Outline

- A quick example mapping from Unix to Windows
  - Based on threads.
- Windows Socket Programming Models
  - Winsock Background and Model
  - CAsyncSocket
  - CSocket
- References:
  - http://www.gamedev.net/reference/articles/article1297.asp
  - MSDN
    - ms-help://MS.MSDNQTR.2003FEB.1033/winsock/winsock/windows_sockets_start_page_2.htm
    - ms-help://MS.MSDNQTR.2003FEB.1033/vclib/html/_MFC_CAsyncSocket.htm
    - ms-help://MS.MSDNQTR.2003FEB.1033/vccore/html/_core_Windows_Sockets_Topics.htm

Concurrency Models

- The units of Concurrent Processing:
  - Unix: Processes
    - Pros:
      - It is more efficient for OS to do task scheduling.
      - The failure of forked processes will not cause the main process down.
    - Cons:
      - The overhead of making copies of processes.
  - Windows: Threads
    - Pros:
      - No overhead of making copies of processes.
    - Cons:
      - It is not efficient to do task scheduling inside threads.
      - The failure of threads will cause the whole process down.

Windows Threads

- `unsigned long _beginthread(`
  `void(__cdecl *start_address) (void *),`
  `unsigned stack_size,`
  `void *arglist)`
  - **`start_address`**: Start address of routine that begins execution of new thread;
  - **`stack_size`**: Stack size for new thread or 0;
  - **`arglist`**: Argument list to be passed to new thread or NULL, causes the program to start another thread and allows both to run at the same time.

A Simple Thread Example

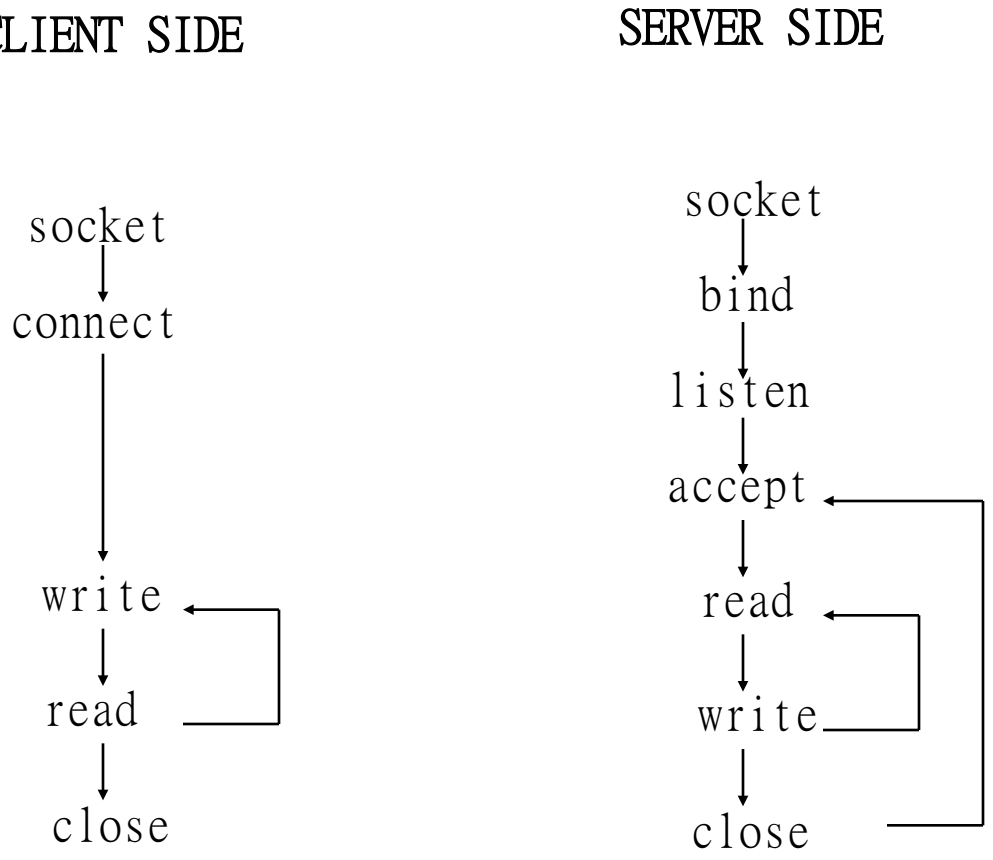
```
void main() {
  _beginthread((void(*)(void*)) addem,
                0, (void *)5);
  // create a new thread in Windows Sockets
  addem(5);
}

int addem(int count) {
  int I;
  for (i=1; i<count; i++) {
    printf("The value of i is %d\n", I);
    fflush(stdout);
  }
}
```

Process Example in Unix

```
void main() {
  pid = fork();
  // create a new thread in Windows Sockets
  addem(5);
}
int addem(int count) {
  int I;
  for (i=1; i<count; i++) {
    printf("The value of i is %d\n", I);
    fflush(stdout);
  }
}
```

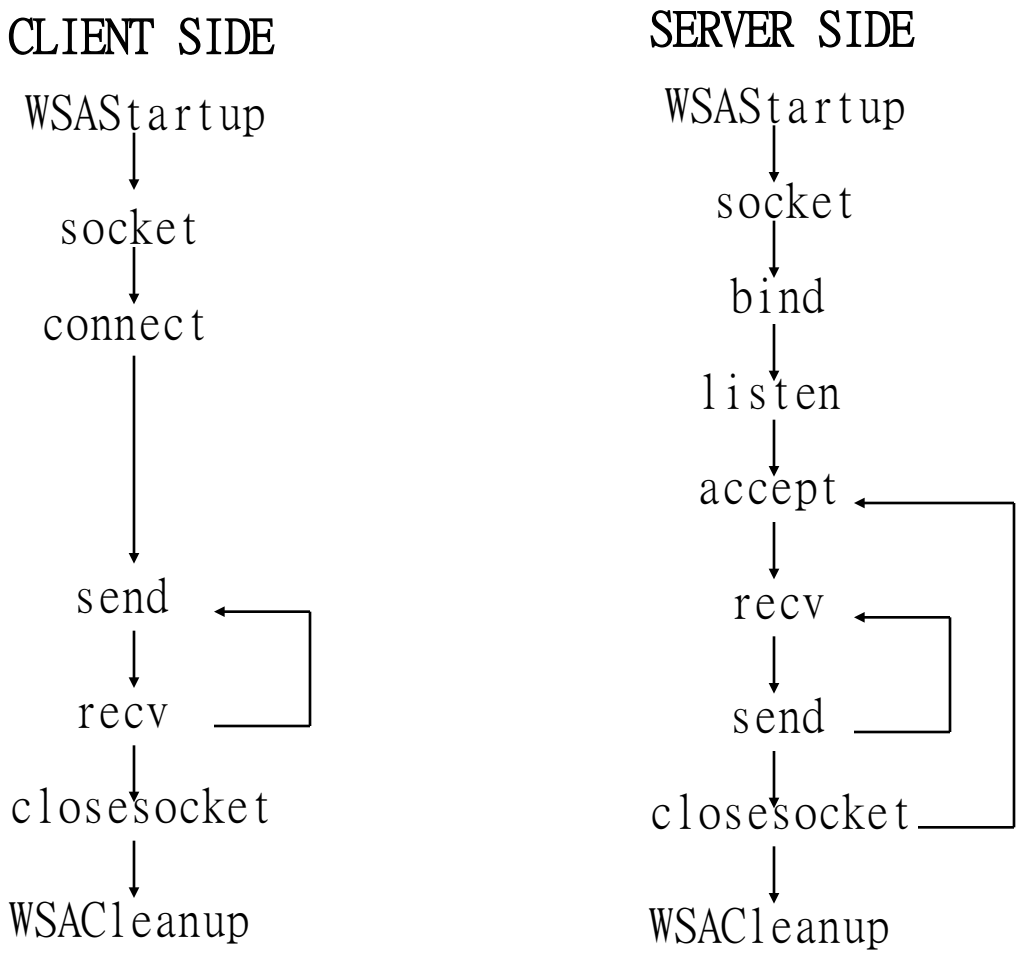
Socket Model

- Unix Sockets



WinSock Model

- Winsock



Concurrent TCP Servers (for Echo)

```
int main(int argc, char *argv[]) {
    char *service = "echo";      strcut socaddr_in fsin;          int alen;
    WSADATA wsadata;

    switch (argc) { … }
    if(WSAStartup(WSVERS,&wsadata)!=0)   errexit("WSAStartup failed\n");
    msock = passiveTCP(service, QLEN);

    While (1) {
        alen = sizeof(fsin);
        ssock = accept (msock, struct sockaddr *) &fsin, &alen);
        if (ssock == INVALID_SOCKET) errexit("accept:error number\n");
        if ( _beginthread((void (*) (void *)) TCPechod, STKSIZE, (void *) ssock) < 0)
            errexit ("_beginthread:%s\n", strerror (errno));
    }
    return 1;  /* not reached */
}
```

Concurrent TCP Servers (cont.)

```
void errexit (const char *format, …)
{
        ……
        WSACleanup ();
        exit (1);
}
SOCKET passiveTCP (const char *service, int qlen)
{
    return passivesock (service, “tcp”, qlen);
}
```

passivesock()

```
SOCKET passivesock ( const char *service, const char *transport, int qlen)
{
    ……
    /* Allocate a socket */
    s = socket (PF_INET, type, ppe->p_proto);
    if ( s == INVALID_SOCKET)
        errexit ( "can't create socket : %d\n", GetLastError ());
    /* Bind the socket */
    if (bind (s, (struct sockaddr *) &sin, sizeof (sin)) == SOCKET_ERROR)
        errexit ("can't bind to %s port: %d\n", service, GetLastError ());
    if (type == SOCK_STREAM && listen (s, qlen) == SOCKET_ERROR)
        errexit ("can't listen on %s port: %d\n", service, GetLastError ());
    return s;
}
```

TCPechod

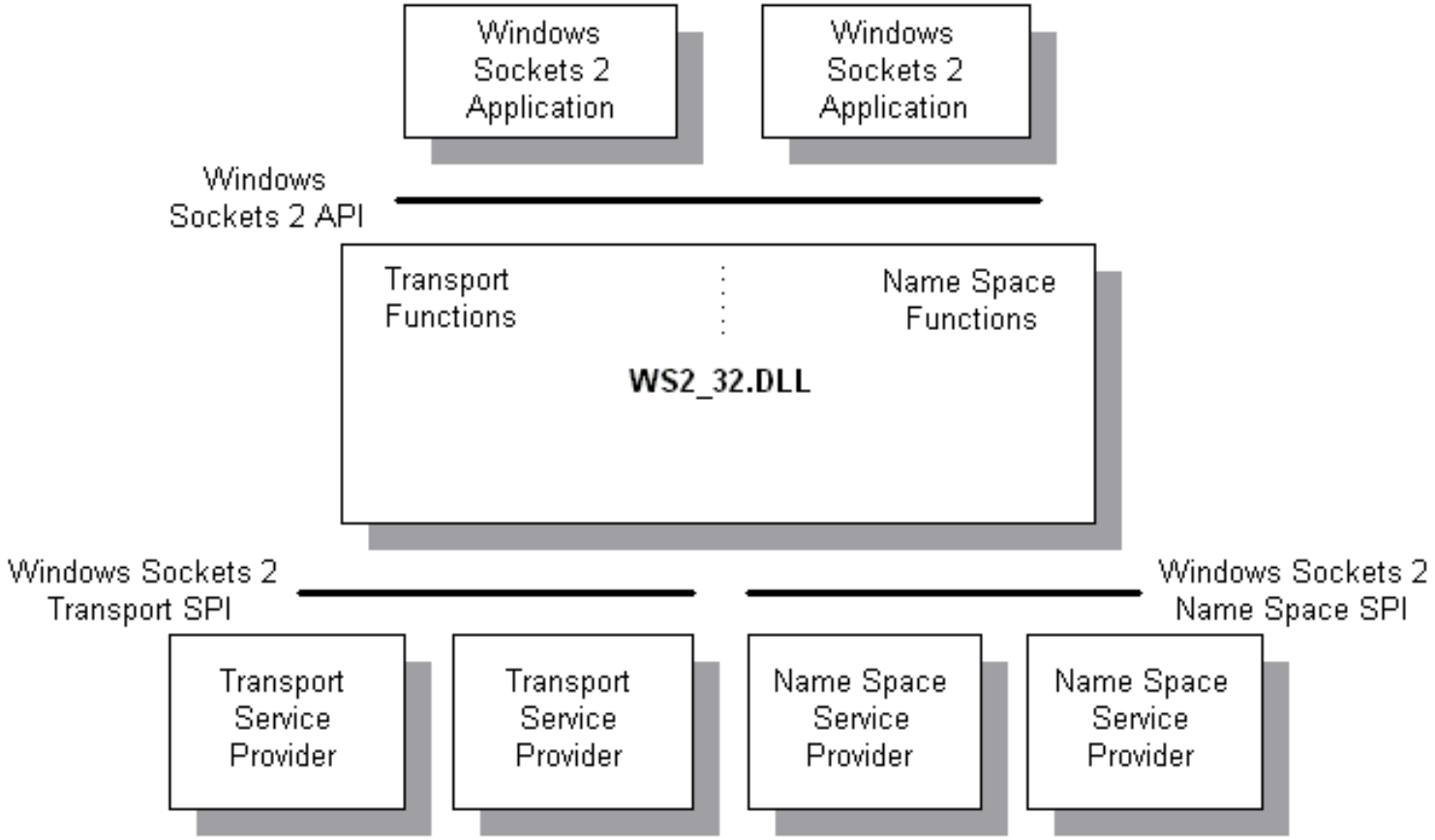
```
int TCPechod (SOCKET fd) {
    char buf [BUFSIZE] ;          int cc;

    cc = recv (fd, buf, sizeof buf, 0);
    while (cc != SOCKET_ERROR && cc > 0) {
        if (send (fd, buf, cc, 0) == SOCKET_ERROR) {
            fprintf ( stderr, "echo send error: %d\n", GetLastError ());
            break;
        }
        cc = recv (fd, buf, sizeof buf, 0);
    }
    if (cc == SOCKET_ERROR) fprintf( stderr, "echo recv error: %d\n", GetLastError ());
    closesocket (fd);
    return 0;
}
```

WinSock

- Windows Sockets (Winsock)
  - enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data
- Winsock follows the Windows Open System Architecture (WOSA) model;
  - Defines a standard service provider interface (SPI) between the application programming interface (API),
  - Uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX.
    - It was later adapted for Windows in Windows Sockets 1.1, with which Windows Sockets 2 applications are backward compatible.

Windows Sockets 2 Architecture

- Compliant with Windows Open System Architecture (WOSA)



Supported Protocols in WinSock

Protocol	Windows NT 4.0	Windows 2000	Windows XP	Windows Server 2003
TCP/IP	Supported	Supported	Supported	Supported
DLC	Supported	Supported	Not supported	Not supported
NetBEUI	Supported	Supported	Not supported	Not supported
IPX	Supported	Supported	Supported	Supported
TP4	Supported	Not supported	Not supported	Not supported
IPv6	Not supported	Not supported	Supported	Supported

Callback Paradigm in Windows SDK

- A common programming paradigm in Windows SDK
  - Queue of callback messages.
  - Sometimes, we call it *event-driven programming*.
- Problem:
  - Do not want to call blocking `read()/write()`.
  - Do not want to call blocking `select()`.
- Solutions:
  - Use asynchronous sockets.

- Note: Threading vs. Event-driven.
  - Read Ousterhout's slide.(http://home.pacbell.net/ouster/threads.pdf)

Asynchronous Select

```
int PASCAL FAR WSAAsyncSelect(SOCKET s,
  HWND hwnd, unsigned int wMsg, long lEvent)
```

- `s` – the socket instance for which we want to enable event notification.
- `hwnd` – identifies the window to which the messages will be posted, since WinSock uses `PostMessage`() to get the notification to you.
- `wMsg` – the message type for the notification message(s).
- `lEvent` – a bit mask identifying the events for which we want notification. Ex: `FD_READ` | `RD_WRITE` | `FD_CONNECT`.

Events

- `FD_READ`: This message indicates that you have just received data, and you must use the `recv`() function to get it.
- `FD_WRITE`: This means that there is room in the buffer to send, and we must send using the `send`() function.
- `FD_CONNECT`: Used only by the client program. This message indicates that the server has received the connection request and has accepted.
- `FD_ACCEPT`: Used only by the server program. This message indicates the receipt of the `connect`() request and you must send back an `accept`().
- `FD_CLOSE`: This message indicates that the other side's socket has shut down. This can be for several reasons.

Code

- Common Part:

```
// the message we'll use for our async notification
#define WM_WSAASYNC (WM_USER +5)
// create and test the socket
Port = socket(AF_INET, SOCK_STREAM, 0);
if (Port == INVALID_SOCKET)
  return(FALSE);
```

- Client Side:

```
// make the socket asynchronous and notify of read, write, connect and
   close events
WSAAsyncSelect(Port, hwnd, WM_WSAASYNC, FD_WRITE | FD_CONNECT |
                                        FD_READ | FD_CLOSE);
```

- Server Side:

```
// make the socket asynchronous and notify of read, write, accept and
   close events
// this is the server socket
WSAAsyncSelect(Port, hwnd, WM_WSAASYNC, FD_READ | FD_WRITE |
                                        FD_ACCEPT | FD_CLOSE);
```

Handling Notification Messages

- Notification Messages is really a misleading term.
  - In reality, WinSock only generates one message that it posts to the queue, and attached to it is the event that occurred.

```
switch(msg) {
  case WM_WSAASYNC:  {
    switch(WSAGETSELECTEVENT(lParam))
    {
      case FD_ACCEPT:  {
        // check for an error
        if (!WSAGETSELECTERROR(lParam))
          return(FALSE);
        // process message
        sAccept =  accept(wParam, NULL, NULL);
          return(0);
      }
    }
  }
}
```

Main Routine in Windows

- Windows main routine:

```
int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,   int       nCmdShow) {
   . . .
   // Perform application initialization: hInstance maps to hWndProc.
   if (!InitInstance (hInstance, nCmdShow))  return FALSE;
   . . .

   // Main message loop:
   while (GetMessage(&msg, NULL, 0, 0)) {
     if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
     }
   }

   //terminates use of the WS2_32.DLL
    WSACleanup();
}
```

Callback Functions in Windows

- Windows callback procedures:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
   LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;
   int iRet;

   switch (message) {
    case WM_COMMAND:
            . . .
    case WM_ASYNC_SELECT:
       // handling notification of messages. (See next Slides)
   }
}
```

Handling Notification Messages (cont.)

```
switch(WSAGETSELECTEVENT(lParam))
{
case FD_READ:
  {
  } break;
case FD_WRITE:
  {
  } break;
case FD_CONNECT:
//  or case FD_ACCEPT:
  {
  } break;
case FD_CLOSE:
  {
  } break;
}
```

WParam and LParam of WindProc

WParam	lParam	
	HIWORD WSAGETSELECTERROR()	LOWORD WSAGETSELECTEVENT()
socket instance	0 if successful, else an error code value that can be called for using WSAGetLastError()	WinSock event such as FD_READ, FD_WRITE, FD_CLOSE, etc.

Shaking Hands

- Client:

```
connect(Port, (LPSOCKADDR)&Server, sizeof(Server));
```

  - The **FD_CONNECT** event happens after the connection has been established
- Server:

```
...
case FD_ACCEPT:
  {
    // holds size of client struct
    int lenclient = sizeof(client_sock);
    // connect to the server
    Port = accept(wParam, &client_sock, &lenclient);
            // wParam is the sockd.
  } break;
...
```

  - Once **accept()** is called, if there is already another connection request, **FD_ACCEPT** is posted again immediately. If not, WinSock waits for one.

MFC Model

- A quicker programming development model,
  - usually with visualization.
- Based on the callback paradigm in SDK.
  - But, directly callback to methods of objects in MFC
- **`CAsyncSocket, CSocket`**:
  - MFC classes for handling sockets.

CAsyncSocket vs. CSocket

- CAsyncSocket
  - This class encapsulates the Windows Sockets API.
  - CAsyncSocket is for programmers who
    - know network programming and want the flexibility of programming directly to the sockets API
    - but also want the convenience of callback functions for notification of network events.
  - The only additional abstraction is converting certain socket-related Windows messages into callbacks.
- CSocket
  - This class, derived from CAsyncSocket, supplies a higher level abstraction for working with sockets through an MFC CArchive object.
  - CSocket manages many aspects of the communication that you would have to do yourself using either the raw API or class CAsyncSocket.
  - Most importantly, CSocket provides blocking, which is essential to the synchronous operation of CArchive.

Steps To Use CAsyncSocket

- Construct a CAsyncSocket object and use the object to create the underlying SOCKET handle.
  - Creation of a socket follows the MFC pattern of two-stage construction.
  - For example:

```
CAsyncSocket sock;
sock.Create( );    // Use the default parameters
CAsyncSocket* pSocket = new CAsyncSocket;
int nPort = 27;
pSocket->Create( nPort, SOCK_STREAM, FD_READ, “csie.nctu.edu.tw” );
```

- Connect together:
  - Client:
    - connect the socket object to a server socket, using CAsyncSocket::Connect.
  - Server:
    - Call CAsyncSocket::Listen for connect.
    - Accept it with CAsyncSocket::Accept.
      - The Accept member function takes a reference to a new, empty CSocket object as its parameter.
      - Do not call Create for this new socket object.

Steps To Use `CAsyncSocket` (cont.)

- Call the CAsyncSocket object's member functions
  - These functions encapsulate the Windows Sockets API functions.
    - See the next slide.
- Destroy the CAsyncSocket object.
  - If the created socket object is on the stack, its destructor is called when the containing function goes out of scope.
  - If you created the socket object on the heap, using the new operator, you are responsible for using the delete operator to destroy the object.

Supported Operations

- Accept : Accepts a connection on the socket.
- AsyncSelect : Requests event notification for the socket.
- Bind : Associates a local address with the socket.
- Close : Closes the socket.
- Connect : Establishes a connection to a peer socket.
- IOCtl : Controls the mode of the socket.
- Listen : Establishes a socket to listen for incoming connection requests.
- Receive : Receives data from the socket.
- ReceiveFrom : Receives a datagram and stores the source address.
- Send : Sends data to a connected socket.
- SendTo : Sends data to a specific destination.
- ShutDown : Disables **Send** and/or **Receive** calls on the socket.

Socket Notification

- `OnReceive`: Notifies this socket that there is data in the buffer for it to retrieve by calling Receive.
- `OnSend`: Notifies this socket that it can now send data by calling Send.
- `OnAccept`: Notifies this listening socket that it can accept pending connection requests by calling Accept.
- `OnConnect`: Notifies this connecting socket that its connection attempt completed: perhaps successfully or perhaps in error.
- `OnClose`: Notifies this socket that the socket it is connected to has closed.

➔

- If you derive from class CAsyncSocket, you must override the above functions for those network events of interest to your application.

Blocking

- A socket can be in "blocking mode" or "nonblocking mode."
  - Blocking: CSocket, or using CAsyncSocket with blocking.
  - Nonblocking: CAsyncSocket
    - the call read() returns immediately and the current error code, retrievable with the GetLastError member function, is WSAEWOULDBLOCK,
    - CSocket never returns WSAEWOULDBLOCK.
- The behavior of sockets is different:
  - 16-bit operating systems (such as Windows 3.1).
  - 32-bit operating systems (such as Windows 95 or Windows 98)
    - use preemptive multitasking and provide multithreading.

Sequence of Operations

Server	Client
// construct a socket CSocket sockSrvr;	// construct a socket CSocket sockClient;

// create the SOCKET
sockSrvr.Create(nPort);
	// create the SOCKET
	sockClient.Create( );[2]

// start listening
sockSrvr.Listen( );	

	// seek a connection
	sockClient.Connect(strAddr, nPort);

// construct a new, empty socket
CSocket sockRecv;
// accept connection
sockSrvr.Accept(sockRecv);[5]
	,[4]

Sequence of Operations (cont.)

Server	Client
// construct file object CSocketFile file(&sockRecv);	// construct file object CSocketFile file(&sockClient);

// construct an archive
CArchive
arIn(&file, CArchive::load);
-or-
CArchive
arOut(&file, CArchive::store);
– or Both –
	// construct an archive
	CArchive
	arIn(&file, CArchive::load);
	-or-
	CArchive
	arOut(&file, CArchive::store);
	– or Both –

// use the archive to pass data: arIn >> dwValue; -or- arOut << dwValue;[6]	// use the archive to pass data: arIn >> dwValue; -or- arOut << dwValue;[6]

Something to Notice

- When you call Accept on the server side, you pass a reference to a new socket object.
  - You must construct this object first, but do not call Create for it.
  - Keep in mind that if this socket object goes out of scope, the connection closes.
  - You can construct the socket on the stack, as shown, or on the heap.
- The archive and the socket file are closed when they go out of scope.
  - The socket object's destructor also calls the Close member function for the socket object when the object goes out of scope or is deleted.

Some More for Unix Socket Programmers

- Socket Data Type
- Error Code
- Maximum Number of Sockets Supported
- Renamed Functions

Socket Data Type

- Typical BSD style

```
s = socket(...);
if (s == -1)    /* or s < 0 */
    {...}
```

- Preferred style

```
s = socket(...);
if (s == INVALID_SOCKET)
    {...}
```

Error Code

- Typical BSD style

```
r = recv(...);
if (r == -1
    && errno == EWOULDBLOCK)
    {...}
```

- Preferred style

```
r = recv(...);
if (r == -1       /* (but see below) */
    && WSAGetLastError() == EWOULDBLOCK)
    {...}
```

- Better style

```
r = recv(...);
if (r == SOCKET_ERROR
    && WSAGetLastError() == WSAEWOULDBLOCK)
    {...}
```

Maximum Number of Sockets Supported

- The maximum number of sockets that an application can actually use is independent of the number of sockets supported by a particular implementation.
- The default value in Winsock2.h is 64.
- For more:
  - define FD_SETSIZE in every source file before including Winsock2.h.
  - The compiler options in the makefile “-DFD_SETSIZE=128"

Renamed Functions

- Close and Closesocket
  - Sockets must be closed by using the closesocket routine.
  - Using the close routine to close a socket is incorrect.
- Ioctl and Ioctlsocket/WSAIoctl