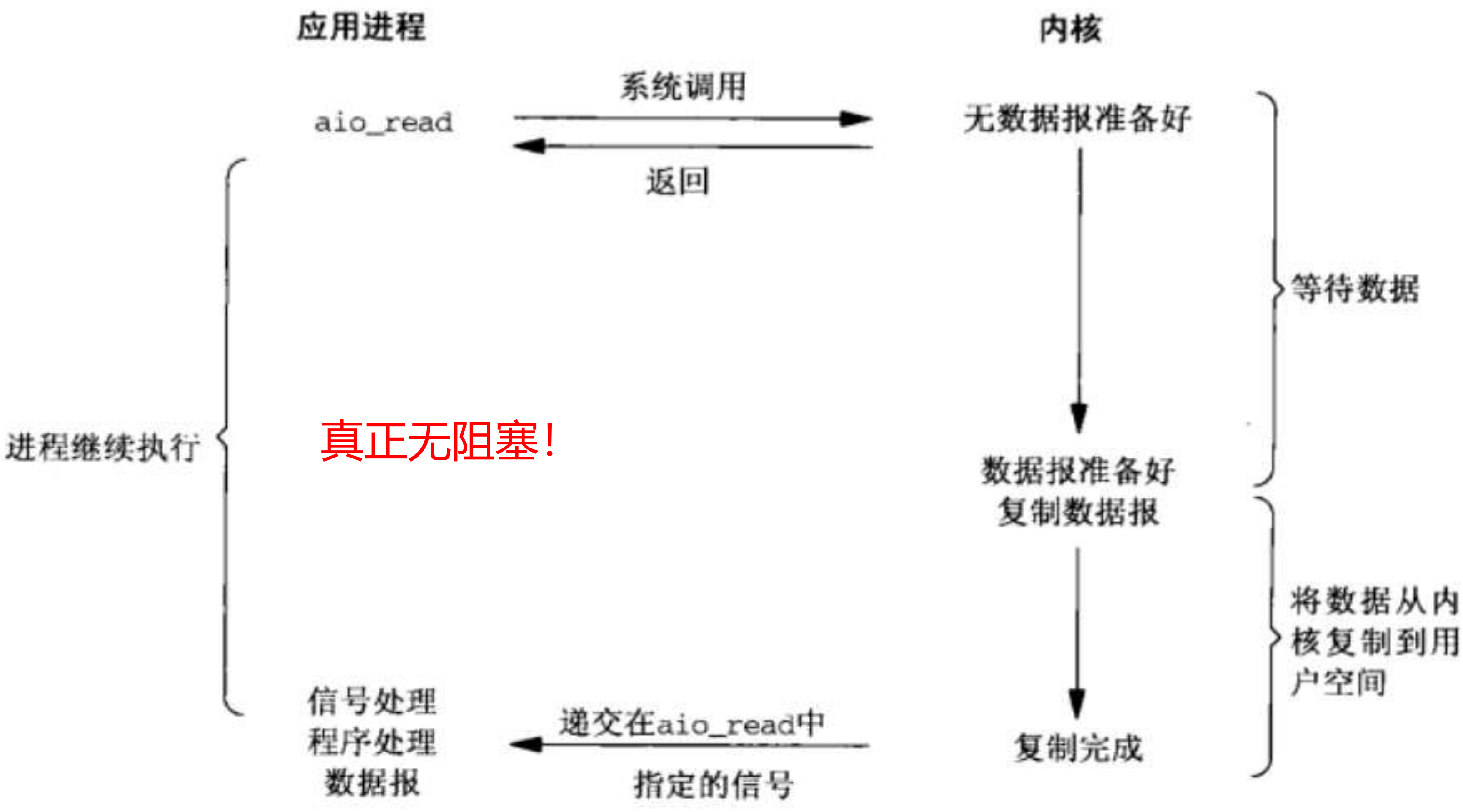


阻塞 -> 只能处理一个select，但是包含多个套接字

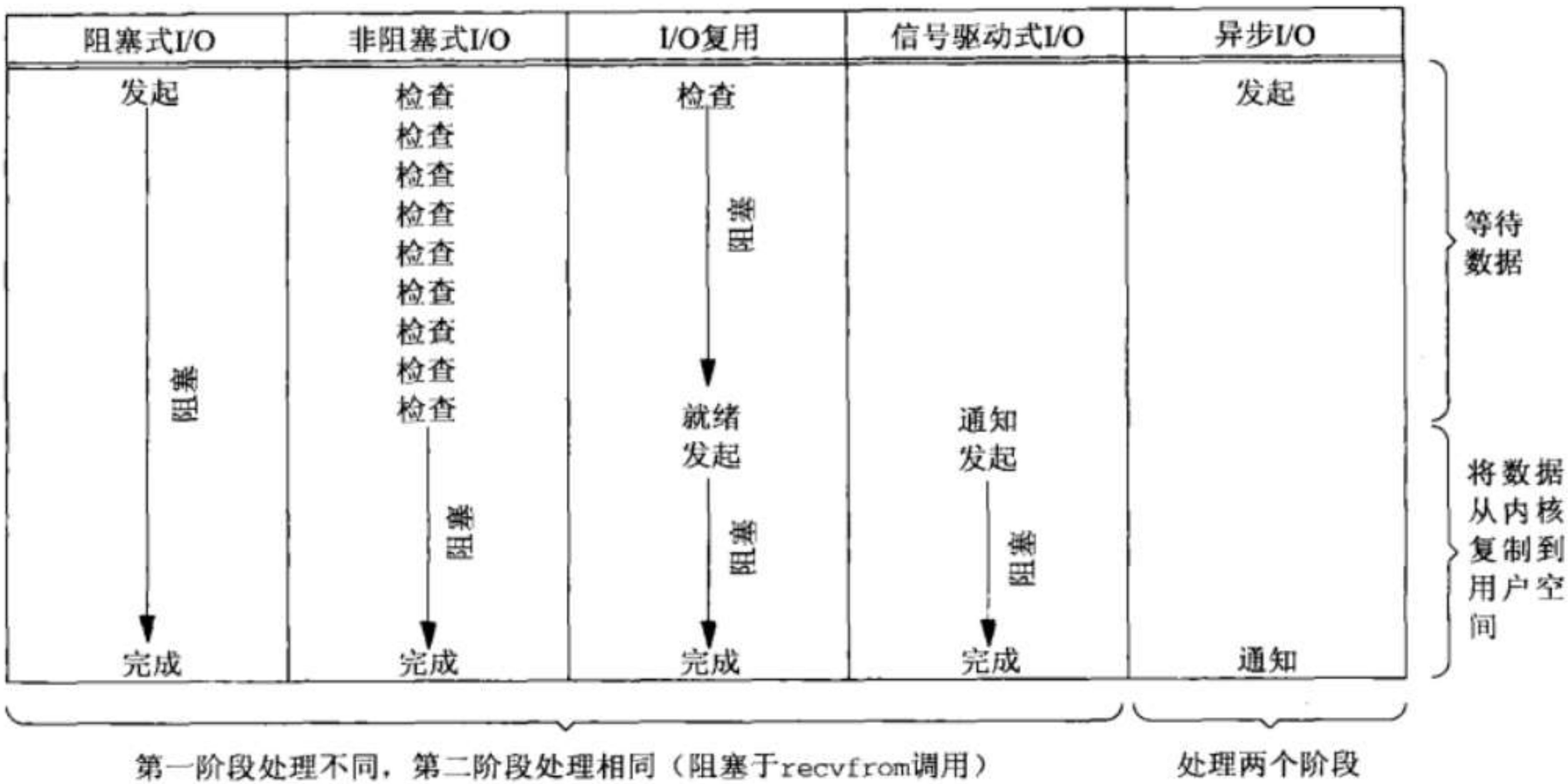
信号驱动模型（类似硬件中断）



异步IO模型（类似DMA）



各种IO模型比较



阻塞式I/O
非阻塞式I/O
I/O复用
信号驱动式I/O
异步I/O
发起
检查
检查
检查
检查
检查
检查
检查
检查
检查
阻塞
就绪
发起
通知
发起
完成
完成
完成
完成
通知
等待数据
将数据从内核复制到用户空间
第一阶段处理不同，第二阶段处理相同（阻塞于recvfrom调用）
处理两个阶段

IO复用模型——select基本流程

```
int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

- 初始化FD集合
- 设置感兴趣的FD
- 调用select
- 对每一个感兴趣的FD
  - 查询是否在返回FD集合中
  - 做对于的处理

select基本示例

- 参考程序select-server/echo-server.c

Select模型存在的问题

- FDSET的内部限制导致了传入FD数量有限
- FDSET被修改导致频繁重复初始化
- 支持的查询太粗糙（读、写、错误）
- 返回后还需要进行线性探测，效率太低

非阻塞式IO——poll函数

int poll(struct pollfd *fds*[], nfds_t *nfds*, int *timeout*);

```
// The structure for two events
struct pollfd fds[2];

// Monitor sock1 for input
fds[0].fd = sock1;
fds[0].events = POLLIN;

// Monitor sock2 for output
fds[1].fd = sock2;
fds[1].events = POLLOUT;

// Wait 10 seconds
int ret = poll( &fds, 2, 10000 );
```

```
if ( ret == -1 )
{ // report error and abort }
else if ( ret == 0 )
{ // timeout; no event detected }
else {
  // If we detect the event,
  // zero it out so we can
  // reuse the structure
  if ( pfd[0].revents & POLLIN ) {
    pfd[0].revents = 0;
    // input event on sock1
  }

  if ( pfd[1].revents & POLLOUT ) {
    pfd[1].revents = 0;
    // output event on sock2
  }
}
```

poll的特点

- 支持多种事件：POLLIN、POLLPRI、POLLOUT、POLLERR、POLLHUP
- 解决了select中FD个数限制、频繁初始化FD的问题
- 提供了更精细的事件控制

- **没有**解决线性探测耗时的问题

epoll函数

```
// Create the epoll descriptor. Only one is
// needed per app, and is used to monitor all
// sockets.  The function argument is ignored
// (it was not before, but now it is), so put
// your favorite number here
int pollingfd = epoll_create( 0xCAFE );

if ( pollingfd < 0 ) {
    // report error
}

// Initialize the epoll structure in case
// more members are added in future
struct epoll_event ev = { 0 };

// Associate the connection class instance
// with the event. You can associate anything
// you want, epoll does not use this
// information. We store a connection class
// pointer, pConnection1
ev.data.ptr = pConnection1;

// Monitor for input, and do not
// automatically rearm the descriptor after
// the event
ev.events = EPOLLIN | EPOLLONESHOT;
```

```
// Add the descriptor into the monitoring
// list. We can do it even if another thread
// is waiting in epoll_wait - the descriptor
// will be properly added
if ( epoll_ctl( epollfd, EPOLL_CTL_ADD,
 pConnection1->getSocket(), &ev ) != 0 ) {
    // report error
}

// Wait for up to 20 events (assuming we
// have added maybe 200 sockets before
// that it may happen)
struct epoll_event pevents[ 20 ];

// Wait for 10 seconds
int ready = epoll_wait( pollingfd, pevents, 20, 10000 );
// Check if epoll actually succeed
if ( ret == -1 ) {
    // report error and abort
}
else if ( ret == 0 ) {
    // timeout; no event detected
}
else
{
    // Check if any events detected
    for ( int i = 0; i < ret; i++ )
    {
        if ( pevents[i].events & EPOLLIN )
        {
            // Get back our connection pointer
            Connection * c = (Connection*) pevents[i].data.ptr;
            c->handleReadEvent();
        }
    }
}
```

epoll的特点

- 只返回有变化的FD，解决线性探测效率问题
- 可以附加任意数据，不用二次查询FD等
- 支持多线程

- epoll并非"更好的poll"
  - 使用epoll_ctl系统调用修改FD或flag效率低
  - 惊群问题
  - 复杂调用下，更难debug
  - Linux特有，Unix需要使用kqueue

如何选择select/poll/epoll

- select——简单做做（<100个FD），或者被迫
- poll
  - 单线程
  - 较少FD（<1000）
  - 套接字生命期短，或方向变化剧烈（FD、flag变化多）
  - 有一定跨平台需求
- epoll
  - 多线程
  - 边缘触发
  - 较多FD（>1000）

更普适的方法——libevent、libuv

- 跨平台，不同平台可选用不同（最优）模型
- 使用统一接口对文件、套接字、信号等不同资源编程
- 提供更多扩展能力
- 避免自己造轮子时引入的bug

libevent echo server

```
static void
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx)
{
        /* We got a new connection! Set up a bufferevent for it. */
        struct event_base *base = evconnlistener_get_base(listener);
        uct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

        ferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);

        ferevent_enable(bev, EV_READ|EV_WRITE);
```

```
/* Clear the sockaddr before using it, in case there are extra
 * platform-specific fields that can mess us up. */
memset(&sin, 0, sizeof(sin));
/* This is an INET address */
sin.sin_family = AF_INET;
/* Listen on 0.0.0.0 */
sin.sin_addr.s_addr = htonl(0);
/* Listen on the given port. */
sin.sin_port = htons(port);

listener = evconnlistener_new_bind(base, accept_conn_cb, NULL,
    LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
    (struct sockaddr*)&sin, sizeof(sin));
if (!listener) {
        perror("Couldn't create listener");
        return 1;
}
evconnlistener_set_error_cb(listener, accept_error_cb);

event_base_dispatch(base);
```

见下页

```
static void
accept_error_cb(struct evconnlistener *listener, void *ctx)
{
        struct event_base *base = evconnlistener_get_base(listener);
        int err = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "Got an error %d (%s) on the listener. "
                "Shutting down.\n", err, evutil_socket_error_to_string(err));

        event_base_loopexit(base, NULL);
}
```

libevent echo server（续）

```
static void
echo_read_cb(struct bufferevent *bev, void *ctx)
{
        /* This callback is invoked when there is data to read on bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output buffer. */
        evbuffer_add_buffer(output, input);
}

static void
echo_event_cb(struct bufferevent *bev, short events, void *ctx)
{
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
}
```

libuv echo server

```
int main() {
    loop = uv_default_loop();

    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    uv_ip4_addr("0.0.0.0", 7000, &addr);

    uv_tcp_bind(&server, (const struct sockaddr*)&addr, 0);
    int r = uv_listen((uv_stream_t*)&server, 128, on_new_connection);
    if (r) {
        fprintf(stderr, "Listen error %s\n", uv_strerror(r));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status < 0) {
        fprintf(stderr, "New connection error %s\n", uv_strerror(status));
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*)malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*)client, alloc_buffer, echo_read);
    } else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

```
void alloc_buffer(uv_handle_t *handle, size_t suggested_size, uv_buf_t *buf) {
    buf->base = (char*)malloc(suggested_size);
    buf->len = suggested_size;
}
```

见下页

libuv echo server（续）

```c
void echo_read(uv_stream_t *client, ssize_t nread, const uv_buf_t *buf) {
    if (nread < 0) {
        if (nread != UV_EOF) {
            fprintf(stderr, "Read error %s\n", uv_err_name(nread));
            uv_close((uv_handle_t*) client, NULL);
        }
    } else if (nread > 0) {
        uv_write_t *req = (uv_write_t *) malloc(sizeof(uv_write_t));
        uv_buf_t wrbuf = uv_buf_init(buf->base, nread);
        uv_write(req, client, &wrbuf, 1, echo_write);
    }

    if (buf->base) {
        free(buf->base);
    }
}
```

```c
void echo_write(uv_write_t *req, int status) {
    if (status) {
        fprintf(stderr, "Write error %s\n", uv_strerror(status));
    }
    free(req);
}
```

各异步库的使用者比较

- libevent
- libuv

各异步库的功能比较

特性	libevent	libev	libuv
优先级	激活的事件组织在优先级队列中，各类事件默认的优先级是相同的，可以通过设置事件的优先级使其优先被处理	也是通过优先级队列来管理激活的时间，也可以设置事件优先级	没有优先级概念，按照固定的顺序访问各类事件
事件循环	event_base用于管理事件	激活的事件组织在优先级队列中，各类事件默认的优先级是相同的，可以通过设置事件的优先级使其优先被处理	
线程安全	event_base和loop都不是线程安全的，一个event_base或loop实例只能在用户的一个线程内访问（一般是主线程），注册到event_base或者loop的event都是串行访问的，即每个执行过程中，会按照优先级顺序访问已经激活的事件，执行其回调函数。所以在仅使用一个event_base或loop的情况下，回调函数的执行不存在并行关系		

各异步库的操作系统支持比较

type	libevent	libev	libuv
dev/poll (Solaris)	y	y	y
event ports	y	y	y
kqueue (*BSD)	y	y	y
POSIX select	y	y	y
Windows select	y	y	y
Windows IOCP	y	N	y
poll	y	y	y
epoll	y	y	y

套接字选项

获取或修改套接字选项

- getsockopt和setsockopt函数
- fcntl
- ioctl

操　作	fcntl	ioctl	路由套接字	POSIX
设置套接字为非阻塞式I/O型	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
设置套接字为信号驱动式I/O型	F_SETFL, O_ASYNC	FIOASYNC		fcntl
设置套接字属主	F_SETOWN	SIOCSPGRP或 FIOSETOWN		fcntl
获取套接字属主	F_GETOWN	SIOCGPGRP或 FIOGETOWN		fcntl
获取套接字接收缓冲区中的字节数		FIONREAD		
测试套接字是否处于带外标志		SIOCATMARK		sockatmark
获取接口列表		SIOCGIFCONF	sysctl	
接口操作		SIOC[GS]IFxxx		
ARP高速缓存操作		SIOCxARP	RTM_xxx	
路由表操作		SIOCxxxRT	RTM_xxx	

获取或修改套接字选项

- getsockopt和setsockopt函数
  - int getsockopt(int *socket*, **int** *level*, **int** *option_name,* void *restrict *option_value*, **socklen_t *restrict** *option_len***);**
  - int setsockopt(int *socket*, **int** *level*, **int** *option_name*, const void **option_value*, **socklen_t** *option_len***);**
- fcntl
  - int fcntl(int *fildes*, **int** *cmd*, **...);**
- ioctl
  - int ioctl(int *fildes*, **int** *request*, **... /* arg */);**

获取或修改套接字选项

```
int optval = 1;

setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, &optval,
sizeof(optval));
```

接入套接字选项的继承

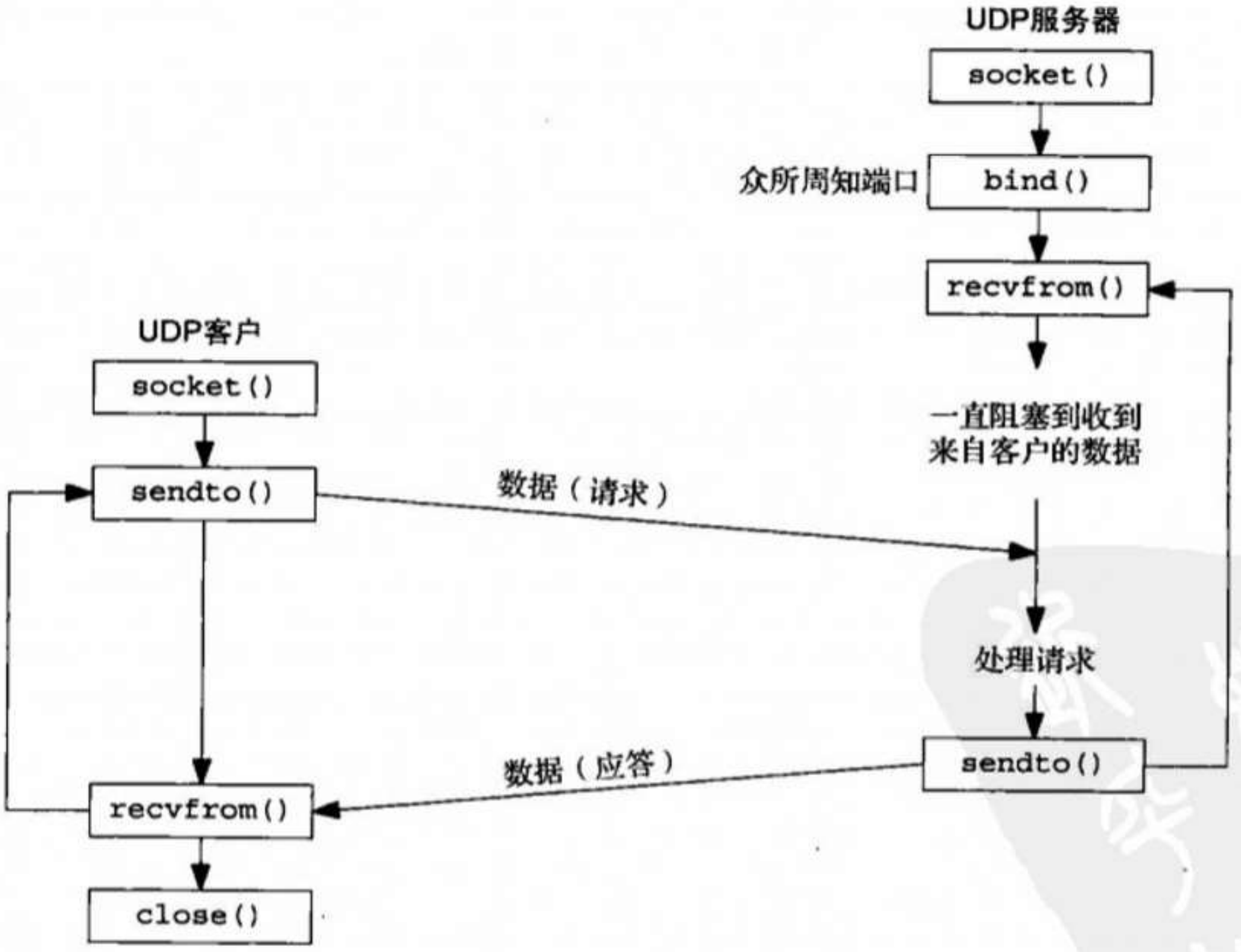
- 对于accept返回的套接字，其已经完成了三次握手，部分选项继承自对应的监听套接字
  - SO_DEBUG
  - SO_DONTROUTE
  - SO_KEEPALIVE
  - SO_LINGER
  - SO_OOBINLINE
  - SO_RCVBUF
  - SO_RCVLOWAT
  - SO_SNDBUF
  - SO_SNDLOWAT
  - TCP_MAXSEG
  - TCP_NODELAY

常用套接字选项

- 非阻塞套接字
- SO_KEEPALIVE——两小时无数据收发时，发送探测分组
- SO_RCVBUF/SO_SNDBUF——收发缓冲区大小设置
- SO_REUSEADDR——允许同一端口bind不同的套接字
- SO_LINGER——对close调用的精细控制

UDP套接字编程

UDP套接字编程


UDP服务器
socket()
众所周知端口
bind()
recvfrom()
一直阻塞到收到
来自客户的数据
处理请求
sendto()
UDP客户
socket()
sendto()
数据（请求）
recvfrom()
数据（应答）
close()

新建UDP套接字

- socket(AF_INET, SOCK_DRAM, 0)

recvfrom收取数据

- ssize_t recvfrom(int *socket*, **void *restrict** *buffer*, **size_t** *length*, int *flags*, **struct sockaddr *restrict** *address*, socklen_t *restrict *address_len*);

- 接收数据报及其发件人
- 也可以用于TCP，但是address_len会为0

sendto发送数据

- ssize_t sendto(int *socket*, **const void ****message*, **size_t** *length*, int *flags*, **const struct sockaddr ****dest_addr*, socklen_t *dest_len*);

UDP程序实例

- 参看程序udp-echo/echo-server.c、udp-echo/udp-hello.c

UDP的connect函数

- UDP无确认
- 因此服务器未运行（端口/服务不存在）时，发送端无法知晓，尽管系统可以收到表征此错误的ICMP包
- 在sendto之前调用connect可以获知此错误
- 此外，对连接后的UDP套接字
  - 可以使用write以省略sendto的地址部分
  - read、recv等只会返回来自连接IP的数据报
- 可以重复调用connect（和TCP不同）更换不同对端

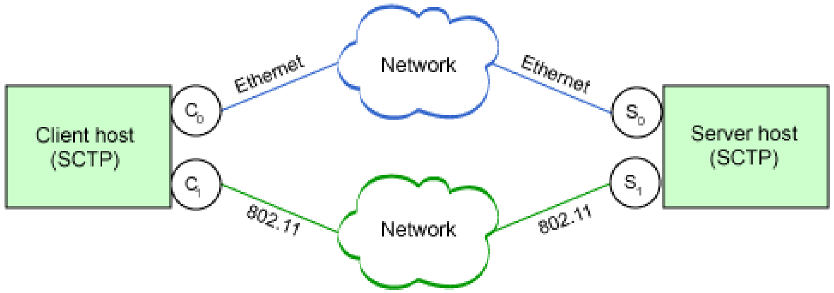
究竟用TCP还是UDP

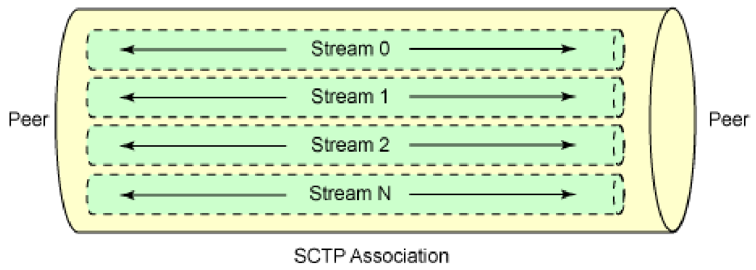
- 客户端主动发起间歇性的无状态的查询——HTTP/HTTPS over TCP
- 客户端和服务器都可以独立主动发包，但是偶尔发生延迟可以容忍——TCP
- 客户端和服务器都可以独立发包，而且无法忍受延迟——UDP
- 其他情况——relUDP或KCP等自定协议

SCTP协议编程

流控制传输协议（SCTP）简介

- 最早为在IP网内传输PSTN（SS7）信令而设计，因此也常用于电信领域（IMS、LTE）
- 面向成帧的消息
- 增加了冗余、多流等能力

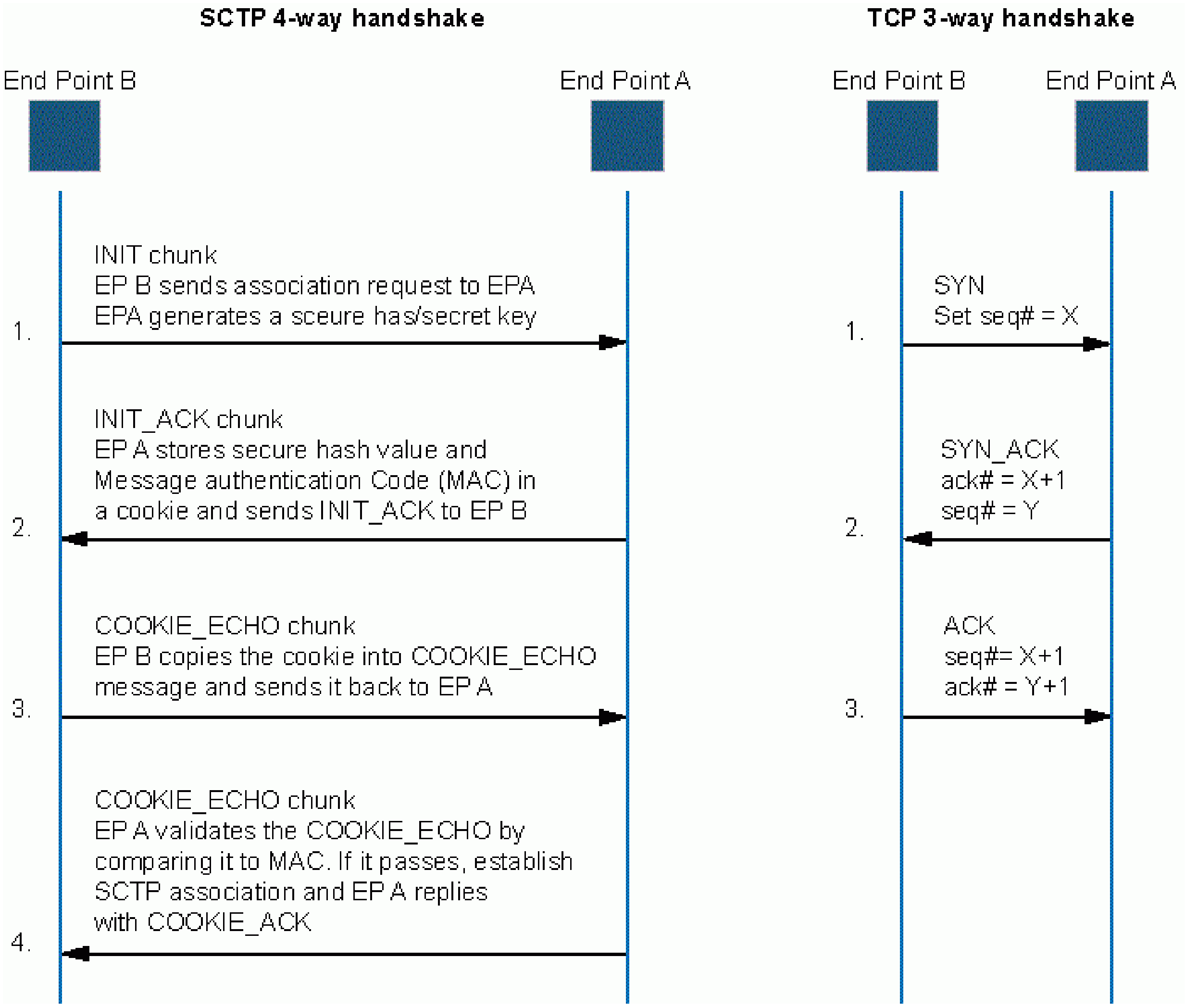




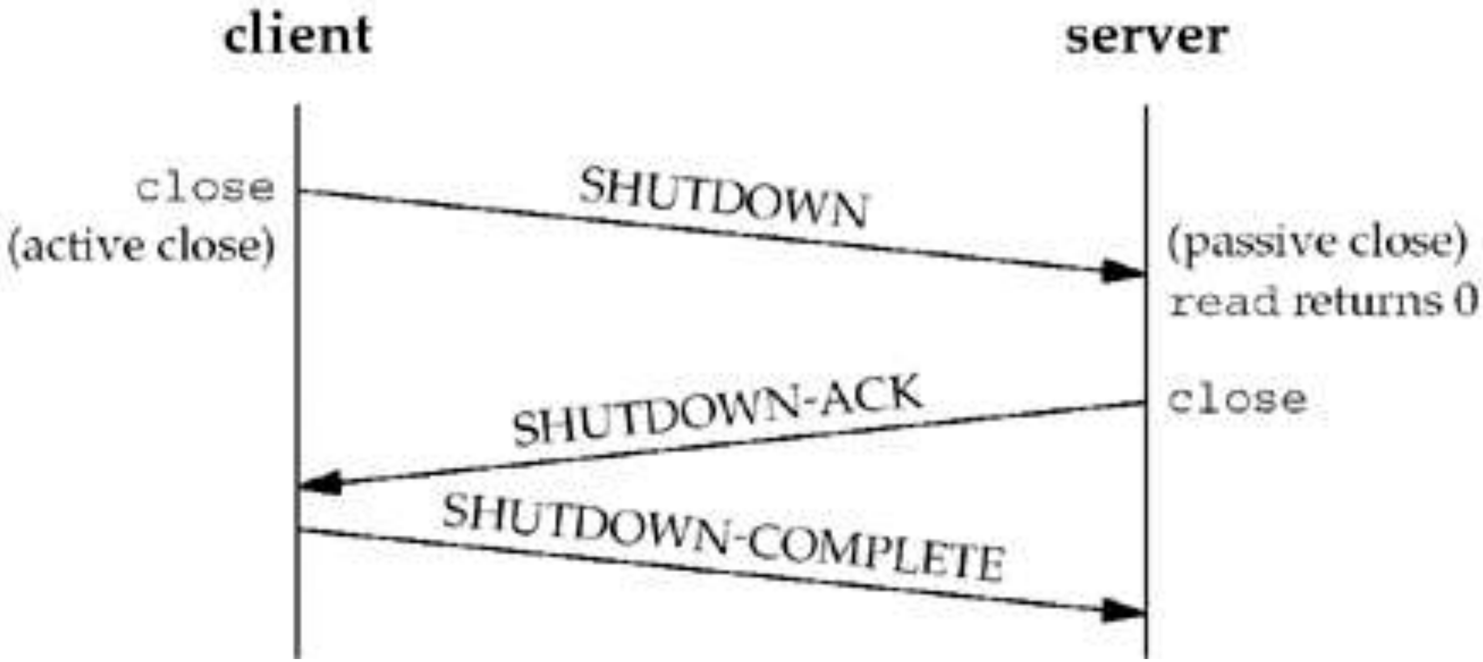
流控制传输协议（SCTP）简介

Bits	Bits 0 - 7	8 - 15	16 - 23	24 - 31
+0	Source port		Destination port	
32	Verification tag			
64	Checksum			
96	Chunk 1 type	Chunk 1 flags	Chunk 1 length	
128	Chunk 1 data			
...	...			
...	Chunk N type	Chunk N flags	Chunk N length	
...	Chunk N data			

SCTP 四次握手


SCTP 4-way handshake
TCP 3-way handshake
End Point B
End Point A
End Point B
End Point A
1.
INIT chunk
EP B sends association request to EPA
EPA generates a sceure has/secret key
2.
INIT_ACK chunk
EP A stores secure hash value and
Message authentication Code (MAC) in
a cookie and sends INIT_ACK to EP B
3.
COOKIE_ECHO chunk
EP B copies the cookie into COOKIE_ECHO
message and sends it back to EP A
4.
COOKIE_ECHO chunk
EP A validates the COOKIE_ECHO by
comparing it to MAC. If it passes, establish
SCTP association and EP A replies
with COOKIE_ACK
1.
SYN
Set seq# = X
2.
SYN_ACK
ack# = X+1
seq# = Y
3.
ACK
seq#= X+1
ack# = Y+1

SCTP 连接释放

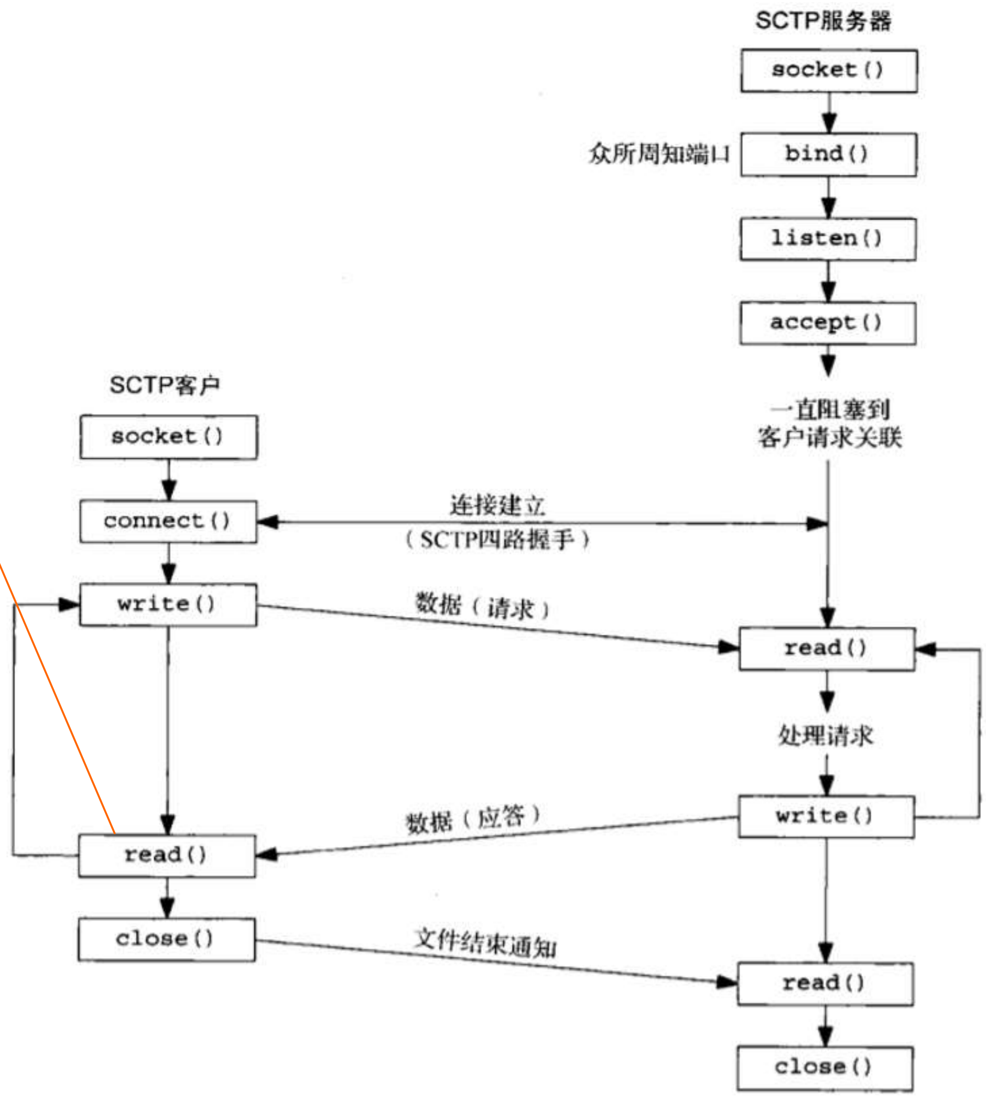

client
server
close
(active close)
SHUTDOWN
(passive close)
read returns 0
close
SHUTDOWN-ACK
SHUTDOWN-COMPLETE

三种传输层协议对比

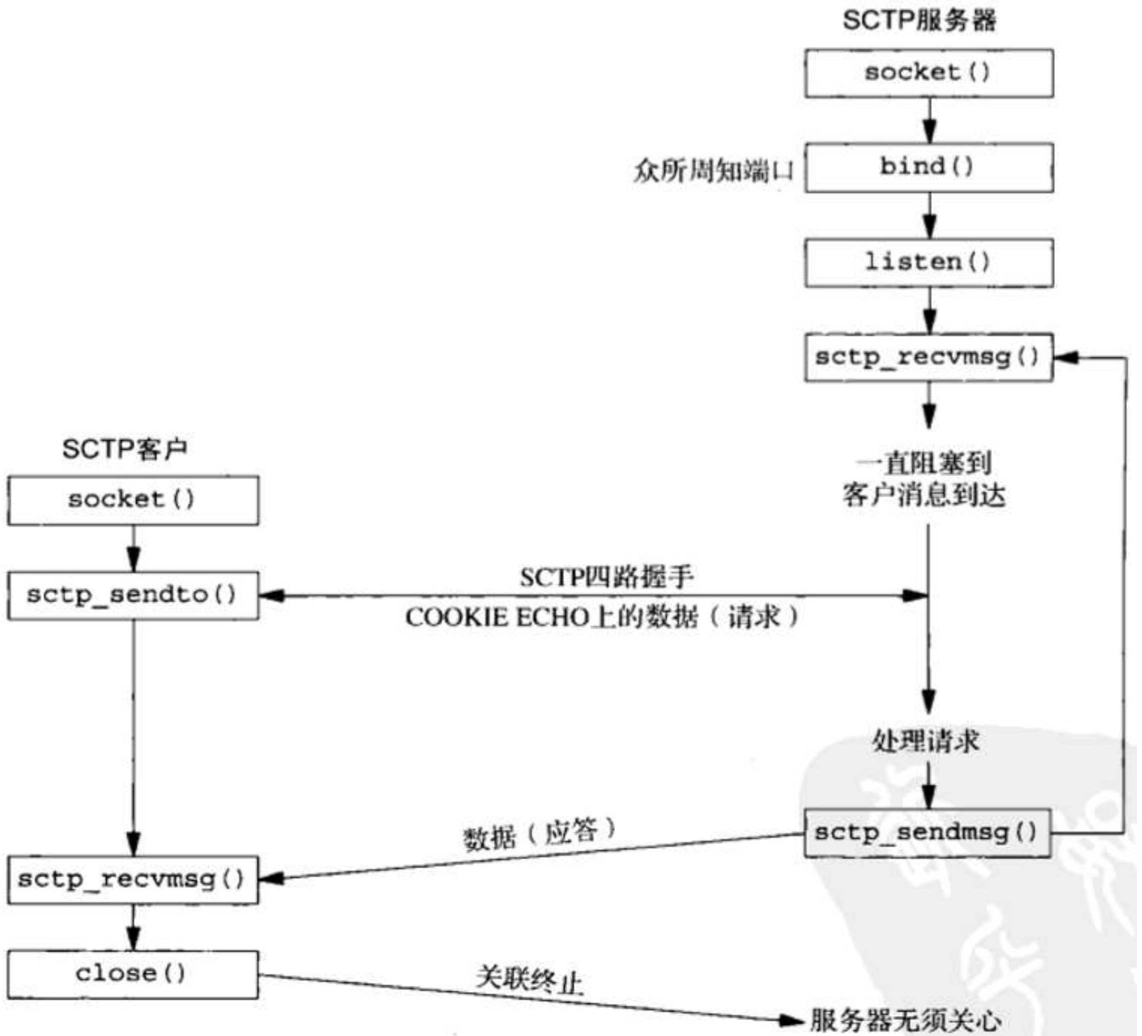
Attribute	TCP	UDP	SCTP
Reliability	Reliable	Unreliable	Reliable
Connection Management	Connection-orientated	Connectionless	Connection-orientated
Transmission	Byte-orientated	Message-orientated	Message-orientated
Flow Control	Yes	No	Yes
Congestion Control	Yes	No	Yes
Fault Tolerance	No	No	Yes
Data Delivery	Strictly Ordered	Unordered	Partially Ordered
Security	Yes	Yes	Improved

SCTP一对一形式

保留消息边界，
因此每次read总是
返回一个消息
（对应一次write）



SCTP一对多形式

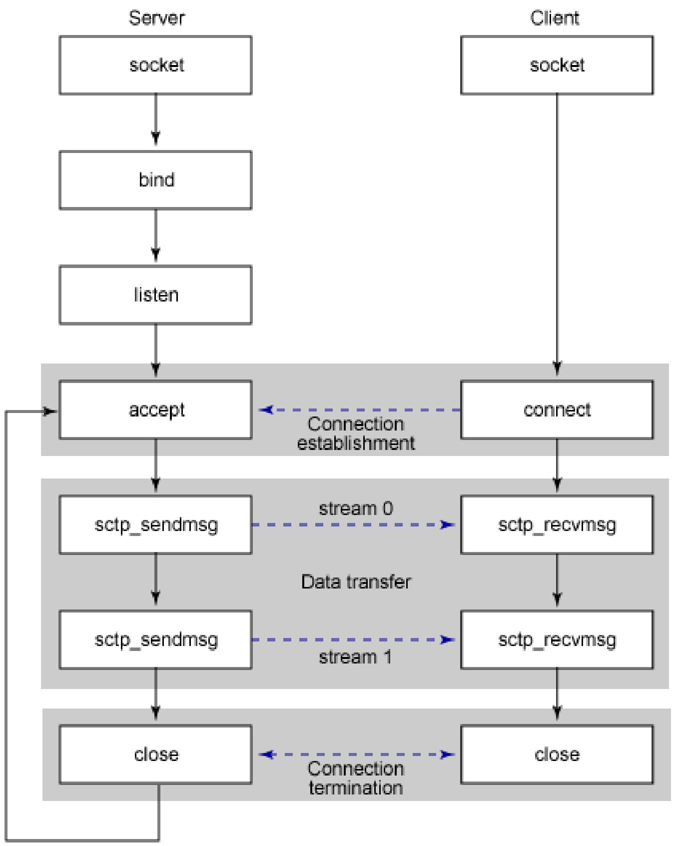


SCTP服务器
socket()
众所周知端口
bind()
listen()
sctp_recvmsg()
一直阻塞到
客户消息到达
SCTP客户
socket()
sctp_sendto()
SCTP四路握手
COOKIE ECHO上的数据（请求）
处理请求
数据（应答）
sctp_sendmsg()
sctp_recvmsg()
close()
关联终止
服务器无须关心

多流形式

```
int sctp_sendmsg(
    int sd,
    const void * msg,
    size_t len,
    struct sockaddr *to, socklen_t tolen,
    uint32_t ppid,
    uint32_t flags,
    uint16_t stream_no,
    uint32_t timetolive,
    uint32_t context
);

int sctp_recvmsg(
    int sd,
    void * msg,
    size_t len,
    struct sockaddr * from, socklen_t * fromlen,
    struct sctp_sndrcvinfo* sinfo,
    int * msg_flags
);
```

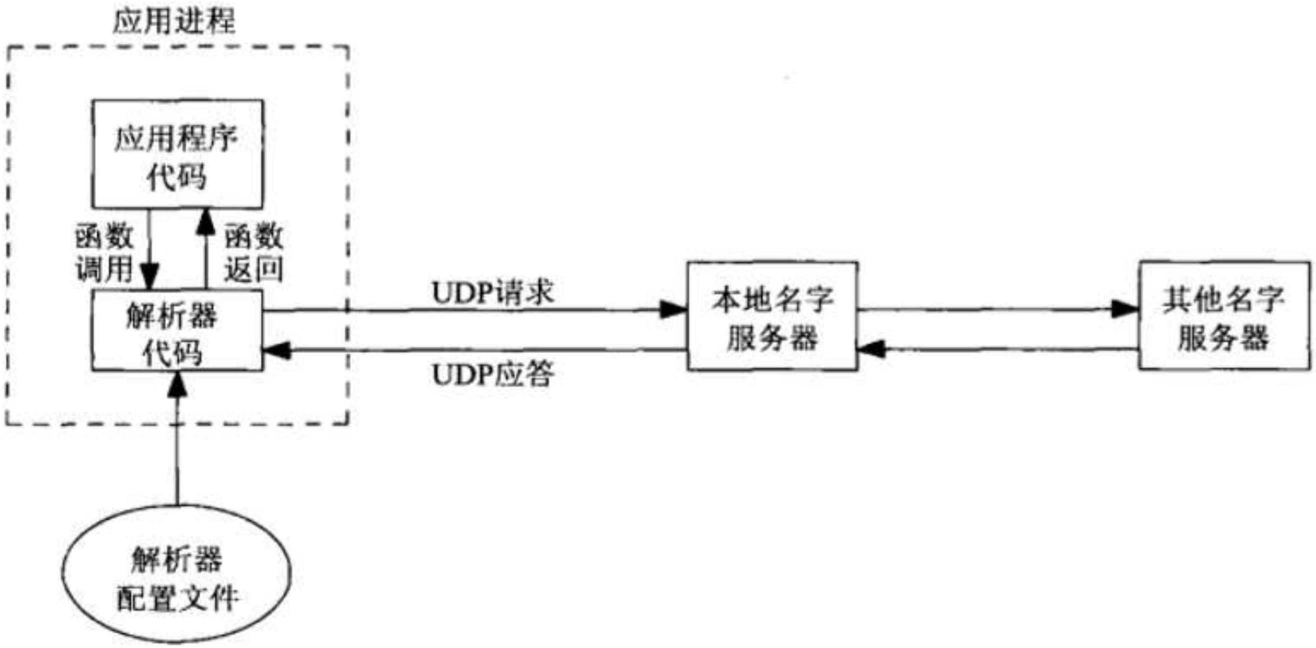


SCTP套接字其他特有编程接口

- int sctp_bindx(int sd, struct sockaddr * addrs, int addrcnt, int flags); // （动态地）绑定特定地址子集
- int sctp_connectx(int sd, struct sockaddr * addrs, int addrcnt, sctp_assoc_t * id); // 连接到一对多宿对端
- int sctp_peeloff(int sd, sctp_assoc_t assoc_id); // 把一个一到多关联转变为一到一关联

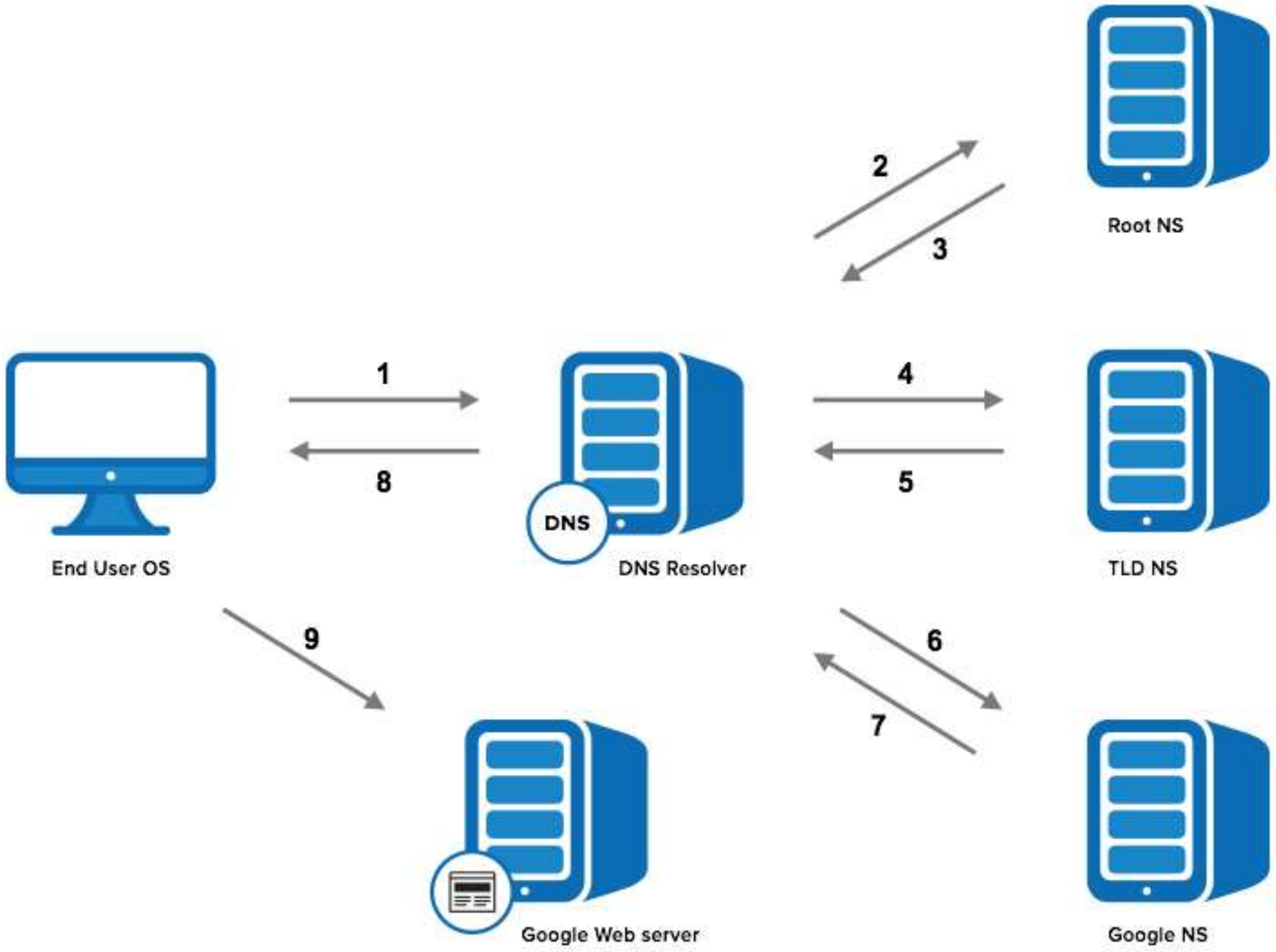
名字与地址转换

域名系统简介

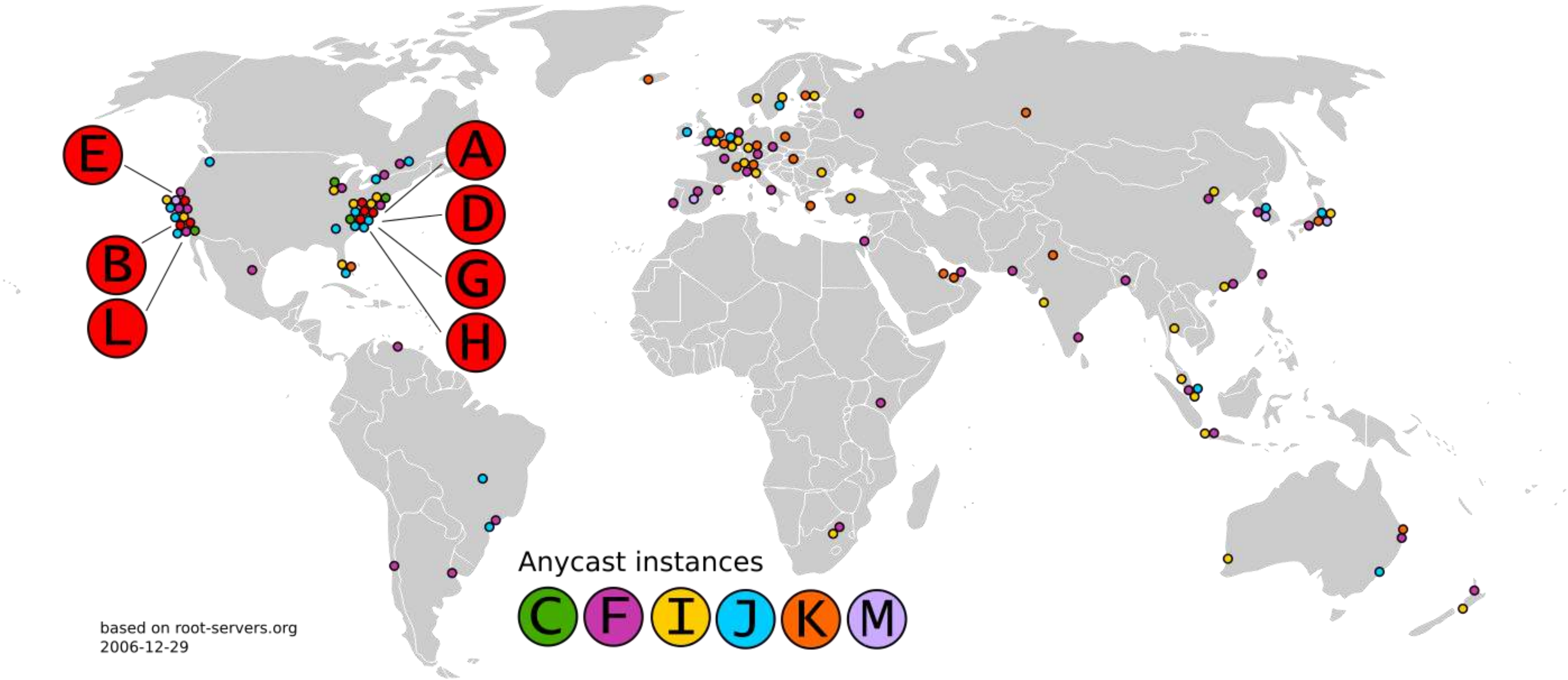


应用进程
应用程序代码
函数调用
函数返回
解析器代码
UDP请求
UDP应答
本地名字服务器
其他名字服务器
解析器配置文件

域名系统简介


2
3
Root NS
1
8
DNS
DNS Resolver
4
5
TLD NS
End User OS
9
6
7
Google Web server
Google NS

根域名服务器分布系统简介



DNS记录常见类型

Resource Record	Description
SOA (Start of Authority)	Indicates that the server is the best authoritative source for data concerning the zone. Each zone must have an SOA record, and only one SOA record can be in a zone.
NS (Name Server)	Identifies a DNS server functioning as an authority for the zone. Each DNS server in the zone (whether primary master or secondary) must be represented by an NS record.
A (Address)	Provides a name-to-address mapping that supplies an IPv4 address for a specific DNS name. This record type performs the primary function of the DNS: converting names to addresses
AAAA (Address)	Provides a name-to-address mapping that supplies an IPv6 address for a specific DNS name. This record type performs the primary function of the DNS: converting names to addresses.
PTR (Pointer)	Provides an address-to-name mapping that supplies a DNS name for a specific address in the in-addr.arpa domain. This is the functional opposite of an A record, used for reverse lookups only.
CNAME (Canonical Name)	Creates an alias that points to the canonical name (that is, the "real" name) of a host identified by an A record. Administrators use CNAME records to provide alternative names by which systems can be identified.
MX (Mail Exchange)	Identifies a system that will direct email traffic sent to an address in the domain to the individual recipient, a mail gateway, or another mail server.

名字-地址转换函数

- IPv4方法（同时是不可重入的方法）：
  - gethostbyname / gethostbyaddr
  - getservbyname / getservbyport
- 通用方法：
  - getaddrinfo + freeaddrinfo
  - getnameinfo

DNS API

- 参考程序dns-client/dns.c

未尽内容

- TCP Push
- Recvmsg/cmsg
- Fast Open实现

- zj0512@gmail.com

谢谢！