Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 5:
Advance Winsock Programming
(5 hrs)

Outline

1. Asynchronous I/O and Nonblocking operations in Winsock
   - Error handling functions
   - Using non blocking sockets
   - Select in conjuction with accept, select with recv/recvfrom and send/sendto
   - Overlapped I/O: Basics and implementation
   - Event-driven programming
     - Using WSAEventSelect()
     - Completion routines

2. Winsock Extensions
   - Advanced polling mechanisms: WSAPoll()
   - Event management with WSAEventSelect()

3. Implementation of basic cross platform application

Asynchronous I/O and Nonblocking operations in Winsock

Asynchronous IO and Non-blocking operation

- Windows' socket API (WinSock 2.2 and later) supports two primary models for avoiding thread-blocking on network operations:

1. **Non-Blocking I/O**, where calls return immediately with an error if they cannot complete.
   - Non-blocking I/O configures a socket so that every send/recv call returns immediately-either with data or with an error indicating "would block"—instead of stalling the calling thread.
   - **Enabling non-blocking mode:**
     1. ioctlsocket()

```
int ioctlsocket(
    SOCKET s,
    long   cmd,
    u_long *argp
);
// Example: turn on non-blocking
u_long mode = 1;            // 1 = non-blocking, 0 = blocking
if (ioctlsocket(s, FIONBIO, &mode) == SOCKET_ERROR) {
    // handle error: WSAGetLastError()
}
```

Non-blocking operation

2. **WSAAsyncSelect()** (event-driven alternative)

```
int WSAAsyncSelect(SOCKET s,HWND hWnd,u_int wMsg,long lEvent);
```

// Example: request FD_READ and FD_WRITE messages

```
if(WSAAsyncSelect(s,hWnd,WM_SOCKET,FD_READ|FD_WRITE)==SOCKET_ERROR) {
    // handle error
}
```

- Associates socket events with WM_SOCKET messages to window procedure.

Behavioral Characteristics

i. **Immediate Returns**
   - send() / recv() will never block.
   - If no data (or unable to send), they return SOCKET_ERROR and WSAGetLastError() yields WSAEWOULDBLOCK

Non-blocking operation

ii. Error Handling

- Must test for WSAEWOULDBLOCK and treat it as "no data right now" rather than a fatal error.

- **Application Responsibilities**
  - **Polling loop**: periodically retry I/O calls in a tight or timed loop.
  - **Timer-driven**: use SetTimer/timeSetEvent to schedule retries at intervals.
  - **Event loop**: integrate with select(), WSAPoll(), or WSAAsyncSelect() to be notified when the socket is ready.

Example Usages

Non-blocking operation

```
// Assume 's' is a connected SOCKET
// 1. Enable non-blocking
u_long mode = 1;
ioctlsocket(s, FIONBIO, &mode);
// 2. Attempt to receive
char buf[512];
int bytes = recv(s, buf, sizeof(buf), 0);
if (bytes > 0) {
    // Data received
} else if (bytes == 0) {
    // Connection closed gracefully
} else {
    int err = WSAGetLastError();
    if (err == WSAEWOULDBLOCK) {
        // No data right now — retry later
    } else {
        // Real socket error — handle
appropriately
    }
}
// 3. Loop or wait for readiness before retrying
//    e.g., select(), WSAPoll(), or
WSAAsyncSelect() events
```

Summary:

- **Non-blocking I/O** forces application to manage readiness notification and retry logic.
- It is **simpler** to set up than overlapped I/O but less efficient at scale due to polling overhead.
- Use **ioctlsocket(FIONBIO)** for raw non-blocking behavior, or **WSAAsyncSelect()** to integrate with a Windows message loop.

Asynchronous IO

2. **Asynchronous (Overlapped) I/O**, where operations are handed off to the OS and completed in the background, with notification delivered via events, callbacks, or I/O completion ports.

Prerequisites

1. **Overlapped-capable socket**

```
SOCKET s = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP,NULL,0,WSA_FLAG_OVERLAPPED);
```

- **OVERLAPPED structure**
  - Allocate and zero-initialize a WSAOVERLAPPED object to track each pending operation.
- **I/O functions**
  - Use WSASend() and WSARecv() (or their "From" variants) rather than send()/recv().
- **Completion mechanism** (in one way from below)
  - **Event object** assigned to OVERLAPPED.hEvent
  - **Completion routine** (callback)
  - **I/O Completion Port (IOCP)**

Functions: *WSASocket(), WSASend(),WSARecv()* – Already discussed

Example Usages


```
//Assume 's' is an overlapped socket and WSAStartup() has been called.
char buffer[1024];
WSABUF wsabuf = {.len =sizeof(buffer),.buf = buffer };
DWORD flags = 0, bytesReceived = 0;
WSAOVERLAPPED overlapped = { 0 };
overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
int result=WSARecv(s,&wsabuf,1,&bytesReceived,&flags,&overlapped,NULL);

if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
    // Handle error
} else {
    // Wait for completion
    WaitForSingleObject(overlapped.hEvent, INFINITE);
    // bytesReceived now contains the number of bytes read
}
CloseHandle(overlapped.hEvent);
```

Error Handling Functions- Asynchronous Operation

1. WSAGetLastError():

- If asynchronous function returns SOCKET_ERROR, and the specific error code retrieved by calling *WSAGetLastError()* is WSA_ISO_PENDING, then it means the operlapped operation has been successfully initiated and the completion will be indicated at a later time.
- Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.
- When the overlapped operation completes the amount of data transferred is indicated either through the `lpNumberOfBytesTransferred` parameter in **WSAGetOverlappedResult()**.

2. WSAGetOverlappedResult()

- WSAGetOverlappedResult() retrieves the results of an overlapped (asynchronous) operation on a socket. It allows an application to obtain the number of bytes transferred and any error status after an overlapped send or receive has completed.

Error Handling Functions- Asynchronous Operation

- **WSAGetOverlappedResult()**

```
BOOL WSAGetOverlappedResult(
    SOCKET                  s,
    LPWSAOVERLAPPED         lpOverlapped,
    LPDWORD                 lpNumberOfBytesTransferred,
    BOOL                    bWait,
    LPDWORD                 lpFlags
);
```

- **S**: The overlapped-capable socket on which the operation was issued.
- **lpOverlapped**: Pointer to the *WSAOVERLAPPED* structure used in the original WSASend or WSARecv call.
- **lpNumberOfBytesTransferred**: Receives the count of bytes sent or received.
- **bWait**: If TRUE, blocks until the overlapped operation has completed; if FALSE, returns immediately even if the operation is still pending.
- **lpFlags**: On receive operations, may be used to retrieve any flags that were set during the transfer (e.g., MSG_PARTIAL). On send operations, this parameter is ignored.

Error Handling Functions- Asynchronous Operation

- **Return Value**
- Returns nonzero (TRUE) if the overlapped operation completed successfully (either before or after the call, depending on bWait).
- Returns zero (FALSE) on failure; call WSAGetLastError() to retrieve the error code.
  - If the operation is still pending and bWait is FALSE, the error code will be WSA_IO_INCOMPLETE.

Error Handling Functions- Asynchronous Operation

Note:

1. **Waiting vs. Polling**
   1. Use bWait = TRUE when we want to block until completion.
   2. Use bWait = FALSE when we want to check status without blocking.
2. **Error Handling**
   1. Always call WSAGetLastError() if WSAGetOverlappedResult returns FALSE.
   2. Handle WSA_IO_INCOMPLETE specially if using polling.
3. **Integration with Event Objects**
   1. If used an event in our OVERLAPPED.hEvent, we can WaitForSingleObject on that event, then call WSAGetOverlappedResult with bWait = FALSE to retrieve the final byte count and check for errors.

Error Handling Functions- Asynchronous Operation

```
//   s              is an overlapped SOCKET
//   overlapped     is a WSAOVERLAPPED struct used in WSARecv()
//   completionDone is signaled (via event or IOCP) when recv completes
DWORD bytesTransferred = 0;
DWORD flags = 0;
// Option A: Block until complete (if you haven't already waited on the event)
if (!WSAGetOverlappedResult(s, &overlapped, &bytesTransferred, TRUE, &flags)) {
    int err = WSAGetLastError();
    // Handle error...
} else {
    // bytesTransferred now contains the number of bytes received
}
// Option B: Poll for completion
if (!WSAGetOverlappedResult(s, &overlapped, &bytesTransferred, FALSE, &flags)) {
    int err = WSAGetLastError();
    if (err == WSA_IO_INCOMPLETE) {
        // Operation still pending—try again later
    } else {
        // Real failure
    }
} else {
    // Completed; process bytesTransferred
}
```

select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()

- The select() function in WinSock allows a program to monitor multiple sockets simultaneously to determine if they are ready for reading, writing, or if an exception occurred. It is commonly used with blocking sockets to prevent blocking on operations such as accept(), recv(), or send().

Using select() with accept()-To avoid blocking on accept(), use select() to check if the listening socket is ready for reading. If it is, that means there is at least one incoming connection waiting to be accepted.

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(listenSock, &readfds);
timeval timeout = {5, 0}; // wait up to 5 seconds
int ready = select(listenSock+1, &readfds, NULL, NULL, &timeout);
if (ready > 0 && FD_ISSET(listenSock, &readfds)) {
    SOCKET clientSock = accept(listenSock, NULL, NULL);
    // Now handle the new client
}
```

select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()

Using select() with recv()/recvfrom() - Use select() to check if a socket has incoming data. This prevents recv() or recvfrom() from blocking if no data is available.

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(s, &readfds);
timeval timeout = {5, 0}; // wait up to 5 seconds
int ready = select(s+1, &readfds, NULL, NULL, &timeout);
if (ready > 0 && FD_ISSET(s, &readfds)) {
    char buf[512];
    int bytes = recv(s, buf, sizeof(buf), 0);
    // or: recvfrom() if using UDP
}
```

select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()

Using select() with send()/sendto() - Use select() to determine if a socket is ready for writing (i.e., send buffer has space). This helps avoid blocking if the send buffer is full.

```
fd_set writefds;
FD_ZERO(&writefds);
FD_SET(s, &writefds);
timeval timeout = {3, 0}; // wait up to 3 seconds
int ready = select(s+1, NULL, &writefds, NULL, &timeout);
if (ready > 0 && FD_ISSET(s, &writefds)) {
    const char *msg = "Hello";
    send(s, msg, strlen(msg), 0);
    // or: sendto() for datagram
}
```

select() Return Values and Timeout

- >0 → Number of sockets ready
- 0 → Timeout occurred, no socket is ready
- <0 → Error occurred (WSAGetLastError() for details)

Overlapped IO Example: Overlapped I/O Program (TCP Client with Overlapped WSARecv)

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")
int main() {
    WSADATA wsaData;
    SOCKET sock;
    struct sockaddr_in serverAddr;
    char recvBuf[1024];
    WSABUF wsaBuf;
    DWORD bytesReceived = 0, flags = 0;
    WSAOVERLAPPED overlapped;
    HANDLE hEvent;
```

Overlapped IO Example: Overlapped I/O Program (TCP Client with Overlapped WSARecv)

```
// 1. Initialize WinSock
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
     printf("WSAStartup failed\n");
     return 1;
 }

 // 2. Create Overlapped-capable socket
 sock = WSASocket(AF_INET, SOCK_STREAM,IPPROTO_TCP,NULL,0, WSA_FLAG_OVERLAPPED);
 if (sock == INVALID_SOCKET) {
     printf("WSASocket failed: %d\n", WSAGetLastError());
     WSACleanup();
     return 1;
 }
// 3. Setup server address
 serverAddr.sin_family = AF_INET;
 serverAddr.sin_port = htons(8080); // Connect to localhost:8080
 serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

Overlapped IO Example: Overlapped I/O Program (TCP Client with Overlapped WSARecv)

```
    // 4. Connect to server
    if(connect(sock,(struct sockaddr*)&serverAddr, sizeof(serverAddr))==SOCKET_ERROR)
    {
        printf("Connect failed: %d\n", WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return 1;
    }
 // 5. Initialize buffer and OVERLAPPED struct
    wsaBuf.buf = recvBuf;
    wsaBuf.len = sizeof(recvBuf);
    memset(&overlapped, 0, sizeof(overlapped));
    hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    overlapped.hEvent = hEvent;
// 6. Post an overlapped receive
    int result = WSARecv(sock, &wsaBuf, 1, &bytesReceived, &flags, &overlapped,NULL);
    if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
        printf("WSARecv failed: %d\n", WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return 1;
    }
```

Overlapped IO Example: Overlapped I/O Program (TCP Client with Overlapped WSARecv)

```
printf("Waiting for data...\n");
// 7. Wait for completion
    WaitForSingleObject(hEvent, INFINITE);

    if (!WSAGetOverlappedResult(sock, &overlapped, &bytesReceived, FALSE, &flags)) {
        printf("WSAGetOverlappedResult failed: %d\n", WSAGetLastError());
    } else {
        recvBuf[bytesReceived] = '\0'; // Null-terminate received string
        printf("Received %lu bytes: %s\n", bytesReceived, recvBuf);
    }

    // 8. Clean up
    CloseHandle(hEvent);
    closesocket(sock);
    WSACleanup();
    return 0;
}
```

Overlapped IO Example: Overlapped I/O Program (TCP Server)

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#pragma comment(lib, "ws2_32.lib")
int main() {
    WSADATA wsaData;
    SOCKET listenSock, clientSock;
    struct sockaddr_in serverAddr, clientAddr;
    int addrLen = sizeof(clientAddr);
    char recvBuf[1024];
    WSABUF wsaBuf;
    DWORD bytesReceived = 0, flags = 0;
    WSAOVERLAPPED overlapped;
    HANDLE hEvent;

    // 1. Initialize WinSock
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        printf("WSAStartup failed\n");
        return 1;
    }
```

Overlapped IO Example: Overlapped I/O Program (TCP Server)

```
// 2. Create overlapped-capable listening socket
    listenSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL,0, WSA_FLAG_OVERLAPPED);
    if (listenSock == INVALID_SOCKET) {
        printf("WSASocket failed: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    // 3. Bind and listen
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenSock, (struct sockaddr*)&serverAddr,sizeof(serverAddr))== SOCKET_ERROR) {
        printf("bind() failed: %d\n", WSAGetLastError());
        closesocket(listenSock);
        WSACleanup();
        return 1;
    }
    if (listen(listenSock, SOMAXCONN) == SOCKET_ERROR) {
        printf("listen() failed: %d\n", WSAGetLastError());
        closesocket(listenSock);
        WSACleanup();
        return 1;
    }
    printf("Server is listening on port 8080...\n");
```

Overlapped IO Example: Overlapped I/O Program (TCP Server)

```
// 4. Accept a client (blocking accept for demo)
clientSock = accept(listenSock, (struct sockaddr*)&clientAddr, &addrLen);
if (clientSock == INVALID_SOCKET) {
    printf("accept() failed: %d\n", WSAGetLastError());
    closesocket(listenSock);
    WSACleanup();
    return 1;
}
printf("Client connected!\n");
// 5. Prepare buffer and overlapped struct
wsaBuf.buf = recvBuf;
wsaBuf.len = sizeof(recvBuf);
memset(&overlapped, 0, sizeof(overlapped));
hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
overlapped.hEvent = hEvent;
// 6. Perform overlapped receive
int result = WSARecv(clientSock, &wsaBuf, 1,&bytesReceived,&flags,&overlapped,NULL);
if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
    printf("WSARecv failed: %d\n", WSAGetLastError());
    closesocket(clientSock);
    closesocket(listenSock);
    WSACleanup();
    return 1;
}
printf("Waiting for client data (Overlapped)...\n");
```

Overlapped IO Example: Overlapped I/O Program (TCP Server)


```
    // 7. Wait for I/O completion
    WaitForSingleObject(hEvent, INFINITE);

    if (!WSAGetOverlappedResult(clientSock, &overlapped, &bytesReceived, FALSE, &flags)) {
        printf("WSAGetOverlappedResult failed: %d\n", WSAGetLastError());
    } else {
        recvBuf[bytesReceived] = '\0';
        printf("Received %lu bytes: %s\n", bytesReceived, recvBuf);
    }

    // 8. Cleanup
    CloseHandle(hEvent);
    closesocket(clientSock);
    closesocket(listenSock);
    WSACleanup();

    return 0;
}
```

Event-Driven Programming in Winsock

- Event-driven programming in WinSock enables applications to efficiently respond to network events such as readiness to read or write, connection requests, or disconnections—without blocking or polling continuously.
- This is ideal for GUI apps, single-threaded servers, or any application that needs to remain responsive.

1. Using WSAEventSelect()

- *WSAEventSelect()* configures a socket to notify the application via an **event object** when specified network events occur (e.g., FD_READ, FD_WRITE, FD_ACCEPT).

int WSAEventSelect(SOCKET s,WSAEVENT hEventObject,long lNetworkEvents);

- s – the socket to monitor.
- hEventObject – a Win32 event handle created via WSACreateEvent().
- lNetworkEvents – a bitmask of events (e.g., FD_READ | FD_WRITE | FD_CLOSE).

Event-Driven Programming in Winsock

Example of WSAEventSelect()

```
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
WSAEVENT hEvent = WSACreateEvent();
// Monitor read and close events
WSAEventSelect(s, hEvent, FD_READ | FD_CLOSE);
```

2. Using Completion Routines (APC model)

- Completion routines (also called **asynchronous procedure calls**, or APCs) are a powerful event-driven mechanism used in **overlapped I/O**. Instead of waiting on an event or polling, the system **automatically invokes callback function** when the operation completes.

Requirements:

- Use WSASend(), WSARecv(), or WSAIoctl() with:
  - OVERLAPPED structure
  - Pointer to the **completion routine**
- The calling thread must be in an **alertable wait state** using:
  - SleepEx()
  - WaitForSingleObjectEx()

Event-Driven Programming in Winsock

```
void CALLBACK CompletionCallback(DWORD dwError,DWORD cbTransferred,
LPWSAOVERLAPPED lpOverlapped,DWORD dwFlags);
```

Example:

```
void CALLBACK OnRecvComplete(DWORD dwError,DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,DWORD dwFlags) {
    printf("Asynchronous receive complete: %lu bytes\n", cbTransferred);
}
WSARecv(s, &wsaBuf, 1, NULL, &flags, &overlapped, OnRecvComplete);
// Enter alertable wait state
SleepEx(INFINITE, TRUE); // The callback runs when the I/O completes
```

- **WSAEventSelect()** is easy to integrate with an event loop or GUI app.
- **Completion routines** are more efficient for asynchronous I/O and avoid explicit waiting.
- Both approaches are superior to manual polling and allow responsive, non-blocking I/O in event-driven applications.

Winsock Extensions

- WinSock provides several advanced mechanisms beyond basic select() or blocking socket operations. These extensions are designed for higher performance, scalability, and finer control of asynchronous network events.

1. Advanced Polling Mechanism: WSAPoll()

- WSAPoll() is a scalable, thread-safe alternative to select() that checks multiple sockets for readiness without the limitations of fd_set size or destruction of input sets.

```
int WSAPoll(LPWSAPOLLFD fdArray,ULONG fds,INT timeout);
```

- fdArray: Pointer to an array of WSAPOLLFD structures.
- fds: Number of entries in the array.
- timeout: Timeout in milliseconds (-1 for infinite, 0 for non-blocking).

Winsock Extensions

WSAPOLLFD Structure

```
typedef struct _WSAPOLLFD {
    SOCKET fd;
    SHORT events;     // Requested events (e.g., POLLRDNORM, POLLWRNORM)
    SHORT revents;    // Returned events
} WSAPOLLFD;
```

Events

Constant	Meaning
POLLRDNORM	Ready for normal read
POLLWRNORM	Ready for normal write
POLLERR	Error occurred
POLLHUP	Connection closed by peer

Winsock Extensions

Example:

```
WSAPOLLFD fds[1];
fds[0].fd = s;
fds[0].events = POLLRDNORM;
int ret = WSAPoll(fds, 1, 5000); // 5-second timeout
if (ret > 0 && (fds[0].revents & POLLRDNORM)) {
    char buf[512];
    int len = recv(s, buf, sizeof(buf), 0);
    // process data
}
```

Advantages over select()

- Handles thousands of sockets (unlike select's 64 or 1024 limit).
- Keeps input structures unchanged.
- Easier integration in multi-threaded and scalable systems.

2. WSAEventSelect() : Already Discussed

Basic cross-platform networking application

- Creating a **basic cross-platform networking application** requires using libraries or APIs that abstract the underlying differences between Windows and Unix-like systems (Linux, macOS).
- The POSIX socket API is nearly identical across Unix platforms, and with some care, it can be made to work on Windows using conditional compilation.
- **Conditional compilation** is a feature in C/C++ that allows you to **include or exclude parts of code** during compilation based on certain conditions. This is especially useful for writing **cross-platform code**, where different code is needed for different operating systems or compilers.
- Conditional compilation uses **preprocessor directives**, which are instructions to the compiler **before actual compilation** begins.

Directive	Purpose
#ifdef	If macro is defined
#ifndef	If macro is **not** defined
#if, #elif, #else, #endif	If condition is true / false
#define	Define a macro
#undef	Undefine a macro

Basic cross-platform networking application [Server]

```
// cross_server.c
#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#define INVALID_SOCKET -1
#define SOCKET_ERROR   -1
typedef int SOCKET;
#endif

#include <stdio.h>
```

```
int main() {
#ifdef _WIN32
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2),&wsaData);
#endif
    SOCKET serverSock=socket(AF_INET,SOCK_STREAM,
0);
    if (serverSock == INVALID_SOCKET) {
        perror("socket failed");
        return 1;
    }
    struct sockaddr_in serverAddr = {0};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(8080);
    if (bind(serverSock, (struct
sockaddr*)&serverAddr,sizeof(serverAddr)) ==
SOCKET_ERROR) {
        perror("bind failed");
        return 1;
    }
```

Basic cross-platform networking application [Server]

```
    listen(serverSock, 5);
    printf("Server listening on port 8080...\n");
    struct sockaddr_in clientAddr;
    socklen_t clientLen = sizeof(clientAddr);
    SOCKET clientSock = accept(serverSock,(struct sockaddr*)&clientAddr,&clientLen);
    char buffer[1024];
    int bytesRead;
    while ((bytesRead = recv(clientSock, buffer, sizeof(buffer) - 1, 0)) > 0) {
        buffer[bytesRead] = '\0';
        printf("Client: %s\n", buffer);
        send(clientSock, buffer, bytesRead, 0); // echo back
    }
#ifdef _WIN32
    closesocket(clientSock);
    closesocket(serverSock);
    WSACleanup();
#else
    close(clientSock);
    close(serverSock);
#endif
    return 0;
}
```

Basic cross-platform networking application [Client]

```
// cross_client.c
#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#define INVALID_SOCKET -1
#define SOCKET_ERROR   -1
typedef int SOCKET;
#endif
#include <stdio.h>
int main() {
#ifdef _WIN32
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2),
&wsaData);
#endif
```

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == INVALID_SOCKET) {
        perror("socket failed");
        return 1;
    }
    struct sockaddr_in serverAddr = {0};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1",
&serverAddr.sin_addr);
    if (connect(sock, (struct
sockaddr*)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR) {
        perror("connect failed");
        return 1;
    }
    char msg[512];
    char buf[1024];
```

Basic cross-platform networking application [Client]

```
while (1) {
        printf("Enter message: ");
        fgets(msg, sizeof(msg), stdin);
        send(sock, msg, strlen(msg), 0);

        int bytes = recv(sock, buf, sizeof(buf) - 1, 0);
        if (bytes <= 0) break;
        buf[bytes] = '\0';
        printf("Echo: %s\n", buf);
    }

#ifdef _WIN32
    closesocket(sock);
    WSACleanup();
#else
    close(sock);
#endif

    return 0;
}
```

End of Chapter 5