Windows Socket Modern System Calls & Concurrent, Connection-Oriented Servers

Prof. Lin Weiguo

Dec 2013

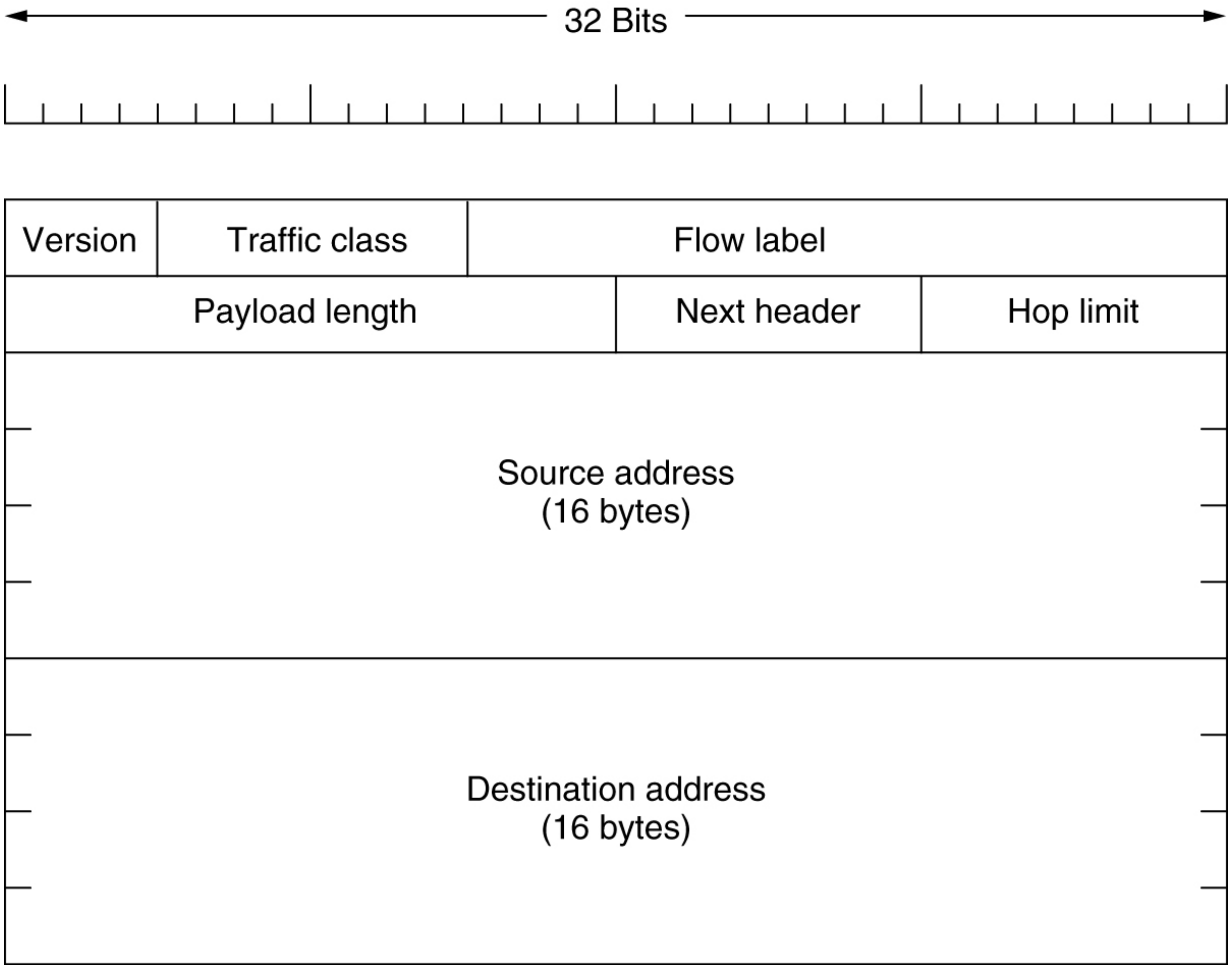
Note

- You should not assume that an example in this presentation is complete. Items may have been selected for illustration. It is best to get your code examples directly from the textbook and modify them to work. Use the lectures to understand the general principles.

Outline

- Windows Socket Modern System Calls
- Concurrent, Connection-Oriented Servers

IP v6 (RFC 2460)



32 Bits

Version

Traffic class

Flow label

Payload length

Next header

Hop limit

Source address
(16 bytes)

Destination address
(16 bytes)

The IPv6 fixed header

Winsock Headers and Libraries

- Include files: winsock2.h
- Library files: ws2_32.lib

```
//winsock 2.2
#include <winsock2.h>
//TCP/IP specific extensions in Windows Sockets 2
#include <ws2tcpip.h>

#pragma comment(lib, " Ws2_32.lib ")
```

IPv6 Address Structure

- 28-byte sockaddr in6

```
// (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4)
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family
    // all this is padding, implementation specific, ignore it: cast it to a struct sockaddr_in or  sockaddr_in6
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

InetPton Function

- The InetPton function is supported on Windows Vista and later.
- The InetPton function provides a protocol-independent conversion of an IP address in its standard text presentation form into its numeric binary form.

```
int InetPton (
    INT   Family,                     //[in] The address family.
    PCTSTR pszAddrString,    //[in] NULL-terminated string
    PVOID pAddrBuf   //[out] buffer in which to store the numeric IP address
);
```

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6
InetPton(AF_INET, "192.0.2.1", &(sa.sin_addr)); // IPv4
InetPton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

InetNtop function

- provides a protocol-independent address-to-string

```
Int InetNtop(
     INT   Family,          //[in] The address family.
     PVOID pAddr,           //[in] IP address in network byte to convert to a string
     PTSTR pStringBuf //[out] a buffer in which to store the NULL-terminated string
                            //representation of the IP address
     size_t StringBufSize //[in] the length of the pStringBuf.
);
```

```
char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something
InetNtop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
printf("The IPv4 address is: %s\n", ip4);

char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something
InetNtop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);
printf("The address is: %s\n", ip6);
```

Prep the socket address structures for connection

- Prepare for a client who wants to connect to a particular server, say "www.microsoft.com" port 3490.

```
char *hostname= "www.microsoft.com";
char *port = "3490";
struct addrinfo *result = NULL;
struct addrinfo *ptr = NULL;
struct addrinfo hints;
// Setup the hints address info structure
// which is passed to the getaddrinfo() function
ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
getaddrinfo(hostname, port, &hints, &result);
// Retrieve each address and print out the hex bytes
 for(ptr=result; ptr != NULL ;ptr=ptr->ai_next)
{  ….}
freeaddrinfo(result);
```

struct addrinfo

- Used as parameters in getaddrinfo()
  - In each returned **addrinfo** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a **socket** or **WSASocket** function call. Also, the **ai_addr** member in each returned **addrinfo** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

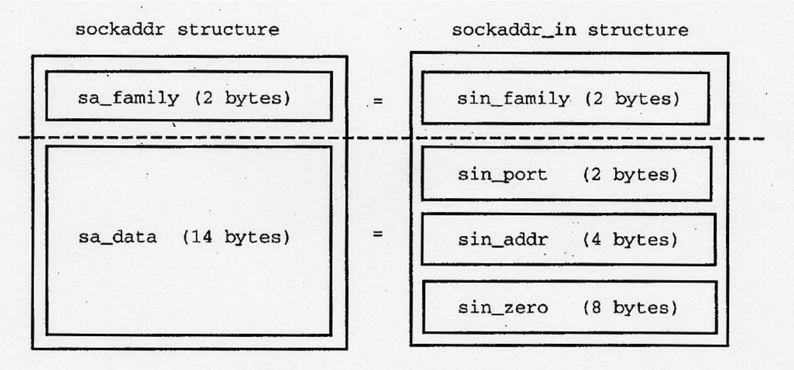
```
struct addrinfo {
    int ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol;  // use 0 for "any"
    size_t ai_addrlen;  // size of ai_addr in bytes
    struct sockaddr *ai_addr; // struct sockaddr_in or _in6
    char *ai_canonname;       // full canonical hostname
    struct addrinfo *ai_next; // linked list, next node
};
```

  - Header   : #include "ws2tcpip.h"

struct *sockaddr*

- The socket abstraction accommodates many protocol families.
  - It supports many address families(AF_INET (IPv4) or AF_INET6 (IPv6)).
  - It defines the following generic endpoint address:
    (address family, endpoint address in that family)
- Data type for generic endpoint address:

```
//cast to switch between the two types
struct sockaddr {
   unsigned short sa_family;    //type of address
   char sa_data[14];             //value of address
};
```



- *sockaddr* and *sockaddr_in* are compatible

getaddrinfo()

- The **getaddrinfo** function provides protocol-independent translation from an ANSI host name to an address.

```
int getaddrinfo(          //Success returns zero.
  __in   const char *nodename,   //a host (node) name or a numeric host address string
                                  // e.g. "www.example.com" or IP
  __in   const char *servname,   //a service name or port number represented as a string.
                                  // e.g. "http" or port number
  __in   const struct addrinfo *hints,    //hints about the type of socket the caller supports
  __out  struct addrinfo **res  //a linked list of one or more addrinfo structures that contains
                               //response information about the host
);
```

Example for a server

- Prepare for a server who wants to listen on your host's IP address, port 3490.

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results
memset(&hints, 0, sizeof(hints)); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me
if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}
// servinfo now points to a linked list of 1 or more struct addrinfos
// ... do everything until you don't need servinfo anymore ....
freeaddrinfo(servinfo); // free the linked-list
```

Show IP Address Example

```
struct addrinfo hints, *res, *p;    int status; char ipstr[INET6_ADDRSTRLEN];
if (argc != 2) { fprintf(stderr,"usage: showip hostname\n");  return 1;  }
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // AF_INET or AF_INET6 to force version
hints.ai_socktype = SOCK_STREAM;
if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
    return 2;
}
printf("IP addresses for %s:\n\n", argv[1]);
for(p = res;p != NULL; p = p->ai_next) {
    void *addr;
    char *ipver;
    if (p->ai_family == AF_INET) { // IPv4
        struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
        addr = &(ipv4->sin_addr);
        ipver = "IPv4";
    } else { // IPv6
        struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
        addr = &(ipv6->sin6_addr);
        ipver = "IPv6";
    }
    InetNtop(p->ai_family, addr, ipstr, sizeof ipstr); // convert the IP to a string
    printf("  %s: %s\n", ipver, ipstr);
}
freeaddrinfo(res); // free the linked list
```

socket()

```
SOCKET socket (
    int af,                //address family: AF_INET,AF_INET6
    int type,              // socket type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
    int protocol           // default 0
);
```

```
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do.)
// See the section on client/server for real examples.]

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

bind()

- binds the socket to the host, running on port 3490,

```
struct addrinfo hints, *res;
SOCKET m_hSocket;
// first, load up address structs with getaddrinfo():
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);
// make a socket:
m_hSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(m_hSocket, res->ai_addr, res->ai_addrlen);
```

Note: By using the *AI_PASSIVE* flag, I'm telling the program to bind to the IP of the host it's running on. If you want to bind to a specific local IP address, drop the AI_PASSIVE and put an IP address in for the first argument to getaddrinfo().

Address already in use?

- rerun a server and **bind()** fails,“Address already in use ”
  - a socket that was connected is still hanging around in the kernel， and it's hogging the port.
- You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
int yes=1;

// lose the pesky "Address already in use" error message
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == SOCKET_ERROR)
{
                printf( " setsockopt failed with error %d\n",WSAGetLastError());
}
```

Retrieves a socket option: getsockopt()

```
int getsockopt(
  __in     SOCKET s,    // a socket.
  __in     int level,        //The level at which the option is defined.
                              //Example: SOL_SOCKET, IPPROTO_TCP
  __in     int optname,  //The socket option for which the value is to be retrieved.
                              //Example: SO_ACCEPTCONN.
                              //The optname value must be a socket option defined within the
                              //specified level, or behavior is undefined.
  __out    char *optval, // buffer in which the value for the requested option is to be returned.
  __inout  int *optlen    // the size of the optval buffer.
);
```

- Example: get receiving buffer

```
int nErroCode;
int nBufLen;
int nOptlen=sizeof(nBuflen);
nErrCode=getsocketopt(s,SOL_SOCKET,SO_RCVBUF,(char *)&nBufLen,&nOptlen);
if  (nErrCode==SOCKET_ERROR)
{
}
```

Sets a socket option: setsockopt

```
int setsockopt(
  __in  SOCKET s,    // a socket.
  __in  int level,         // The level at which the option is defined
  __in  int optname,  // The socket option for which the value is to be set
                              //(for example, SO_BROADCAST)
  __in  const char *optval, // the buffer in which the value for the requested option is specified.
  __in  int optlen       // The size of the buffer pointed to by the optval parameter.
);
```

- Example: set receiving buffer

```
int nErroCode;
int nBufLen;
int nOptlen=sizeof(nBuflen);
nBufLen *=10;   //set the buffer 10 times of the original
nErrCode=setsocketopt(s,SOL_SOCKET,SO_RCVBUF,(char *)&nBufLen,&nOptlen);
if  (nErrCode==SOCKET_ERROR)
{
}
```

level = SOL_SOCKET

Value	Type	Meaning
SO_BROADCAST	BOOL	Enables transmission and receipt of broadcast messages on the socket.
SO_CONDITIONAL_ACCEPT	BOOL	Enables sockets to delay the acknowledgment of a connection until after the **WSAAccept** condition function is called.
SO_DONTLINGER	BOOL	Does not block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with **l_onoff** set to zero.
SO_DONTROUTE	BOOL	Disable routing: send directly to an interface. When this option is set, it succeeds but is ignored for both AF_INET and AF_INET6 sockets. This option is not supported on ATM sockets (results in an error).
SO_KEEPALIVE	BOOL	Sends keep-alives. Not supported on ATM sockets (results in an error).
SO_LINGER	**LINGER**	Lingers on close if unsent data is present.
SO_OOBINLINE	BOOL	Receives OOB data in the normal data stream. For a discussion of this topic, see Protocol Independent Out-Of-band Data.
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE and does not necessarily correspond to the size of the TCP receive window.
SO_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. For more information, see **bind**. Not applicable on ATM sockets.
SO_EXCLUSIVEADDRUSE	BOOL	Enables a socket to be bound for exclusive access. Does not require administrative privilege.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE and does not necessarily correspond to the size of a TCP send window..
SO_UPDATE_ACCEPT_CONTEXT	int	Updates the accepting socket with the context of the listening socket.

connect()

```
struct addrinfo hints, *res;
SOCKET m_hSocket;
// first, load up address structs with getaddrinfo():
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:
m_hSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!
connect(m_hSocket, res->ai_addr, res->ai_addrlen);
```

listen()

- The **listen** function places a socket in a state in which it is listening for an incoming connection.

```
//int listen(int sockfd, int backlog);
listen(m_hSocket,5);
```

- Note:backlog is the number of connections allowed on the incoming queue. Incoming connections are going to wait in this queue until you accept() them and this is the limit on how many can queue up. Most systems silently limit this number to about 20; you can probably get away with setting it to 5 or 10.

accept()

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
SOCKET  m_hListenSocket, m_hAcceptedSocket;
// !! don't forget your error checking for these calls !!
// first, load up address structs with getaddrinfo():
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me
getaddrinfo(NULL, MYPORT, &hints, &res);  // #define MYPORT "3490"
// make a socket, bind it, and listen on it:
m_hListenSocket= socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(m_hListenSocket, res->ai_addr, res->ai_addrlen);
listen(m_hListenSocket, BACKLOG);             // #define BACKLOG 10
// now accept an incoming connection:
addr_size = sizeof their_addr;
m_hAccepedSocket= accept(m_hListenSocket, (struct sockaddr *)&their_addr, &addr_size);
// ready to communicate on socket descriptor m_hAccepedSocket!
…
```

send() and recv()

- **send()** returns the number of bytes actually sent out
  - *this might be less than the number you told it to send!*
- **recv()** returns the number of bytes actually read into the buffer
  - **recv()** can return 0.This can mean the remote side has closed the connection on you!

```
//int send(SOCKET s, const void *buf, int len, int flags);
char *msg = “Hello,World!";
int len, bytes_sent;
…
len = strlen(msg);
bytes_sent = send(m_hSocket, msg, len, 0);
…



//int recv(SOCKET s,  void *buf, int len, int flags);
```

sendto() and recvfrom() for DGRAM

- **sendto()** returns the number of bytes actually sent

```
int sendto(SOCKET s, const void *buf, int len,  int flags, const struct sockaddr *to, int  tolen);
```

- **recvfrom()** returns the number of bytes received

```
int recvfrom(SOCKET s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen );
```

Note: to/from is a pointer to a remote/local struct sockaddr_storage that will be filled with the IP address and port of the originating machine. Tolen/fromlen is a pointer to a remote/local int that should be initialized to sizeof *to/from or sizeof(struct sockaddr_storage). When the function returns, tolen/fromlen will contain the length of the address actually stored in from.

closesocket() and shutdown()

- The **closesocket** function closes an existing socket.

```
int closesocket(SOCKET s );
```

- The **shutdown** function disables sends or receives on a socket.

```
int shutdown(SOCKET s, int how );
```

SD_SEND0	Shutdown send operations.
SD_RECEIVE1	Shutdown receive operations.
SD_BOTH2	Shutdown both send and receive operations.

getpeername, getsockname

- The **getpeername** function retrieves the address of the peer to which a socket is connected.

```
int getpeername(SOCKET s, struct sockaddr *name,  int *namelen );
```

Note: Once you have the address, you can use inet_ntop(), getnameinfo(), or gethostbyaddr() to print or get more information.

- The **getsockname** function retrieves the local name for a socket.

```
int getsockname(SOCKET s, struct sockaddr *name,  int *namelen
);
```

Concurrent, Connection-Oriented Servers

Ch11, Comers & Stevens, Volume III

Pros of Blocking

- **Easy to program -** Blocking is very easy to program. All user code can exist in one place, and in a sequential order.
- **Easy to port to Unix -** Since Unix uses blocking sockets, portable code can be written easily.
- **Work well in threads -** Since blocking sockets are sequential they are inherently encapsulated and therefore very easily used in threads.

Cons of Blocking

- **User Interface "Freeze" with clients**
  - Blocking socket calls do not return until they have accomplished their task. When such calls are made in the main thread of an application, the application cannot process the user interface messages. This causes the User Interface to "freeze" because the update, repaint and other messages cannot be processed until the blocking socket calls return control to the applications message processing loop.

Threading

- Threading is almost always used with blocking sockets. Non-blocking sockets can be threaded as well, but they require some extra handling and their advantages are lost with blocking sockets. Threading will be discussed briefly as it is important in writing blocking socket servers. Threading can also be used to write advanced blocking clients.

Threading Advantages

- **Prioritization -** Individual threads priorities can be adjusted. This allows individual server tasks or individual connections to be given more or less CPU time.
- **Encapsulation -** Each connection will be contained and less likely to interfere with other connections.
- **Security -** Each thread can have different security attributes.
- **Multiple Processors -** Threading automatically will take advantage of multiple processors.
- **No Serialization -** Threading provides true concurrency. Without threading all requests must be handled by a single thread. For this to work each task to be performed must be broken up into small pieces that can always execute quickly. If any task part blocks or takes time to execute all other task parts will be put on hold until it is complete. After each task part is complete, the next one is processed, etc. With threading, each task can be programmed as a complete task and the operating system will divide CPU time among the tasks.

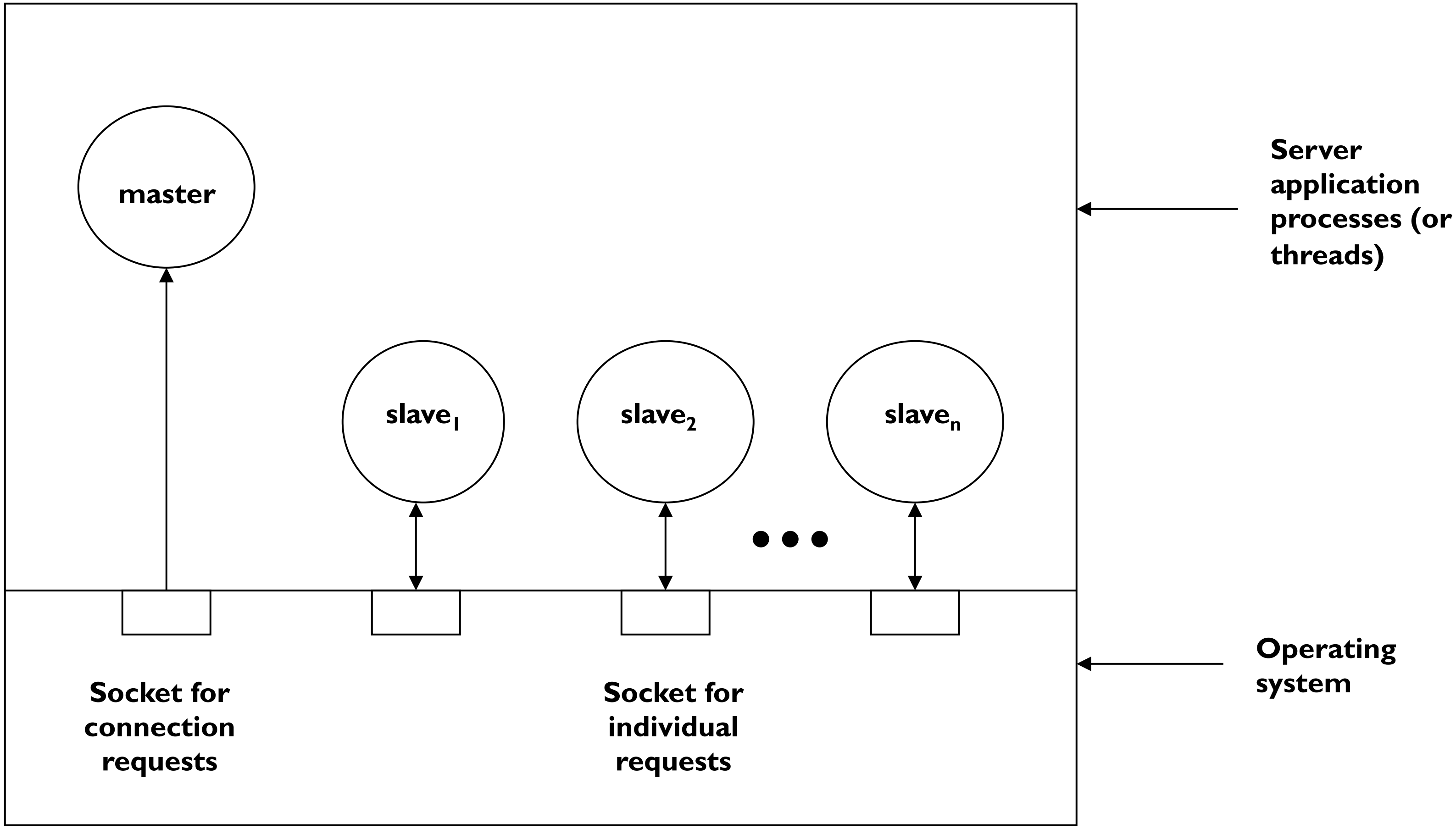
Thread Pooling

- Instead of creating and destroying threads on demand, threads are borrowed from a list of inactive but already created list (pool).
- When a thread is no longer needed it is redeposited into the pool instead of being destroyed. While threads are in the pool they are marked inactive and thus do not consume CPU cycles. For a further improvement, the size of the pool can be adjusted dynamically to meet the current needs of the system.

Iterative Vs. Concurrent Implementations

- Iterative Servers
  - Requires a client to wait while it handles prior requests
  - Long service requests (i.e. transferring a large file) could cause a excessive response delays
- Concurrent Servers
  - Prevents a single client from holding all resources
  - Allows communication with multiple clients simultaneously
  - Concurrent servers offer better response times than an iterative server

Process Structure


master
slave$_1$
slave$_2$
• • •
slave$_n$
Server application processes (or threads)
Operating system
Socket for connection requests
Socket for individual requests

The thread structure of a concurrent, connection-oriented server. A master server thread accepts each incoming connection, and creates a slave thread to handle it.

Echo Server: Thread Function Example

```
void tcpEchod(SOCKET fd){
        char buf[MAXLINE];
        int         n;
        n = recv(fd, buf, sizeof buf, 0);
        while (n != SOCKET_ERROR && n > 0) {
          if (send(fd, buf, n, 0) == SOCKET_ERROR) {
                fprintf(stderr, "echo send error: %d\n", WSAGetLastError());

                break;
          }
          n = recv(fd, buf, sizeof buf, 0);
        }
        if (n == SOCKET_ERROR)
          fprintf(stderr, "echo recv error: %d\n", WSAGetLastError());
        closesocket(fd);
}
```

Echo Server: Main Function Example

```
int main(int argc, char **argv)
{
    WSADATA wsadata;
    SOCKET listenfd, connfd;
    int clilen;
    struct sockaddr_in cliaddr, servaddr;
    if (WSAStartup(MAKEWORD(2,2), &wsadata) != 0)  errexit("WSAStartup failed\n");
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == INVALID_SOCKET)  errexit("cannot create socket: error number %d\n", WSAGetLastError());
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
    if (bind(listenfd, (SA *) &servaddr, sizeof(servaddr)) == SOCKET_ERROR)
          errexit("can't bind to port %d: error number %d\n", SERV_PORT, WSAGetLastError());
    if (listen(listenfd, 5) == SOCKET_ERROR)
          errexit("can't listen on port %d: error number %d\n", SERV_PORT, WSAGetLastError());
```

Echo Server: Main Function Example –cont.

```
    //…… cont.
    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (SA *) &cliaddr, &clilen);
        if (connfd == INVALID_SOCKET)
                errexit("accept failed: error number %d\n", WSAGetLastError());
        // create a service thread to handle this client
        if (_beginthread((void (*)(void *))tcpEchod, 0, (void *)connfd) < 0)
                errexit("_beginthread failed: error number %d\n", GetLastError());
    }
    return 1;
}
```

Echo Client: Sending Function Sample

```
void str_cli(SOCKET sockfd){
    char sendline[MAXLINE+1], recvline[MAXLINE+1];
    int         cc, n_out, n_in;
    while (fgets(sendline, MAXLINE, stdin) != NULL) {
            sendline[MAXLINE] = '\0';  /* ensure line null-termination  */
            n_out = strlen(sendline);
            // send out the data of n_out bytes
            send(sockfd, sendline, n_out, 0);

            // receive n_out bytes of data from server
            for (n_in = 0; n_in < n_out; n_in += cc)
            {
                    cc = recv(sockfd, &recvline[n_in], n_out - n_in, 0);
                    if (cc == SOCKET_ERROR)  errexit("socket recv failed: %d\n", WSAGetLastError());
            }
            recvline[n_in] = '\0’;
            fputs(recvline, stdout);
    }
    closesocket(sockfd);
}
```

Echo Client: Main Function Sample

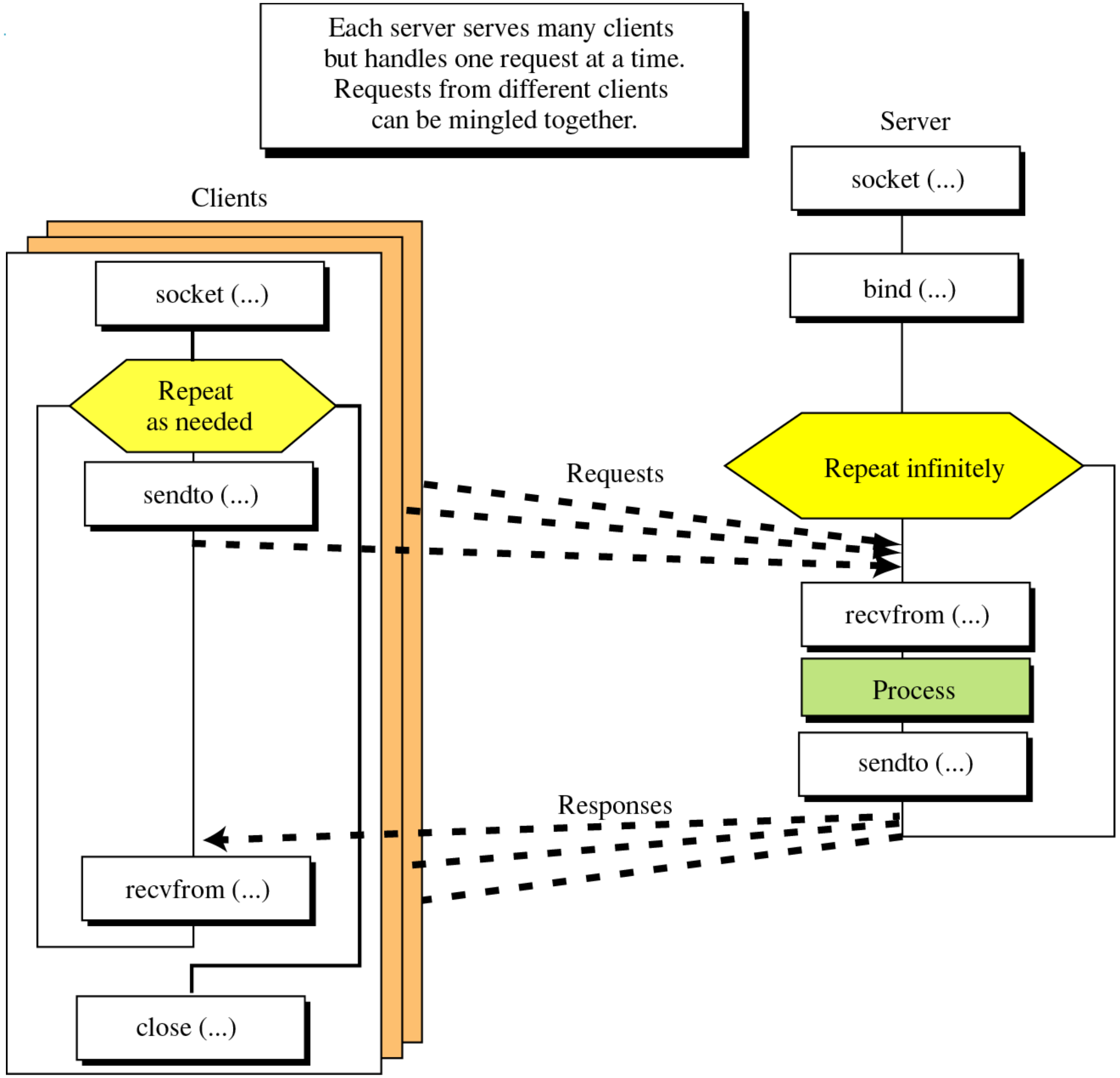
```
void main(int argc, char *argv[]){
        SOCKET sockfd;
        struct sockaddr_in servaddr;
        WSADATA     wsadata;
        if (WSAStartup(MAKEWORD(2,2), &wsadata) != 0)     errexit("WSAStartup failed\n");
        if (argc != 2)    errexit("usage: tcpEchoCli01 <IPaddress>");
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd == INVALID_SOCKET)
                errexit("cannot create socket: error number %d\n", WSAGetLastError());
        memset(&servaddr, 0, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        if ( (servaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE)
                errexit("inet_addr error: error number %d\n", WSAGetLastError());
        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
                errexit("connect error: error number %d\n", WSAGetLastError());
        str_cli(sockfd);
        WSACleanup();
        return 0;
}
```

References

- Ch11, Douglas Comer, David Stevens, *Internetworking With TCP/IP, Volume III*
- Chapter 28, Internet Essentials, Programming With Microsoft Visual C++ NET 6th Ed. – George/ Kruglinski Shepherd. 2002
- Network Programming for Microsoft Windows , 2nd Ed. – Anthony Jones, Jim Ohlund. 2002.
- Windows Sockets API Specification.
- MSDN: Windows Socket 2
- The Winsock Programmer's FAQ
- Beej's Guide to Network Programming
- Introduction to Indy: http://www.swissdelphicenter.ch/en/showarticle.php?id=4

Appendix
TCP Client/Server Interaction

Socket interface for Connectionless Iterative Server



Each server serves many clients but handles one request at a time. Requests from different clients can be mingled together.

TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family = AF_INET;                    /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);          /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
    DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);

    if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
        DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Later, a client decides to talk to the server…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
echoServAddr.sin_family      = AF_INET;             /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port        = htons(echoServPort); /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
        DieWithError("accept() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
   if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
       DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

TCP Client/Server Interaction

close(sock);

close(clntSocket)

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits

- Client must know the server’s address and port
- Server only needs to know its own port
- No corsendrelation between `send()` and `recv()`

Client	Server
(“Hello Bob”)	
	recv() -> “Hello ”
	recv() -> “Bob”
	send(“Hi ”)
	send(“Jane”)
recv() -> “Hi Jane”	

Closing a Connection

- close() used to delimit communication
- Analogous to EOF

Echo Client

```
send(string)

while (not received entire string)
    recv(buffer)
    print(buffer)

close(socket)
```

Echo Server

```
recv(buffer)
while(client has not closed connection)
    send(buffer)
    recv(buffer)

close(client socket)
```