

기본 학습: 소켓 프로그래밍 기본

Chapter 4 | TCP 서버/클라이언트 ●

Chapter 5 | 멀티스레드 ●

Chapter 6 | UDP 서버/클라이언트 ●

Chapter 7 | 소켓 옵션 ●

Chapter 8 | GUI 소켓 애플리케이션 ●

TCP 서버/클라이언트

* 학습목표

- TCP 서버/클라이언트의 기본 구조와 동작 원리를 이해한다.
- TCP 애플리케이션 작성에 필요한 소켓 함수를 익힌다.
- 애플리케이션 프로토콜의 필요성을 이해하고, 메시지 설계 기법을 익힌다.

01. TCP 서버/클라이언트 구조

02. TCP 서버/클라이언트 분석

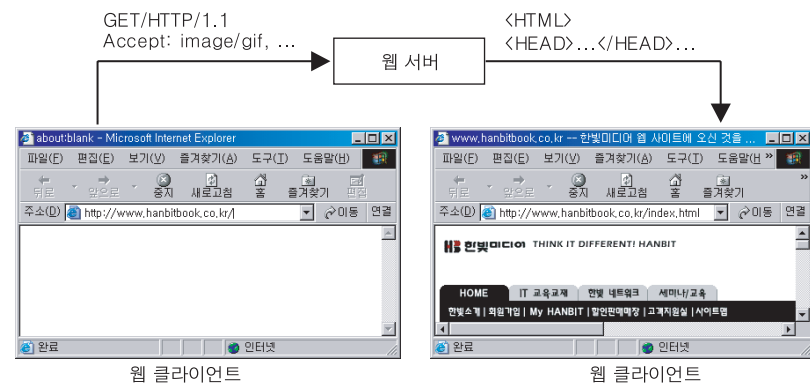
03. 애플리케이션 프로토콜과 메시지 설계

TCP 서버/클라이언트 구조

지금까지 1~3장에 걸쳐 소켓 애플리케이션을 위한 기초를 다졌다. 1장에서는 네트워크 기본 이론과 더불어 네트워크 애플리케이션의 구조로 자주 사용되는 서버/클라이언트 개념을 소개하였고, 2~3장에서는 소켓 애플리케이션을 작성하기 위한 기본 함수를 공부하였다.

이 장에서는 지금까지 배운 내용을 토대로 간단한 TCP 서버/클라이언트를 작성하고, 관련 소켓 함수를 공부할 것이다. 우선 실생활에서 자주 사용하는 TCP 서버/클라이언트의 예를 살펴보자.

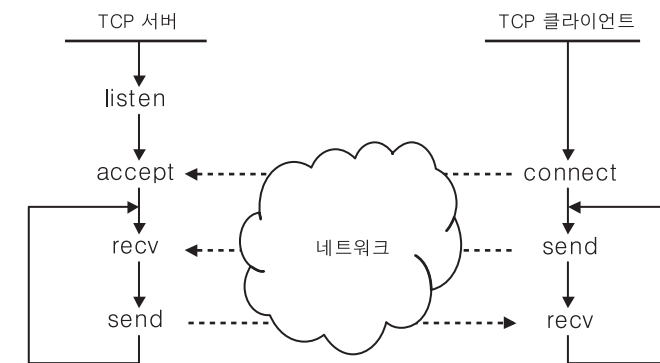
[그림 4-1]은 웹 서버와 웹 클라이언트가 동작하는 모습을 보여준다. PC에서 사용하는 대표적인 웹 클라이언트인 인터넷 익스플로러(Internet Explorer)는 사용자가 입력한 주소를 참조하여 접속 대기 중인 웹 서버에 접속한 후, HTTP를 이용하여 요청 메시지(예를 들면, 웹 페이지 이름)를 보낸다. 웹 서버는 인터넷 익스플로러가 보낸 데이터를 분석한 후, 역시 HTTP를 이용하여 응답 메시지(예를 들면, 웹 페이지 데이터)를 다시 보낸다. 인터넷 익스플로러는 웹 서버가 보낸 데이터를 받아 화면에 표시한다. HTTP는 TCP에 기반한 프로토콜이므로 웹 서버/클라이언트는 대표적인 TCP 서버/클라이언트 애플리케이션이라 할 수 있다.



▶ [그림 4-1] 웹 서버/클라이언트

TCP 서버/클라이언트의 동작 방식은 다음과 같다([그림 4-2] 참조). 괄호 안에는 이 장에서 배울 소켓 함수를 표시하였다.

- ① 서버는 먼저 실행하여 클라이언트가 접속하기를 기다린다(**listen**).
- ② 클라이언트가 서버에 접속(**connect**)하여 데이터를 보낸다(**send**).
- ③ 서버는 클라이언트 접속을 수용하고(**accept**), 클라이언트가 보낸 데이터를 받아서(**recv**) 처리한다.
- ④ 서버는 처리한 데이터를 클라이언트에 보낸다(**send**).
- ⑤ 클라이언트는 서버가 보낸 데이터를 받아서(**recv**) 자신의 목적에 맞게 사용한다.



▶ [그림 4-2] TCP 서버/클라이언트 동작 방식

이와 같은 방식은 텔넷 서버/클라이언트, FTP 서버/클라이언트 등에도 동일하게 적용된다. 이제 TCP 서버/클라이언트의 동작 원리를 소켓 프로그래밍 관점에서 좀더 구체적으로 살펴보도록 하자.

서버/클라이언트의 개념을 하드웨어 관점에서만 생각하려는 경향이 있다. 즉, 어떤 시스템은 서버고, 어떤 시스템은 클라이언트라는 것이다. 이는 하드웨어 성능을 염두에 둔 표현으로, 자칫 오해할 소지가 있다. 시스템이 서버 또는 클라이언트라는 것은 상황에 따라 하드웨어 또는 소프트웨어 관점에서 생각하는 것이 좋다. 예를 들어, “웹 서버를 작성한다”고 할 때는 ‘웹 서버 소프트웨어’를 의미하고, “웹 서버를 교체한다”고 할 때는 ‘웹 서버 소프트웨어를 실행하는 하드웨어’를 의미하는 것으로 생각하면 된다.

서버/클라이언트의 개념을 소프트웨어 관점에서 본다면 하나의 하드웨어가 서버 또는 클라이언트가 될 수 있다. 일반적인 기준으로 볼 때 성능이 떨어지는 시스템이라도 서버 소프트웨어를 실행하여 특정 서비스를 외부에 제공한다면 서버고, 역으로 고성능 시스템이라도 클라이언트 소프트웨어를 실행하여 다른 시스템의 서비스를 이용한다면 클라이언트인 것이다. 이렇게 본다면 한 시스템이 서버와 클라이언트 역할을 동시에 할 수도 있다.

1 동작 원리

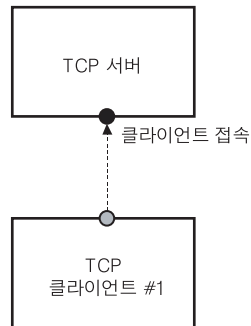
[그림 4-3]~[그림 4-6]은 TCP 서버에 TCP 클라이언트가 접속하여 통신을 수행하는 과정을 보여준다. 각 단계별 동작을 요약하면 다음과 같다.

- [그림 4-3] 서버는 소켓을 생성한 후 클라이언트가 접속하기를 기다린다. 이때 서버가 사용하는 소켓은 특정 포트 번호(예를 들면, 9000)와 결합되어(bind) 있어서 이 포트 번호로 접속하는 클라이언트만 수용할 수 있다.
- [그림 4-4] 클라이언트가 접속한다. 이때 TCP 프로토콜 수준에서 연결 설정을 위한 패킷 교환이 이루어진다.
- [그림 4-5] TCP 프로토콜 수준의 연결 절차가 끝나면, 서버는 접속한 클라이언트와 통신할 수 있는 **새로운 소켓을 생성**한다. 서버가 클라이언트와 데이터를 주고받을 때는 이 소켓을 사용한다. **기존의 소켓은 새로운 클라이언트 접속을 수용하는 용도로 계속 사용**한다.
- [그림 4-6] 두 클라이언트가 접속한 후의 상태를 나타낸 것이다. 서버측에는 총 세 개의 소켓이 존재하며, 이 중 두 소켓이 실제 클라이언트와 통신하는 용도로 사용된다.

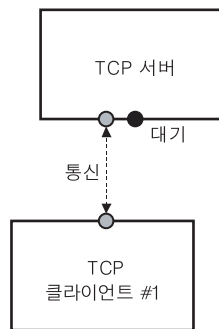
TCP 연결 시 SYN, SYN/ACK, ACK 세 개의 패킷 교환이 일어난다.



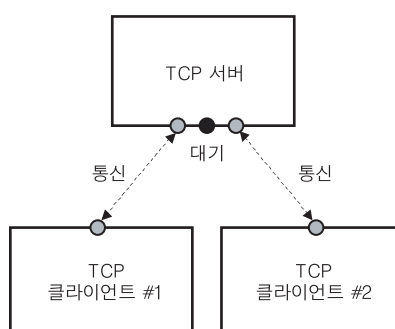
▶ [그림 4-3] 동작 원리(1)



▶ [그림 4-4] 동작 원리(2)

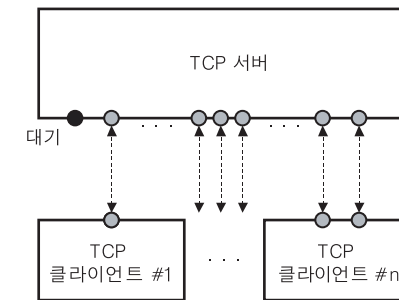


▶ [그림 4-5] 동작 원리(3)



▶ [그림 4-6] 동작 원리(4)

TCP 서버/클라이언트가 통신하는 상황을 일반적인 형태로 나타내면 [그림 4-7]과 같다. 서버측 소켓과 클라이언트측 소켓이 1 대 1로 대응하는 것을 알 수 있다. ‘TCP 클라이언트 #n’의 경우처럼 한 클라이언트가 두 개 이상의 소켓을 사용하여 서버에 접속하는 것도 가능하다.



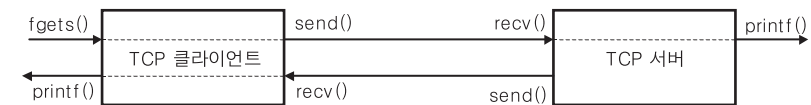
▶ [그림 4-7] TCP 서버/클라이언트 통신

2 [실습] 간단한 TCP 서버/클라이언트

간단한 TCP 서버/클라이언트를 작성하고 실행하도록 하자. 서버와 클라이언트의 동작은 다음과 같다([그림 4-8] 참조).

- 서버: 클라이언트가 보낸 데이터를 받아(*recv*), 이를 문자열로 간주하여 무조건 화면에 출력한다(*printf*). 그리고 받은 데이터를 변경 없이 다시 클라이언트에 보낸다(*send*).
- 클라이언트: 사용자가 키보드로 입력한 문자열을(*fgets*) 서버에 보낸다(*send*). 서버가 받은 데이터를 그대로 되돌려 보내면, 클라이언트는 이를 받아(*recv*) 화면에 출력한다(*printf*).

받은 데이터를 그대로 다시 보낸다는 뜻으로 에코(echo) 서버라고 부른다.



▶ [그림 4-8] TCP 서버/클라이언트 예제

여기서는 예제 코드를 실행함으로써 동작 방식을 대략적으로 파악하는 데 중점을 두며, 코드 분석은 2절에서 할 것이다.

따라하기

1 Tcpserver와 Tcpsclient라는 콘솔 애플리케이션 프로젝트 두 개를 생성한 후 각각 다음과 같이 입력한다. 프로젝트 생성 옵션과 컴파일, 링크 방법은 1장 예제와 동일하다.

[예제] Tcpsserver.cpp

```
001 #include <winsock2.h>
002 #include <stdlib.h>
003 #include <stdio.h>
004
005 #define BUFSIZE 512
006
007 // 소켓 함수 오류 출력 후 종료
008 void err_quit(char *msg)
009 {
010     LPVOID lpMsgBuf;
011     FormatMessage(
012         FORMAT_MESSAGE_ALLOCATE_BUFFER|
013         FORMAT_MESSAGE_FROM_SYSTEM,
014         NULL, WSAGetLastError(),
015         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
016         (LPTSTR)&lpMsgBuf, 0, NULL);
017     MessageBox(NULL, (LPCTSTR)lpMsgBuf, msg, MB_ICONERROR);
018     LocalFree(lpMsgBuf);
019     exit(-1);
020 }
021
022 // 소켓 함수 오류 출력
023 void err_display(char *msg)
024 {
025     LPVOID lpMsgBuf;
026     FormatMessage(
027         FORMAT_MESSAGE_ALLOCATE_BUFFER|
028         FORMAT_MESSAGE_FROM_SYSTEM,
029         NULL, WSAGetLastError(),
030         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
```

```
031         (LPTSTR)&lpMsgBuf, 0, NULL);
032     printf("[%s] %s", msg, (LPCTSTR)lpMsgBuf);
033     LocalFree(lpMsgBuf);
034 }
035
036 int main(int argc, char* argv[])
037 {
038     int retval;
039
040     // 윈속 초기화
041     WSADATA wsa;
042     if(WSAStartup(MAKEWORD(2,2), &wsa) != 0)
043         return -1;
044
045     // socket()
046     SOCKET listen_sock = socket(AF_INET, SOCK_STREAM, 0);
047     if(listen_sock == INVALID_SOCKET) err_quit("socket()");
048
049     // bind()
050     SOCKADDR_IN serveraddr;
051     ZeroMemory(&serveraddr, sizeof(serveraddr));
052     serveraddr.sin_family = AF_INET;
053     serveraddr.sin_port = htons(9000);
054     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
055     retval = bind(listen_sock, (SOCKADDR *)&serveraddr,
056                   sizeof(serveraddr));
057     if(retval == SOCKET_ERROR) err_quit("bind()");
058
059     // listen()
060     retval = listen(listen_sock, SOMAXCONN);
061     if(retval == SOCKET_ERROR) err_quit("listen()");
062
063     // 데이터 통신에 사용할 변수
064     SOCKET client_sock;
065     SOCKADDR_IN clientaddr;
066     int addrlen;
```

```

066     char buf[BUFSIZE+1];
067
068     while(1){
069         // accept()
070         addrlen = sizeof(clientaddr);
071         client_sock = accept(listen_sock,
072                             (SOCKADDR *) &clientaddr, &addrlen);
072         if(client_sock == INVALID_SOCKET){
073             err_display("accept()");
074             continue;
075         }
076         printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
077              inet_ntoa(clientaddr.sin_addr),
078              ntohs(clientaddr.sin_port));
079
080         // 클라이언트와 데이터 통신
081         while(1){
082             // 데이터 받기
083             retval = recv(client_sock, buf, BUFSIZE, 0);
084             if(retval == SOCKET_ERROR){
085                 err_display("recv()");
086                 break;
087             }
088             else if(retval == 0)
089                 break;
090
091             // 받은 데이터 출력
092             buf[retval] = '\0';
093             printf("[TCP/%s:%d] %s\n",
094                  inet_ntoa(clientaddr.sin_addr),
095                  ntohs(clientaddr.sin_port), buf);
096
097             // 데이터 보내기
098             retval = send(client_sock, buf, retval, 0);
099             if(retval == SOCKET_ERROR){

```

```

098             err_display("send()");
099             break;
100         }
101     }
102
103     // closesocket()
104     closesocket(client_sock);
105     printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
106          inet_ntoa(clientaddr.sin_addr),
107          ntohs(clientaddr.sin_port));
108
109     // closesocket()
110     closesocket(listen_sock);
111
112     // 윈속 종료
113     WSACleanup();
114     return 0;
115 }

```

[예제] TCPCClient.cpp

```

001 #include <winsock2.h>
002 #include <stdlib.h>
003 #include <stdio.h>
004
005 #define BUFSIZE 512
006
007 // 소켓 함수 오류 출력 후 종료
008 void err_quit(char *msg)
009 {
010     LPVOID lpMsgBuf;
011     FormatMessage(
012         FORMAT_MESSAGE_ALLOCATE_BUFFER|
013         FORMAT_MESSAGE_FROM_SYSTEM,
014         NULL, WSAGetLastError(),

```

```

015     MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
016     (LPTSTR)&lpMsgBuf, 0, NULL);
017     MessageBox(NULL, (LPCTSTR)lpMsgBuf, msg, MB_ICONERROR);
018     LocalFree(lpMsgBuf);
019     exit(-1);
020 }
021
022 // 소켓 함수 오류 출력
023 void err_display(char *msg)
024 {
025     LPVOID lpMsgBuf;
026     FormatMessage(
027         FORMAT_MESSAGE_ALLOCATE_BUFFER|
028         FORMAT_MESSAGE_FROM_SYSTEM,
029         NULL, WSAGetLastError(),
030         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
031         (LPTSTR)&lpMsgBuf, 0, NULL);
032     printf("[%s] %s", msg, (LPCTSTR)lpMsgBuf);
033     LocalFree(lpMsgBuf);
034 }
035
036 // 사용자 정의 데이터 수신 함수
037 int recvn(SOCKET s, char *buf, int len, int flags)
038 {
039     int received;
040     char *ptr = buf;
041     int left = len;
042
043     while(left > 0){
044         received = recv(s, ptr, left, flags);
045         if(received == SOCKET_ERROR)
046             return SOCKET_ERROR;
047         else if(received == 0)
048             break;
049         left -= received;
050         ptr += received;
051     }

```

```

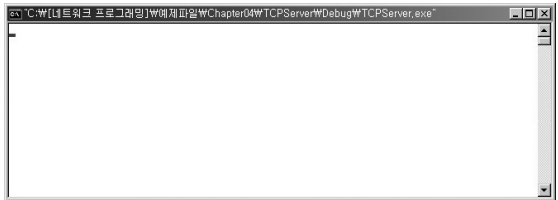
052
053     return (len - left);
054 }
055
056 int main(int argc, char* argv[])
057 {
058     int retval;
059
060     // 윈속 초기화
061     WSADATA wsa;
062     if(WSAStartup(MAKEWORD(2,2), &wsa) != 0)
063         return -1;
064
065     // socket()
066     SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
067     if(sock == INVALID_SOCKET) err_quit("socket()");
068
069     // connect()
070     SOCKADDR_IN serveraddr;
071     serveraddr.sin_family = AF_INET;
072     serveraddr.sin_port = htons(9000);
073     serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");
074     retval = connect(sock, (SOCKADDR *)&serveraddr,
075                     sizeof(serveraddr));
076     if(retval == SOCKET_ERROR) err_quit("connect()");
077
078     // 데이터 통신에 사용할 변수
079     char buf[BUFSIZE+1];
080     int len;
081
082     // 서버와 데이터 통신
083     while(1){
084         // 데이터 입력
085         ZeroMemory(buf, sizeof(buf));
086         printf("\n[보낼 데이터]");
087         if(fgets(buf, BUFSIZE+1, stdin) == NULL)
088             break;

```

```
089 // '\n' 문자 제거
090 len = strlen(buf);
091 if(buf[len-1] == '\n')
092     buf[len-1] = '\0';
093 if(strlen(buf) == 0)
094     break;
095
096 // 데이터 보내기
097 retval = send(sock, buf, strlen(buf), 0);
098 if(retval == SOCKET_ERROR){
099     err_display("send()");
100     break;
101 }
102 printf("[TCP 클라이언트] %d바이트를 보냈습니다.\n", retval);
103
104 // 데이터 받기
105 retval = recv(sock, buf, retval, 0);
106 if(retval == SOCKET_ERROR){
107     err_display("recv()");
108     break;
109 }
110 else if(retval == 0)
111     break;
112
113 // 받은 데이터 출력
114 buf[retval] = '\0';
115 printf("[TCP 클라이언트] %d바이트를 받았습니다.\n", retval);
116 printf("[받은 데이터] %s\n", buf);
117 }
118
119 // closesocket()
120 closesocket(sock);
121
122 // 윈속 종료
123 WSACleanup();
124 return 0;
125 }
```

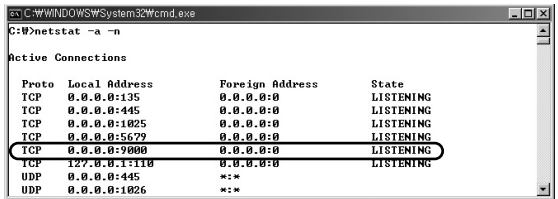
2 실습 환경에 따라 일부 코드를 변경하도록 한다. 예제 코드는 실습의 편의를 위해 서버와 클라이언트가 같은 컴퓨터에서 실행되는 경우를 가정하고 있으므로, 서로 다른 컴퓨터에서 실행하려면 TCPClient.cpp 파일의 73행을 서버의 IP 주소로 변경해야 한다. 서버는 포트 번호 9000을 사용하고 있으므로, 다른 포트 번호를 사용하려면 TCPServer.cpp 파일의 53행과 TCPClient.cpp 파일의 72행을 변경하면 된다. 컴파일, 링크 후 다음 순서를 따라 실행해보자.

① TCP 서버(TCPServer.EXE)를 실행한다.



▶ [그림 4-9] 실행 화면(1) – TCP 서버

② ‘명령 프롬프트’를 실행한 후 ‘netstat -a -n’ 명령을 실행한다. netstat 명령은 TCP/IP 네트워크 연결 상태 정보를 표시하는 유틸리티로, 윈도우 운영체제에서 기본으로 제공한다. ‘-a’ 옵션은 모든 연결 정보와 포트 정보를 표시하도록 하며, ‘-n’ 옵션은 IP 주소와 포트 번호를 숫자로 표시하도록 한다. [그림 4-10]은 필자의 컴퓨터에서 실행한 화면으로, TCP 포트 번호 9000번의 상태가 LISTENING(연결 대기 중)임을 보여준다.



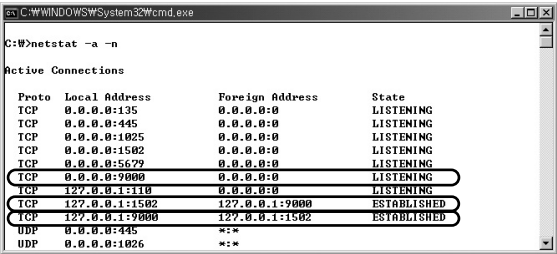
▶ [그림 4-10] 실행 화면(2) – netstat 명령

③ TCP 클라이언트(TCPClient.EXE)를 실행한다.



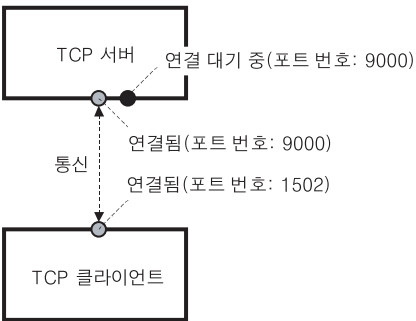
▶ [그림 4-11] 실행 화면(3) – TCP 클라이언트

④ 다시 'netstat -a -n' 명령을 실행한다. 예제 프로그램과 관련된 정보는 총 세 개며, 각각 포트 번호 9000, 1502, 9000번이 사용되고 있다. 첫번째 행은 LISTENING(연결 대기 중) 상태고, 나머지 두 행은 ESTABLISHED(연결됨) 상태임을 알 수 있다.



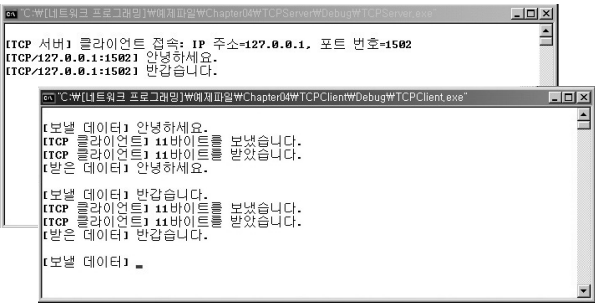
▶ [그림 4-12] 실행 화면(4) - netstat 명령

[그림 4-13]은 [그림 4-12]의 결과를 토대로 예제 프로그램 실행 중 TCP 서버/클라이언트의 상태를 나타낸 것이다. 서버와 클라이언트가 같은 컴퓨터에서 실행 중이므로 세 개의 상태 정보(두 개는 서버, 나머지 하나는 클라이언트)가 netstat 명령의 결과로 출력된다. 서버측에서 만든 소켓은 모두 같은 포트 번호(9000)를 사용한다는 점에 주목하도록 하자.



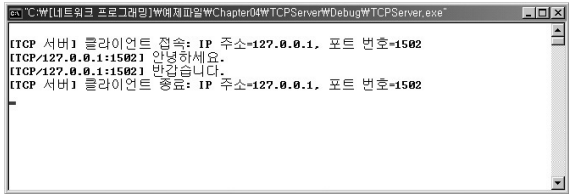
▶ [그림 4-13] TCP 서버/클라이언트 상태(1)

⑤ 클라이언트측에서 문자를 입력한 후 Enter 키를 누르면, 입력 문자열이 서버에 전송된다. 클라이언트 화면에는 전송된 바이트 수가 표시되고, 서버 화면에는 클라이언트로부터 받은 문자열이 그대로 표시된다.



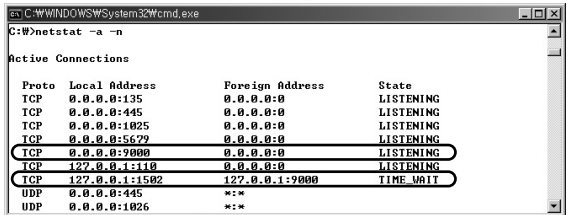
▶ [그림 4-14] 실행 화면(5) - TCP 서버/클라이언트

⑥ 글자를 입력하지 않은 상태에서 Enter 키를 누르면 클라이언트는 종료하고, 서버 화면에 다음과 같이 표시된다.



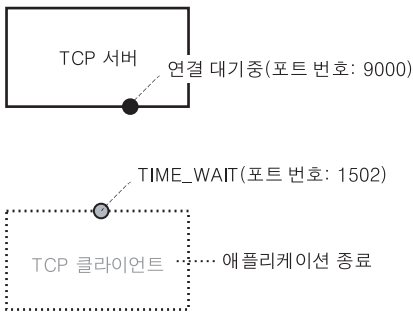
▶ [그림 4-15] 실행 화면(6) - 접속 종료

⑦ 'netstat -a -n' 명령을 실행하여 TCP 상태를 확인한다. LISTENING 상태인 포트(9000)와 TIME_WAIT 상태인 포트(1502)가 존재함을 알 수 있다.



▶ [그림 4-16] 실행 화면(7) - netstat 명령

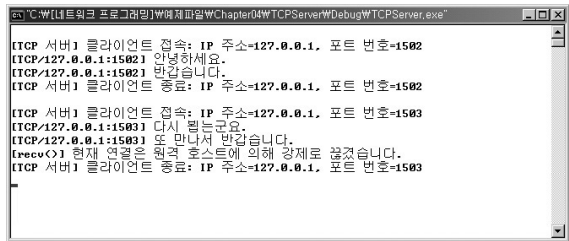
[그림 4-17]은 [그림 4-16]의 결과를 토대로 TCP 서버/클라이언트의 상태를 나타낸 것이다. 서버는 계속 실행 중이므로 포트 번호(9000)가 계속 유지된다. 한편, 클라이언트는 종료했음에도 불구하고 사용하던 포트 번호(1502)가 TIME_WAIT 상태로 남아 있다.



▶ [그림 4-17] TCP 서버/클라이언트 상태(2)

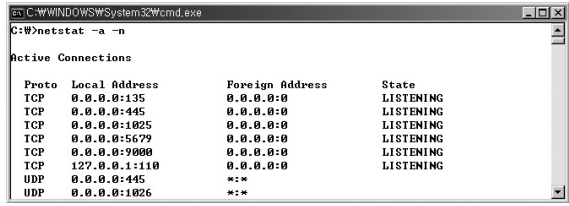
약 5분 후 다시 'netstat -a -n' 명령을 실행하면 TIME_WAIT 상태인 포트는 사라지고 [그림 4-10]과 동일한 결과를 볼 수 있을 것이다.

⑧ TCP 클라이언트를 다시 실행하여 TCP 서버에 데이터를 몇 개 보낸 후 'Ctrl+C'를 눌러 종료해보자. 클라이언트가 강제 종료되었음을 암시하는 메시지가 서버 화면에 표시된다.



▶ [그림 4-18] 실행 화면(8) - 접속 종료

⑨ 'netstat -a -n' 명령을 실행하여 TCP 상태를 확인해보자. 이번에는 LISTENING 상태인 포트(9000)만 존재하고, TIME_WAIT 상태인 포트는 존재하지 않음을 알 수 있다.



▶ [그림 4-19] 실행 화면(9) - netstat 명령



저자 한마디

[그림 4-9]~[그림 4-19]의 결과를 요약하면 다음과 같다.

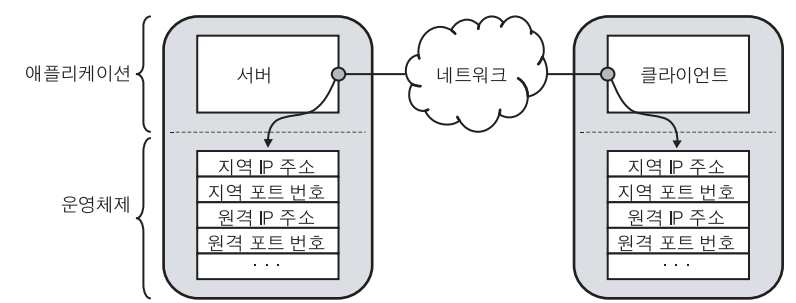
- 정상적인 방법으로 애플리케이션을 종료하면, 이 애플리케이션이 사용하던 TCP 포트는 일정 시간(5분 이내) TIME_WAIT 상태에 있다가 사라진다(그림 4-16).
- 비정상적인 방법으로 애플리케이션을 종료하면(강제 종료), 이 애플리케이션이 사용하던 포트는 TIME_WAIT 상태를 거치지 않고 사라진다(그림 4-19).

TCP 서버/클라이언트 분석

이 절에서는 TCP 서버/클라이언트 예제를 분석하고, 관련 소켓 함수를 공부할 것이다. 분석에 앞서 1장에서 소개한 소켓의 개념을 좀더 구체적으로 살펴보자.

애플리케이션 관점에서 소켓은 운영체제의 TCP/IP 구현에서 제공하는 데이터 구조체를 참조하기 위한 매개체가 된다. [그림 4-20]은 서버/클라이언트가 소켓을 이용하여 통신할 때 운영체제가 제공하는 데이터 구조체를 보여주고 있다. TCP/IP를 이용하여 애플리케이션이 통신을 수행하기 위해서는 다음과 같은 요소가 결정되어야 한다.

- 프로토콜
소켓을 생성할 때 결정한다(2장 참조).
- 지역(local) IP 주소와 지역 포트 번호
서버 또는 클라이언트 자신의 주소를 의미한다.
- 원격(remote) IP 주소와 원격 포트 번호
서버 또는 클라이언트가 통신하는 상대의 주소를 의미한다.



▶ [그림 4-20] 소켓 데이터 구조체(1)

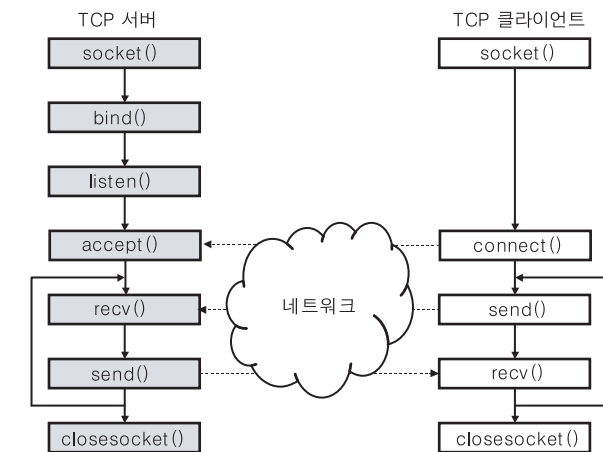
이 절에서 설명할 소켓 함수는 (데이터 전송 함수를 제외하면) 지역 주소와 원격 주소를 결정하고, TCP 상태(LISTENING, ESTABLISHED, ...)를 변경하기 위한 일련의 절차라

할 수 있다. 예제에서 사용한 함수를 각각 서버 함수, 클라이언트 함수 그리고 데이터 전송 함수로 구분하여 살펴보도록 하자.

1 서버 함수

일반적으로 TCP 서버는 다음과 같은 순서로 소켓 함수를 호출한다([그림 4-21] 참조).

- ① socket() 함수를 이용하여 소켓을 생성한다.
- ② bind() 함수를 이용하여 지역 IP 주소와 지역 포트 번호를 결정한다.
- ③ listen() 함수를 이용하여 TCP 상태를 LISTENING으로 변경한다.
- ④ accept() 함수를 이용하여 자신에 접속한 클라이언트와 통신할 수 있는 새로운 소켓을 생성한다. 이때 원격 IP 주소와 원격 포트 번호가 결정된다.
- ⑤ send(), recv() 등의 데이터 전송 함수를 이용하여 클라이언트와 통신을 수행한 후, closesocket() 함수를 이용하여 소켓을 닫는다.
- ⑥ 새로운 클라이언트 접속이 들어올 때마다 ④~⑤ 과정을 반복한다.



▶ [그림 4-21] TCP 서버/클라이언트

TCP 서버/클라이언트 공통 함수(socket(), closesocket(), send(), recv())를 제외한 서버 함수를 차례대로 살펴보도록 하자.

bind() 함수

bind() 함수는 서버의 지역 IP 주소와 지역 포트 번호를 결정하는 역할을 한다.

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

성공: 0, 실패: SOCKET_ERROR

- s
클라이언트 접속을 수용할 목적으로 만든 소켓으로, 지역 IP 주소와 지역 포트 번호가 아직 결정되지 않은 상태다.
- name
소켓 주소 구조체(TCP/IP의 경우 SOCKADDR_IN 타입) 변수를 지역 IP 주소와 지역 포트 번호로 초기화한 후, 이 변수의 주소값을 여기에 대입한다.
- namelen
소켓 주소 구조체 변수의 길이(바이트 단위)를 대입한다.

TCPServer 예제에서 bind() 함수를 사용한 부분은 다음과 같다.

```
050     SOCKADDR_IN serveraddr;
051     ZeroMemory(&serveraddr, sizeof(serveraddr));
052     serveraddr.sin_family = AF_INET;
053     serveraddr.sin_port = htons(9000);
054     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
055     retval = bind(listen_sock, (SOCKADDR *)&serveraddr,
056                   sizeof(serveraddr));
056     if(retval == SOCKET_ERROR) err_quit("bind()");
```

50-51행: 소켓 주소 구조체 변수를 선언한 후 0으로 초기화한다. 윈도우 API 함수인 ZeroMemory() 또는 C 표준 함수인 memset()을 사용하면 된다.

52행: 인터넷 주소 체계를 사용한다는 의미로 AF_INET을 대입한다.

53행: 서버의 지역 포트 번호로 9000번을 설정한다. htons() 함수를 이용하여 바이트 정렬을 변경한 값을 대입한다.

54행: 서버의 지역 IP 주소를 설정한다. 서버의 경우 특정 IP 주소를 대입하기보다는 INADDR_ANY 값을 사용하는 것이 바람직하다. 서버가 두 개 이상의 IP 주소를 가진 경우(multihomed host), INADDR_ANY 값을 지역 주소로 설정하면 클라이언트가 어느 주소로 접속하든지 처리할 수 있다.

55-56행: bind() 함수를 호출한다. 두 번째 인자는 항상 (SOCKADDR *) 타입으로 변환해야 한다.



저자 한마디

크기 대입 방법

다음과 같은 코드를 종종 볼 수 있다.

```
SOCKADDR_IN serveraddr;
...
retval = bind(listen_sock, (SOCKADDR *)&serveraddr, sizeof(SOCKADDR_IN));
```

위의 코드는 전혀 문제가 없다. 그런데, 만약 servaddr 변수 타입을 바꾼다면 어떻게 될까? 이 경우, 다음 코드에서 보는 바와 같이 두 부분을 수정해야 한다.

```
SOCKADDR_IRDA serveraddr;
...
retval = bind(listen_sock, (SOCKADDR *)&serveraddr, sizeof(SOCKADDR_IRDA));
...
```

그러나 본문 예제 코드처럼 변수 크기를 직접 대입하도록 했다면 한 부분만 수정하면 된다. 간단하지만 이와 같은 코딩 스타일을 사용하면, 코드 수정 시 실수할 가능성을 줄일 수 있어 여러 모로 도움이 된다.

```
SOCKADDR_IRDA serveraddr;
...
retval = bind(listen_sock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));
```

listen() 함수

listen() 함수는 소켓과 결합된 TCP 포트 상태를 LISTENING으로 바꾸는 역할을 한다. 이는 클라이언트 접속을 받아들일 수 있는 상태가 됨을 의미한다.

```
int listen(
    SOCKET s,
    int backlog
);
```

성공: 0, 실패: SOCKET_ERROR

- s
클라이언트 접속을 수용할 목적으로 만든 소켓으로, bind() 함수에 의해 지역 IP 주소와 지역 포트 번호가 설정된 상태다.
- backlog
서버가 당장 처리하지 않더라도 접속 가능한 클라이언트의 개수다. 클라이언트의 접속 정보는 연결 큐(connection queue)에 저장되며, backlog는 이 연결 큐의 길이를 나타낸다고 볼 수 있다. 하부 프로토콜에서 지원 가능한 최대값을 사용하려면 SOMAXCONN 값을 대입한다.

TCPServer 예제에서 listen() 함수를 사용한 부분은 다음과 같다.

```
059     retval = listen(listen_sock, SOMAXCONN);
060     if(retval == SOCKET_ERROR) err_quit("listen()");
```

59-60행: backlog를 최대값으로 하여 listen() 함수를 호출하고 오류 처리를 한다.

accept() 함수

accept() 함수는 서버에 접속한 클라이언트와 통신할 수 있도록 새로운 소켓을 생성하여 리턴하는 역할을 한다. 또한 접속한 클라이언트의 IP 주소와 포트 번호(서버 입장에서는 원격 IP 주소와 원격 포트 번호, 클라이언트 입장에서는 지역 IP 주소와 지역 포트 번호)를 알려 준다.

서버가 당장 처리하지 않는다는 것은 곧이어 소개할 accept() 함수를 호출하지 않음을 뜻한다.
backlog 값을 바꾸려면 언제든지 listen() 함수를 다시 호출하면 된다.

여기서 접속했다는 사실은 TCP 프로토콜 수준에서 연결 설정이 성공적으로 이루어졌음을 의미한다.

```
SOCKET accept (
    SOCKET s,
    struct sockaddr* addr,
    int* addrlen
);
```

성공: 새로운 소켓, 실패: INVALID_SOCKET

- s
클라이언트 접속을 수용할 목적으로 만든 소켓이다.
- addr
소켓 주소 구조체 변수를 정의한 후, 이 변수의 주소값을 여기에 대입한다. accept() 함수는 addr이 가리키는 메모리 영역을 클라이언트의 IP 주소와 포트 번호로 채워 넣는다.
- addrlen
정수형 변수를 addr이 가리키는 메모리 영역의 크기로 초기화한 후, 이 변수의 주소값을 여기에 대입한다. accept() 함수가 리턴하면, 정수형 변수는 addrlen은 함수가 초기화한 메모리 크기값(바이트 단위)을 가진다.

클라이언트의 IP 주소와 포트 번호를 알 필요가 없다면 addr과 addrlen에 NULL을 사용하면 된다.

접속한 클라이언트가 없을 경우 accept() 함수는 서버를 대기 상태(wait state)로 만든다. 이때 작업 관리자(윈도우 NT/2000 계열)를 실행하여 CPU 사용률을 확인하면 0으로 표시된다(그림 4-22). 클라이언트가 접속하면 서버는 깨어나고 accept() 함수는 비로소 리턴하게 된다.



▶ [그림 4-22] 서버의 CPU 사용률

TCPServer 예제에서 accept() 함수를 사용한 부분은 다음과 같다.

```
062 // 데이터 통신에 사용할 변수
063 SOCKET client_sock;
064 SOCKADDR_IN clientaddr;
065 int addrlen;
...
068 while(1){
069 // accept()
070 addrlen = sizeof(clientaddr);
071 client_sock = accept(listen_sock, (SOCKADDR *)
    &clientaddr, &addrlen);
072 if(client_sock == INVALID_SOCKET){
073     err_display("accept()");
074     continue;
075 }
076 printf("\n[TCP 서버] 클라이언트 접속: IP 주소=%s, 포트 번호=%d\n",
077     inet_ntoa(clientaddr.sin_addr),
078     ntohs(clientaddr.sin_port));
079 // 클라이언트와 데이터 통신
080 while(1){
...
101 }
102
103 // closesocket()
104 closesocket(client_sock);
105 printf("[TCP 서버] 클라이언트 종료: IP 주소=%s, 포트 번호=%d\n",
106     inet_ntoa(clientaddr.sin_addr),
107     ntohs(clientaddr.sin_port));
107 }
```

- 63행: accept() 함수의 리턴값을 저장할 SOCKET 타입 변수다.
- 64행: accept() 함수의 두 번째 인자로 사용되며, accept() 함수 리턴 후 클라이언트의 IP 주소와 포트 번호가 여기에 저장된다.
- 65행: accept() 함수의 세 번째 인자로 사용된다.
- 68행: 일반적으로 서버는 계속 클라이언트 요청을 처리해야 하므로 무한 루프를 돈다.

70행: `accept()` 함수를 호출하기 전에, 세 번째 인자로 사용할 정수형 변수 `addrlen`을 소켓 주소 구조체 변수(`clientaddr`)의 크기로 초기화한다.

71–75행: `accept()` 함수를 호출하고 오류 처리를 한다. 이전에 사용한 소켓 함수와 달리 오류가 발생하면 `err_display()` 함수를 이용하여 화면에 구체적인 오류 메시지를 표시한 후 다시 67행으로 돌아간다. 심각한 오류가 아니라면 이와 같이 서버를 계속 구동하는 것이 바람직하다.

76–77행: 클라이언트의 IP 주소와 포트 번호를 화면에 출력한다. `inet_ntoa()`와 `ntohs()` 함수의 의미는 3장을 참조하기 바란다.

80–101행: `accept()` 함수가 리턴한 소켓을 이용하여 클라이언트와 통신을 한다. 이 부분은 데이터 전송 함수(`send()`, `recv()`)를 설명할 때 분석할 것이다.

104–106행: 클라이언트와 통신이 끝나면 소켓을 닫고, 종료한 클라이언트의 IP 주소와 포트 번호를 화면에 출력한다.



저자 한마디

값-결과 인자

`accept()` 함수의 세 번째 인자와 같이 함수 호출 전에 초기화하여 해당 함수에 값을 전달하고, 함수 호출 후에는 결과를 저장하고 있는 인자를 값-결과 인자(value-result argument 또는 value-result parameter)라 부른다. 이 글을 쓴 시점에서 가장 최근의 Platform SDK 문서(윈도우 서버 2003 버전까지 포함)를 찾아보면, 다음과 같은 `accept()` 함수 도움말을 볼 수 있다. 함수에 값을 전달하는 인자는 [in], 함수 호출 결과를 저장하는 인자는 [out]으로 표시되어 있다. `accept()` 함수의 세 번째 인자는 두 가지 목적으로 사용되므로 [in, out]으로 표시하는 것이 바람직하다.

Platform SDK: Windows Sockets 2

accept

The **accept** function permits an incoming connection attempt on a socket.

```
SOCKET accept(  
    SOCKET s,  
    struct sockaddr* addr,  
    int* addrlen  
);
```

Parameters

s
[in] Descriptor identifying a socket that has been placed in a listening state with the `listen` function. The connection is actually made with the socket that is returned by `accept`.

addr
[out] Optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the `addr` parameter is determined by the address family that was established when the socket from the `sockaddr` structure was created.

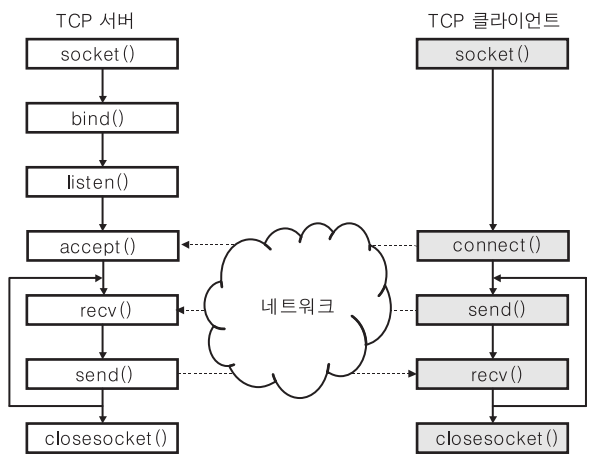
addrlen
[out] Optional pointer to an integer that contains the length of `addr`.

▶ Platform SDK 문서 – `accept()` 함수

2. 클라이언트 함수

일반적으로 TCP 클라이언트는 다음과 같은 순서로 소켓 함수를 호출한다([그림 4-23] 참조).

- ① `socket()` 함수를 이용하여 소켓을 생성한다.
- ② `connect()` 함수를 이용하여 서버에 접속한다.
- ③ `send()`, `recv()` 등의 데이터 전송 함수를 이용하여 서버와 통신을 수행한 후, `closesocket()` 함수를 이용하여 소켓을 닫는다.



▶ [그림 4-23] TCP 서버/클라이언트

TCP 서버/클라이언트 공통 함수(`socket()`, `closesocket()`, `send()`, `recv()`)를 제외한 유일한 클라이언트 함수인 `connect()`를 살펴보도록 하자.

connect() 함수

`connect()` 함수는 클라이언트가 서버에 접속하여 TCP 프로토콜 수준의 연결이 이루어지도록 한다.

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

성공: 0, 실패: SOCKET_ERROR

- s
서버와 통신을 하기 위해 만든 소켓이다.
- name
소켓 주소 구조체 변수를 서버 주소(원격 IP 주소와 원격 포트 번호)로 초기화한 후, 이 변수의 주소값을 여기에 대입한다.
- namelen
소켓 주소 구조체 변수의 길이(바이트 단위)를 대입한다.

클라이언트는 서버와 달리 bind() 함수를 호출하지 않는다. bind() 함수를 호출하지 않은 상태에서 connect() 함수를 호출하면 운영체제는 자동으로 지역 IP 주소와 지역 포트 번호를 설정한다. 이때 자동으로 할당되는 포트 번호는 운영체제에 따라 다를 수 있으며, 윈도우의 경우 1024~5000 범위 중 하나가 할당된다.

TCPClient 예제에서 connect() 함수를 사용한 부분은 다음과 같다.

```
070     SOCKADDR_IN serveraddr;  
071     serveraddr.sin_family = AF_INET;  
072     serveraddr.sin_port = htons(9000);  
073     serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");  
074     retval = connect(sock, (SOCKADDR *)&serveraddr,  
                        sizeof(serveraddr));  
075     if(retval == SOCKET_ERROR) err_quit("connect()");
```

70-73행: 소켓 주소 구조체 변수를 초기화한다. 서버와 클라이언트를 같은 컴퓨터에서 실행하는 경우 원격 IP 주소에 루프백 주소를 대입하면 된다.

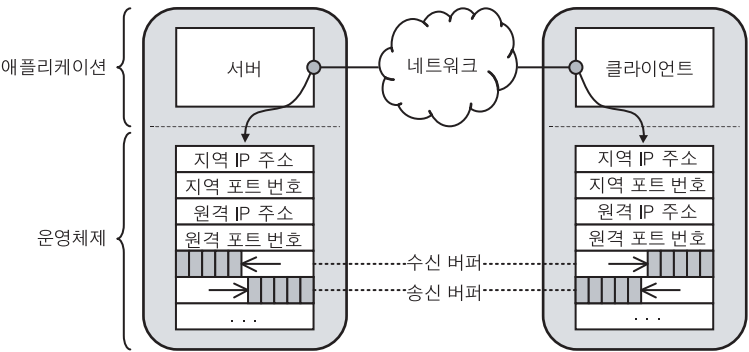
74-75행: connect() 함수를 호출하고 오류 처리를 한다.

3 데이터 전송 함수

데이터 전송 함수는 크게 데이터를 보내는 함수와 데이터를 받는 함수로 구분할 수 있다. 가장 기본이 되는 함수는 send()와 recv()며, 그 밖에 WSA*() 형태의 확장 함수가 존재한다. 여기서는 예제에 사용한 기본 함수만 살펴볼 것이다.

원속 버전 2에서는 확장 함수로 WSASend()와 WSARecv()를 제공한다.

데이터 전송 함수를 다루기 전에 소켓 데이터 구조체를 다시 살펴보자. [그림 4-24]는 TCP 소켓과 연관된 데이터 구조체를 나타낸 것이다(UDP 소켓은 6장을 참조하기 바란다). 각각 자신과 상대방의 IP 주소와 포트 번호 외에 데이터 송수신 버퍼가 있음을 알 수 있다. **송신 버퍼**(send buffer)는 데이터를 전송하기 전에 임시로 저장해두는 영역이고, **수신 버퍼**(receive buffer)는 받은 데이터를 애플리케이션이 처리하기 전까지 임시로 저장해두는 영역이다. 송신 버퍼와 수신 버퍼를 통틀어서 **소켓 버퍼**(socket buffer)라 부른다. 이 절에서 살펴볼 send()와 recv() 함수는 소켓을 통해 (간접적으로) 소켓 버퍼를 접근할 수 있도록 만든 함수라고 볼 수 있다.



▶ [그림 4-24] 소켓 데이터 구조체(2)

데이터 전송 함수를 사용할 때는 하부 프로토콜의 특성을 잘 알고 있어야 한다. 이 절에서 다루는 TCP 프로토콜은 애플리케이션이 보낸 데이터의 경계를 구분하지 않는다는 특징이 있다. 예를 들면, 클라이언트가 100, 200, 300바이트 데이터를 차례로 보낼 경우 서버가 100, 200, 300바이트 데이터의 경계를 구분하지 못하고 350, 250바이트 데이터를 읽을 수 있다. 따라서 TCP 서버/클라이언트를 작성할 때는 데이터 경계 구분을 위한 상호 약속이 필요하며, 이를 애플리케이션 수준에서 처리해야 한다.

send() 함수

send() 함수는 애플리케이션 데이터를 송신 버퍼에 복사함으로써 궁극적으로 하부 프로토콜(예를 들면, TCP/IP)에 의해 데이터가 전송되도록 한다. send() 함수는 데이터 복사가 성공적으로 이루어지면 곧바로 리턴하므로 send() 함수가 성공했다고 실제 데이터 전송이 완료된 것은 아니다.

```
int send(
    SOCKET s,
    const char* buf,
    int len,
    int flags
);
```

성공: 보낸 바이트 수, 실패: SOCKET_ERROR

- s
통신할 대상과 연결된(connected) 소켓이다.
- buf
보낼 데이터를 담고 있는 애플리케이션 버퍼의 주소다.
- len
보낼 데이터 크기(바이트 단위)다.
- flags
send() 함수의 동작을 바꾸는 옵션이다. 대부분 0을 사용하며, 드물게 MSG_DONTROUTE (7장 SO_DONTROUTE 참조)와 MSG_OOB(9장 참조)를 사용하는 경우가 있다.

send() 함수는 첫번째 인자로 사용한 소켓의 특성에 따라 다음과 같이 두 종류의 성공적인 리턴을 할 수 있다.

- 블로킹(blocking) 소켓
지금까지 생성한 모든 소켓은 블로킹 소켓이다. 블로킹 소켓에 대해 send() 함수를 호출할 때, 송신 버퍼의 여유 공간이 send() 함수의 세 번째 인자인 len보다 작을 경우 해당 프로세스는 대기 상태(wait state)가 된다. 송신 버퍼에 충분한 공간이 생기면 프로세스는 깨어나고, len 크기만큼 데이터 복사가 이루어진 후 send() 함수가 리턴한다. 이 경우 send() 함수의 리턴값은 len과 같게 된다.
- 년블로킹(Nonblocking) 소켓
9장에서 소개할 ioctlsocket() 함수를 이용하면 블로킹 소켓을 년블로킹 소켓으로 바꿀 수 있다. 년블로킹 소켓에 대해 send() 함수를 호출하면, 송신 버퍼의 여유 공간만큼 데이터를 복사한 후 실제 복사한 데이터 바이트 수를 리턴한다. 이 경우 send() 함수의 리턴값은 최소 1, 최대 len이 된다.

send() 함수 관련 코드는 recv() 함수를 공부한 후 분석할 것이다.

recv() 함수

recv() 함수는 수신 버퍼에 도착한 데이터를 애플리케이션 버퍼로 복사하는 역할을 한다.

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

성공: 받은 바이트 수 또는 0(연결 종료시), 실패: SOCKET_ERROR

- s
통신할 대상과 연결된(connected) 소켓이다.
- buf
받은 데이터를 저장할 애플리케이션 버퍼의 주소다.
- len
수신 버퍼로부터 복사할 최대 데이터 크기(바이트 단위)다. 이 값은 buf가 가리키는 버퍼의 크기보다 크지 않아야 한다.
- flags
recv() 함수의 동작을 바꾸는 옵션이다. 대부분 0을 사용하며, 드물게 MSG_PEEK와 MSG_OOB(9장 참조)를 사용하는 경우가 있다. recv() 함수는 수신 버퍼의 데이터를 애플리케이션 버퍼로 복사한 후 해당 데이터를 수신 버퍼에서 삭제한다. 그러나 MSG_PEEK 옵션을 사용하면 수신 버퍼에 데이터가 계속 남아있게 된다.

recv() 함수는 두 종류의 성공적인 리턴을 할 수 있다.

- 수신 버퍼에 데이터가 도달한 경우
recv() 함수의 세 번째 인자인 len보다 크지 않은 범위 내에서 가능한 많은 데이터를 애플리케이션 버퍼로 복사한다. 이 경우 복사한 바이트 수가 리턴되며, 가능한 최대 리턴값은 len이 된다.
- 접속이 정상 종료된 경우
상대 애플리케이션이 closesocket() 함수를 사용하여 접속을 종료하면, TCP 프로토콜 수준에서 접속 종료를 위한 패킷 교환 절차가 이루어진다. 이 경우 recv() 함수는 0을 리턴한다. recv() 함수의 리턴값이 0인 경우를 정상 종료(normal close = graceful close)라 부른다.

TCP 종료 시 FIN, ACK, FIN, ACK 네 개의 패킷 교환이 일어난다. 그러나 때로는 FIN, FIN/ACK, ACK 세 개의 패킷이 교환되기도 한다.

먼저 closesocket() 함수를 호출한 소켓의 TCP 포트는 FIN_WAIT 상태를 거친 후 사라진다(그림 4-16). FIN_WAIT 상태에 머무르는 시간은 TCP/IP 구현마다 다르지만 일반적으로 5분을 넘지 않는다.

recv() 함수 사용 시 특히 주의할 점은 세 번째 인자인 len으로 지정한 크기보다 작은 데이터가 애플리케이션 버퍼로 복사될 수 있다는 사실이다. 이는 TCP가 메시지 경계를 구분하지 않는다는 특성에 기인한 것이다. 따라서 자신이 받을 데이터의 크기를 미리 알고 있다면, 이 크기만큼 받을 때까지 recv() 함수를 여러 번 호출해야 한다. 예제에서는 사용자 정의 함수인 recvn()을 정의해서 처리하고 있다. 행 단위로 분석하면 다음과 같다([그림 4-25] 참조).

```

037 int recvn(SOCKET s, char *buf, int len, int flags)
038 {
039     int received;
040     char *ptr = buf;
041     int left = len;
042
043     while(left > 0){
044         received = recv(s, ptr, left, flags);
045         if(received == SOCKET_ERROR)
046             return SOCKET_ERROR;
047         else if(received == 0)
048             break;
049         left -= received;
050         ptr += received;
051     }
052
053     return (len - left);
054 }

```

37행: recvn() 함수 인자는 recv() 함수와 동일하다.

39행: recv() 함수의 리턴값을 저장하는 변수다.

40행: 포인터 변수 ptr이 애플리케이션 버퍼의 시작 주소를 가리키도록 한다. 데이터를 읽을 때마다 ptr 변수는 증가한다.

41행: left 변수는 아직 읽지 않은 데이터 크기를 나타낸다.

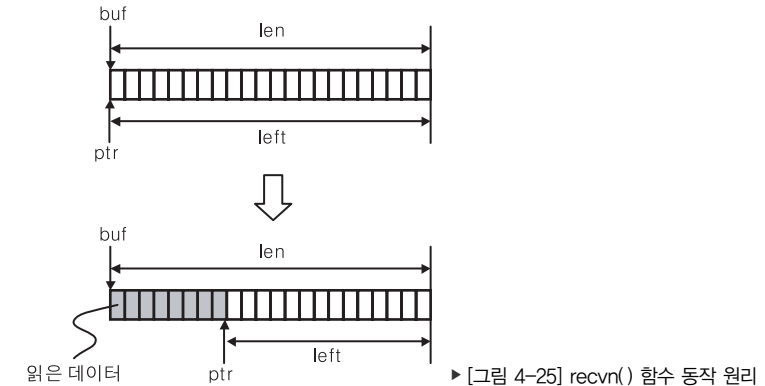
43행: 아직 읽지 않은 데이터가 있다면 계속 루프를 돈다.

44-46행: recv() 함수를 호출하고 오류가 발생하면 리턴한다.

47-48행: recv() 함수 리턴값이 0이면(정상 종료), 상대가 데이터를 더 보내지 않을 것이므로 루프를 빠져나간다.

49-50행: ptr, left 변수를 갱신한다.

53행: 읽은 바이트 수를 리턴한다. 정상 종료를 제외하면 left 변수는 항상 0이므로 리턴값은 len이 된다.



TCPClient 예제에서 send(), recv() 함수를 사용한 부분은 다음과 같다.

```

078 char buf[BUFSIZE+1];
079 int len;
...
082 while(1){
083     // 데이터 입력
084     ZeroMemory(buf, sizeof(buf));
085     printf("\n[보낼 데이터]");
086     if(fgets(buf, BUFSIZE+1, stdin) == NULL)
087         break;
088
089     // '\n' 문자 제거
090     len = strlen(buf);
091     if(buf[len-1] == '\n')
092         buf[len-1] = '\0';
093     if(strlen(buf) == 0)
094         break;
095
096     // 데이터 보내기
097     retval = send(sock, buf, strlen(buf), 0);
098     if(retval == SOCKET_ERROR){
099         err_display("send()");
100         break;
101     }
102     printf("[TCP 클라이언트] %d바이트를 보냈습니다.\n", retval);

```

```
103
104     // 데이터 받기
105     retval = recv(sock, buf, retval, 0);
106     if(retval == SOCKET_ERROR){
107         err_display("recv()");
108         break;
109     }
110     else if(retval == 0)
111         break;
112
113     // 받은 데이터 출력
114     buf[retval] = '\0';
115     printf("[TCP 클라이언트] %d바이트를 받았습니다.\n", retval);
116     printf("[받은 데이터] %s\n", buf);
117 }
```

78행: 사용자가 입력한 문자열(=보낼 데이터) 또는 서버로부터 받은 데이터를 저장할 버퍼다.

79행: 입력 문자열 길이를 계산할 때 사용한다.

84-87행: fgets() 함수를 이용하여 사용자로부터 문자열을 입력받는다.

90-92행: '\n' 문자를 제거한다. 데이터 출력 시 줄바꿈 여부를 서버가 결정하도록 하기 위함이다.

93-94행: 글자를 입력하지 않고 Enter 키만 눌렀다면 루프를 빠져나간다.

97-102행: send() 함수를 호출하고 오류 처리를 한다. 블로킹 소켓을 사용하고 있으므로 send() 함수의 리턴값은 strlen(buf) 값과 같게 된다.

105-111행: recvn() 함수를 호출하고 오류 처리를 한다. 서버가 보낼 데이터의 크기를 미리 알고 있으므로 recvn() 함수를 사용하는 것이 편리하다.

114-116행: 받은 데이터 끝에 '\0' 을 추가하여 화면에 출력한다.

TCPServer 예제에서 send(), recv() 함수를 사용한 부분은 다음과 같다. 여기서 사용한 소켓(client_sock)은 accept() 함수의 리턴값으로 생성된 것임을 유의하도록 하자.

```
066     char buf[BUFSIZE+1];
...
080     while(1){
081         // 데이터 받기
082         retval = recv(client_sock, buf, BUFSIZE, 0);
083         if(retval == SOCKET_ERROR){
084             err_display("recv()");
085             break;
086         }
087         else if(retval == 0)
088             break;
089
090         // 받은 데이터 출력
091         buf[retval] = '\0';
092         printf("[TCP/%s:%d] %s\n",
093             inet_ntoa(clientaddr.sin_addr),
094             ntohs(clientaddr.sin_port), buf);
095
096         // 데이터 보내기
097         retval = send(client_sock, buf, retval, 0);
098         if(retval == SOCKET_ERROR){
099             err_display("send()");
100             break;
101         }
102     }
```

66행: 받은 데이터를 저장할 애플리케이션 버퍼다.

80행: recv() 함수의 리턴값이 0(정상 종료) 또는 SOCKET_ERROR(오류 발생)가 될 때까지 계속 루프를 돌며 데이터를 수신한다.

82-88행: recv() 함수를 호출하고 오류 처리를 한다. 서버는 받을 데이터 크기를 미리 알 수 없으므로 recvn() 함수를 사용할 수 없다.

91-93행: 받은 데이터 끝에 '\0' 을 추가하여 화면에 출력한다.

96-100행: send() 함수를 호출하고 오류 처리를 한다.



애플리케이션 프로토콜과 메시지 설계

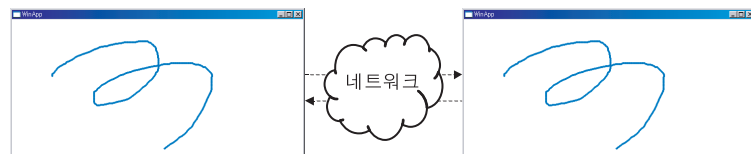
TCP 서버/클라이언트의 기본 구조는 정형화되어 있어서 일단 각 함수의 의미를 이해하고 나면 뼈대가 되는 코드는 그대로 복사해서 사용할 수 있다. 앞에서 다룬 TCP 서버/클라이언트 예제는 기본 함수를 모두 갖추고 있으므로 이것을 기본으로 다른 기능을 덧붙여도 된다. 그렇다면 애플리케이션의 고유한 기능을 결정하는 부분은 어디일까? 바로 데이터 처리 부분이다. 어떤 데이터를 어떤 형식으로 주고받을지 그리고 어떻게 처리할지를 결정하는 것은 네트워크 애플리케이션 개발자가 해야 할 중요한 작업 중 하나다.

이 절에서는 TCP 서버/클라이언트 응용으로 간단한 파일 전송 프로그램을 작성할 것이다. 우선 애플리케이션 프로토콜의 개념과 메시지 설계 시 주의할 점을 다룬 후 예제를 작성하도록 하자.

1 애플리케이션 프로토콜

애플리케이션 프로토콜은 애플리케이션 수준(application-level)에서 주고받는 데이터의 형식과 의미 그리고 처리 방식 등을 정의한 프로토콜이다. 일단 애플리케이션 프로토콜이 결정되면 소켓 프로그래밍을 이용하여 데이터를 주고받도록 작성하면 되는 것이다.

애플리케이션 프로토콜의 기본은 주고받을 메시지 형식을 정하는 것이다. [그림 4-26]을 예로 들어보자. 이 프로그램은 네트워크를 통해 자신의 화면과 상대방 화면에 동시에 그림을 그릴 수 있는 기능을 제공한다.



▶ [그림 4-26] 네트워크 그림판

두 프로그램이 주고 받아야 할 요소는 다음과 같다.

- **선의 시작과 끝점**

직선을 연결하여 곡선을 그리므로 매번 직선 정보를 네트워크로 전달한다.

- **두께와 색상**

직선의 그릴 때 필요한 속성이다.

주고받을 데이터를 구조체로 표현하면 다음과 같다.

```
struct DrawMessage1
{
    int x1, y1; // 선의 시작점
    int x2, y2; // 선의 끝점
    int width;  // 선 두께
    int color;  // 선 색상
};
```

만약 이 프로그램에 원 그리기 기능을 추가해야 한다면 다음과 같은 요소가 필요할 것이다.

- **원의 중심 좌표**

- **원의 반지름**

- **내부 색상**

원을 채울 때 사용할 색상이다.

- **두께와 색상**

테두리 선을 그릴 때 필요한 속성이다.

주고받을 데이터를 구조체로 표현하면 다음과 같다.

```
struct DrawMessage2
{
    int x1, y1; // 원의 중심 좌표
    int r;      // 원의 반지름
    int fillcolor; // 내부 색상
    int width;  // 선 두께
    int color;  // 선 색상
};
```

이와 같이 정의하면 네트워크를 통해 데이터를 받은 쪽에서는 어떤 타입(선 또는 원)인지 구분할 수 없으므로 다음과 같이 타입을 나타내는 필드를 추가해야 한다.

```
struct DrawMessage1
{
    int type;    // = LINE
    int x1, y1; // 선의 시작점
    int x2, y2; // 선의 끝점
    int width;  // 선 두께
    int color;  // 선 색상
};

struct DrawMessage2
{
    int type;    // = CIRCLE
    int x1, y1;  // 원의 중심 좌표
    int r;       // 원의 반지름
    int fillcolor; // 내부 색상
    int width;   // 선 두께
    int color;   // 선 색상
};
```

2 메시지 설계

통신 양단이 주고받을 데이터 요소를 구조체로 정의하는 것만으로는 아직 충분하지 않다. 메시지를 설계할 때 고려해야 할 사항을 알아보도록 하자.

경계 구분

TCP같이 메시지 경계 구분을 하지 않는 프로토콜을 사용할 경우, 애플리케이션 수준에서 이를 처리해야 한다. 다음과 같이 세 가지 방법을 생각해볼 수 있다. 편의상 송신자와 수신자에서 처리해야 할 작업으로 구분하였다.

[송신자]

- ① 항상 고정 길이 데이터를 보낸다.
- ② 경계 구분을 위해 특별한 표시(EOR, End Of Record)를 삽입한다.
- ③ 보낼 데이터 길이를 고정 길이 데이터로 보낸 후, 가변 길이 데이터를 이어서 보낸다.

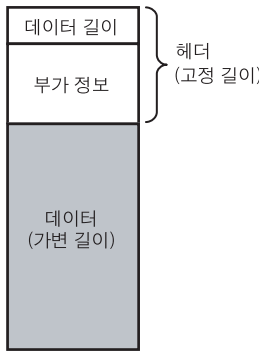
[수신자]

- ① 항상 고정 길이 데이터를 받는다.
- ② EOR이 나올 때까지 데이터를 읽은 후 처리한다.
- ③ 고정 길이 데이터를 읽어 뒤따라올 데이터의 길이를 알아낸다. 이 길이만큼 가변 길이 데이터를 읽어 처리한다.

방법 ①은 송신자와 수신자가 처리하기에 가장 간편하지만, 미래에 사용할 가장 긴 데이터를 감안하여 고정 길이를 정해야 한다는 문제가 있다. 이렇게 하면 길이가 짧은 데이터를 주고받을 때는 낭비하는 부분이 생긴다.

방법 ②는 송신자와 수신자 쪽에서 데이터를 처리하기 쉽지 않다는 문제가 있다. 첫째, 송신자 쪽에서는 데이터 중간에 EOR과 똑같은 패턴이 있을 경우를 특별하게 처리해주어야 한다. 둘째, 수신자 쪽에서는 데이터를 한 바이트씩 읽어서 처리해야 하므로 효율성이 떨어지게 된다. 또한 데이터에 속한 EOR인지, 경계를 나타내는 EOR인지 구분하는 작업도 해야 한다.

방법 ③은 구현하기도 쉽고 효율성도 높으므로 일반적으로 많이 사용한다. [그림 4-27]은 이 방법을 적용할 경우 메시지 구조를 보여준다. 이 경우 두 번의 `recvn()` 함수 호출로 데이터를 읽을 수 있다.



▶ [그림 4-27] 메시지 구조

바이트 정렬

서로 다른 바이트 정렬 방식을 사용하는 시스템 사이에 데이터를 교환할 때는 바이트 정렬 방식을 통일해야 한다. 특별한 전제가 없다면 빅 엔디안 방식으로 통일하는 것이 좋다. 바이트 정렬 관련 함수는 3장에서 다루었으므로 참고하기 바란다.

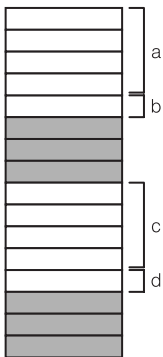
`recv()`가 아니고 `recvn()`임을 유의하자. `recvn()`은 2절에서 소개한 사용자 정의 함수다.

멤버 정렬

멤버 정렬(member alignment)이란 구조체(공용체, 클래스 포함) 멤버의 시작 주소에 대한 제약 사항을 의미한다. 간단한 예를 들어보자. 다음과 같이 메시지를 정의하여 보내면 sizeof(msg)는 10(=4+1+4+1)이 아닌 16이 된다. [그림 4-28]은 변수 msg의 메모리 구조를 보여준다.

```
struct MyMessage
{
    int a; // 4바이트
    char b; // 1바이트
    int c; // 4바이트
    char d; // 1바이트
};

MyMessage msg;
//...
send(sock, (char *)&msg, sizeof(msg), 0);
```



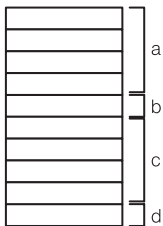
▶ [그림 4-28] msg 변수 메모리 구조

양쪽 프로그램이 동일한 구조체 멤버 정렬 방식을 사용한다면 이와 같이 메시지를 전송해도 문제가 되지 않는다. 특별한 이유로 인해 정확히 10바이트를 보내려고 한다면 다음과 같이 #pragma pack 컴파일러 명령을 사용하면 된다. 이때 변수 msg의 메모리 구조는 [그림 4-29]와 같다.

```
#pragma pack(1) // 멤버 정렬 방식 전환: 1바이트 경계
struct MyMessage
{
```

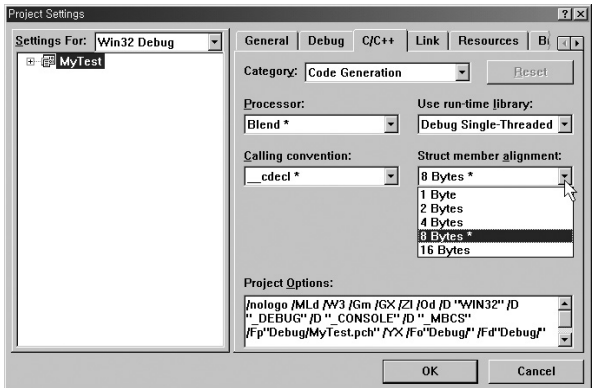
```
    int a; // 4바이트
    char b; // 1바이트
    int c; // 4바이트
    char d; // 1바이트
};
#pragma pack() // 디폴트 멤버 정렬 방식으로 복귀

MyMessage msg;
//...
send(sock, (char *)&msg, sizeof(msg), 0);
```

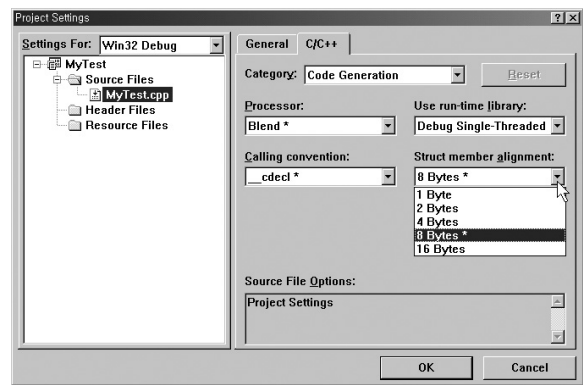


▶ [그림 4-29] msg 변수 메모리 구조

#pragma pack 컴파일러 명령은 구조체 단위로 멤버 정렬 방식을 설정할 때 사용한다. 일반적으로 권장하지는 않지만 프로젝트 전체 혹은 파일 단위로 디폴트 멤버 정렬 방식을 변경할 수도 있다. 비주얼 C++의 [Project]→[Settings...] 대화상자에서 [C/C++]→[Code Generation]→[Struct member alignment] 옵션을 바꾸면 된다.



▶ [그림 4-30] 디폴트 멤버 정렬 방식 변경 - 프로젝트 전체



▶ [그림 4-31] 디폴트 멤버 정렬 방식 변경 - 파일 단위

3. [실습] 파일 전송 프로그램

TCP/IP를 이용한 파일 전송 프로그램을 작성해보자. 서버는 파일을 받기만 하고, 클라이언트는 보내기만 한다고 가정한다. 클라이언트가 서버에 보낼 메시지는 [그림 4-32]와 같이 정의하였다. 처음 256바이트에 파일 이름을, 다음 4바이트에 파일 크기를 보냄으로써 서버에 파일 정보를 넘겨준다. 마지막으로 실제 파일 데이터를 전송하면, 서버는 자신이 받을 크기를 미리 알고 있으므로 전송 완료 여부를 알 수 있다.



▶ [그림 4-32] 메시지 형식

8 따라하기

- 1 [그림 4-32]의 메시지 형식을 토대로 파일 전송 서버(FileReceiver)와 클라이언트(FileSender)를 작성하도록 하자. 이 예제는 행 단위 분석을 따로 하지 않을 것이므로 실습을 통해 동작을 파악한 후 각자 분석해 보도록 하자. 2절에서 배운 소켓 함수와 더불어 C 표준 파일 입출력 함수만 사용하였으므로, 주석을 참고하면서 두 코드를 대조하면 쉽게 동작 과정을 이해할 수 있을 것이다.

FileSender와 FileReceiver라는 콘솔 애플리케이션 프로젝트 두 개를 생성한 후 각각 다음과 같이 입력한다. 프로젝트 생성 옵션과 컴파일, 링크 방법은 1장 예제와 동일하다.

[예제] FileSender.cpp

```
001 #include <winsock2.h>
002 #include <stdlib.h>
003 #include <stdio.h>
004
005 #define BUFSIZE 4096
006
007 // 소켓 함수 오류 출력 후 종료
008 void err_quit(char *msg)
009 {
010     LPVOID lpMsgBuf;
011     FormatMessage(
012         FORMAT_MESSAGE_ALLOCATE_BUFFER|
013         FORMAT_MESSAGE_FROM_SYSTEM,
014         NULL, WSAGetLastError(),
015         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
016         (LPTSTR)&lpMsgBuf, 0, NULL);
017     MessageBox(NULL, (LPCTSTR)lpMsgBuf, msg, MB_ICONERROR);
018     LocalFree(lpMsgBuf);
019     exit(-1);
020 }
021
022 // 소켓 함수 오류 출력
023 void err_display(char *msg)
024 {
025     LPVOID lpMsgBuf;
026     FormatMessage(
027         FORMAT_MESSAGE_ALLOCATE_BUFFER|
028         FORMAT_MESSAGE_FROM_SYSTEM,
029         NULL, WSAGetLastError(),
030         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
031         (LPTSTR)&lpMsgBuf, 0, NULL);
```

이 메시지 형식은 [그림 4-27]을 응용한 것임을 알 수 있을 것이다.

```

032     printf("[%s] %s", msg, (LPCTSTR)lpMsgBuf);
033     LocalFree(lpMsgBuf);
034 }
035
036 int main(int argc, char* argv[])
037 {
038     int retval;
039
040     if(argc < 2){
041         fprintf(stderr, "Usage: %s [FileName]\n", argv[0]);
042         return -1;
043     }
044
045     // 윈속 초기화
046     WSADATA wsa;
047     if(WSAStartup(MAKEWORD(2,2), &wsa) != 0)
048         return -1;
049
050     // socket()
051     SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
052     if(sock == INVALID_SOCKET) err_quit("socket()");
053
054     // connect()
055     SOCKADDR_IN serveraddr;
056     serveraddr.sin_family = AF_INET;
057     serveraddr.sin_port = htons(9000);
058     serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");
059     retval = connect(sock, (SOCKADDR *)&serveraddr,
060                     sizeof(serveraddr));
061
062     if(retval == SOCKET_ERROR) err_quit("connect()");
063
064     // 파일 열기
065     FILE *fp = fopen(argv[1], "rb");
066     if(fp == NULL){
067         perror("파일 입출력 오류");
068     }

```

```

066         return -1;
067     }
068
069     // 파일 이름 보내기
070     char filename[256];
071     ZeroMemory(filename, 256);
072     sprintf(filename, argv[1]);
073     retval = send(sock, filename, 256, 0);
074     if(retval == SOCKET_ERROR) err_quit("send()");
075
076     // 파일 크기 얻기
077     fseek(fp, 0, SEEK_END);
078     int totalbytes = ftell(fp);
079
080     // 파일 크기 보내기
081     retval = send(sock, (char *)&totalbytes,
082                 sizeof(totalbytes), 0);
083     if(retval == SOCKET_ERROR) err_quit("send()");
084
085     // 파일 데이터 전송에 사용할 변수
086     char buf[BUFSIZE];
087     int numread;
088     int numtotal = 0;
089
090     // 파일 데이터 보내기
091     rewind(fp); // 파일 포인터를 제일 앞으로 이동
092     while(1){
093         numread = fread(buf, 1, BUFSIZE, fp);
094         if(numread > 0){
095             retval = send(sock, buf, numread, 0);
096             if(retval == SOCKET_ERROR){
097                 err_display("send()");
098                 break;
099             }
100             numtotal += numread;

```

```

100     }
101     else if(numread == 0 && numtotal == totalbytes){
102         printf("파일 전송 완료!: %d 바이트\n", numtotal);
103         break;
104     }
105     else{
106         perror("파일 입출력 오류");
107         break;
108     }
109 }
110 fclose(fp);
111
112 // closesocket()
113 closesocket(sock);
114
115 // 윈속 종료
116 WSACleanup();
117 return 0;
118 }

```

[예제] FileReceiver.cpp

```

001 #include <winsock2.h>
002 #include <stdlib.h>
003 #include <stdio.h>
004
005 #define BUFSIZE 4096
006
007 // 소켓 함수 오류 출력 후 종료
008 void err_quit(char *msg)
009 {
010     LPVOID lpMsgBuf;
011     FormatMessage(
012         FORMAT_MESSAGE_ALLOCATE_BUFFER|
013         FORMAT_MESSAGE_FROM_SYSTEM,

```

```

014         NULL, WSAGetLastError(),
015         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
016         (LPTSTR)&lpMsgBuf, 0, NULL);
017     MessageBox(NULL, (LPCTSTR)lpMsgBuf, msg, MB_ICONERROR);
018     LocalFree(lpMsgBuf);
019     exit(-1);
020 }
021
022 // 소켓 함수 오류 출력
023 void err_display(char *msg)
024 {
025     LPVOID lpMsgBuf;
026     FormatMessage(
027         FORMAT_MESSAGE_ALLOCATE_BUFFER|
028         FORMAT_MESSAGE_FROM_SYSTEM,
029         NULL, WSAGetLastError(),
030         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
031         (LPTSTR)&lpMsgBuf, 0, NULL);
032     printf("[%s] %s", msg, (LPCTSTR)lpMsgBuf);
033     LocalFree(lpMsgBuf);
034 }
035
036 // 사용자 정의 데이터 수신 함수
037 int recvn(SOCKET s, char *buf, int len, int flags)
038 {
039     int received;
040     char *ptr = buf;
041     int left = len;
042
043     while(left > 0){
044         received = recv(s, ptr, left, flags);
045         if(received == SOCKET_ERROR)
046             return SOCKET_ERROR;
047         else if(received == 0)
048             break;

```



```

049     left -= received;
050     ptr += received;
051 }
052
053     return (len - left);
054 }
055
056 int main(int argc, char* argv[])
057 {
058     int retval;
059
060     // 윈속 초기화
061     WSADATA wsa;
062     if(WSAStartup(MAKEWORD(2,2), &wsa) != 0)
063         return -1;
064
065     // socket()
066     SOCKET listen_sock = socket(AF_INET, SOCK_STREAM, 0);
067     if(listen_sock == INVALID_SOCKET) err_quit("socket()");
068
069     // bind()
070     SOCKADDR_IN serveraddr;
071     ZeroMemory(&serveraddr, sizeof(serveraddr));
072     serveraddr.sin_family = AF_INET;
073     serveraddr.sin_port = htons(9000);
074     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
075     retval = bind(listen_sock, (SOCKADDR *)&serveraddr,
076                  sizeof(serveraddr));
077     if(retval == SOCKET_ERROR) err_quit("bind()");
078
079     // listen()
080     retval = listen(listen_sock, SOMAXCONN);
081     if(retval == SOCKET_ERROR) err_quit("listen()");
082
083     // 데이터 통신에 사용할 변수

```

```

083     SOCKET client_sock;
084     SOCKADDR_IN clientaddr;
085     int addrlen;
086     char buf[BUFSIZE];
087
088     while(1){
089         // accept()
090         addrlen = sizeof(clientaddr);
091         client_sock = accept(listen_sock,
092                             (SOCKADDR *)&clientaddr, &addrlen);
093
094         if(client_sock == INVALID_SOCKET){
095             err_display("accept()");
096             continue;
097         }
098
099         printf("\nFileSender 접속: IP 주소=%s, 포트 번호=%d\n",
100              inet_ntoa(clientaddr.sin_addr),
101              ntohs(clientaddr.sin_port));
102
103         // 파일 이름 받기
104         char filename[256];
105         ZeroMemory(filename, 256);
106         retval = recv(client_sock, filename, 256, 0);
107         if(retval == SOCKET_ERROR){
108             err_display("recv()");
109             closesocket(client_sock);
110             continue;
111         }
112         printf("-> 받을 파일 이름: %s\n", filename);
113
114         // 파일 크기 받기
115         int totalbytes;
116         retval = recv(client_sock, (char *)&totalbytes,
117                      sizeof(totalbytes), 0);
118         if(retval == SOCKET_ERROR){
119             err_display("recv()");

```

```

115         closesocket(client_sock);
116         continue;
117     }
118     printf("-> 받을 파일 크기: %d\n", totalbytes);
119
120     // 파일 열기
121     FILE *fp = fopen(filename, "wb");
122     if(fp == NULL){
123         perror("파일 입출력 오류");
124         closesocket(client_sock);
125         continue;
126     }
127
128     // 파일 데이터 받기
129     int numtotal = 0;
130     while(1){
131         retval = recvn(client_sock, buf, BUFSIZE, 0);
132         if(retval == SOCKET_ERROR){
133             err_display("recv()");
134             break;
135         }
136         else if(retval == 0)
137             break;
138         else{
139             fwrite(buf, 1, retval, fp);
140             if(ferror(fp)){
141                 perror("파일 입출력 오류");
142                 break;
143             }
144             numtotal += retval;
145         }
146     }
147     fclose(fp);
148
149     // 전송 결과 출력

```

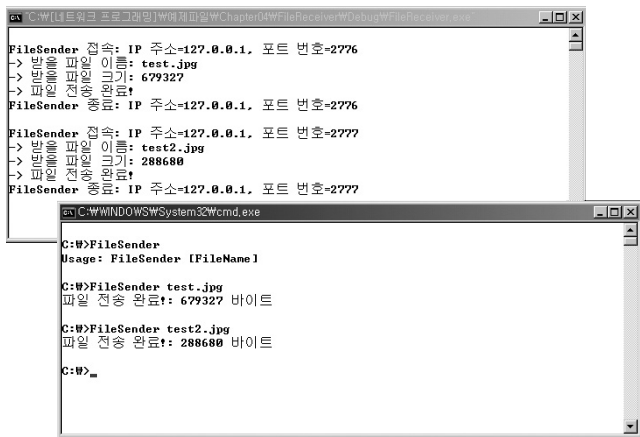
```

150         if(numtotal == totalbytes)
151             printf("-> 파일 전송 완료!\n");
152         else
153             printf("-> 파일 전송 실패!\n");
154
155         // closesocket()
156         closesocket(client_sock);
157         printf("FileSender 종료: IP 주소=%s, 포트 번호=%d\n",
158             inet_ntoa(clientaddr.sin_addr),
159             ntohs(clientaddr.sin_port));
160
161         // closesocket()
162         closesocket(listen_sock);
163
164         // 윈속 종료
165         WSACleanup();
166         return 0;
167     }

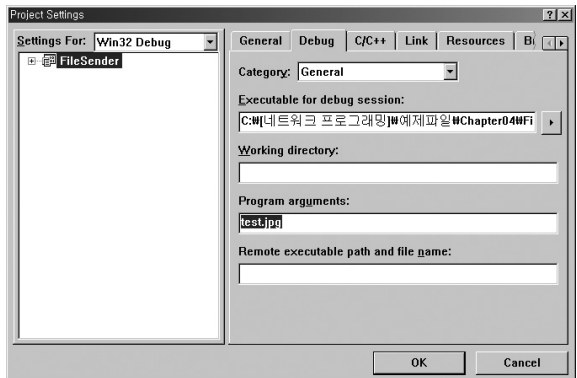
```

2 실습 환경에 따라 일부 코드를 변경하도록 한다. 예제 코드는 실습의 편의를 위해 두 프로그램이 같은 컴퓨터에서 실행되는 경우를 가정하고 있으므로, 서로 다른 컴퓨터에서 실행하려면 FileSender.cpp 파일의 54행을 FileReceiver의 IP 주소로 변경해야 한다.

컴파일, 링크 후 실행 결과는 [그림 4-33]과 같다. FileSender는 전송할 파일 이름을 명령행 인자로 받으므로, 명령 프롬프트에서 실행하였다. 비주얼 C++ 통합 환경에서 실행하려면, [Project]→[Settings...]→[Debug]→[Program arguments] 부분에 파일명을 입력하면 된다([그림 4-34]). 이때 전송할 파일의 경로명을 주지 않을 경우, FileSender 프로세스의 현재 디렉토리(current directory) 위치에서 찾게 되는데, 비주얼 C++ 통합 환경에서는 프로젝트 폴더가 현재 디렉토리가 된다는 점에 주의하자.



▶ [그림 4-33] 실행 화면)



▶ [그림 4-34] 비주얼 C++ 통합 환경에서 명령행 인자 사용하기



요약

- 1 TCP 서버에 TCP 클라이언트가 접속하면 매번 새로운 소켓이 생성되며, 이 소켓이 실제 데이터 전송에 사용된다.
- 2 TCP/IP를 이용하여 애플리케이션이 통신을 수행하기 위해서는 프로토콜, 지역(local) IP 주소와 지역 포트 번호, 원격(remote) IP 주소와 원격 포트 번호 등이 결정되어야 한다.
- 3 TCP 서버의 주요 함수는 socket(), bind(), listen(), accept(), recv(), send(), closesocket()이다.
- 4 bind() 함수는 서버의 지역 IP 주소와 지역 포트 번호를 결정한다.
- 5 listen() 함수는 소켓과 결합된 TCP 포트 상태를 LISTENING으로 바꾼다.
- 6 accept() 함수는 서버에 접속한 클라이언트와 통신할 수 있도록 새로운 소켓을 생성하여 리턴한다.
- 7 TCP 클라이언트의 주요 함수는 socket(), connect(), recv(), send(), closesocket()이다.
- 8 connect() 함수는 클라이언트가 서버에게 접속하여 TCP 프로토콜 수준의 연결이 이루어지도록 한다.
- 9 send() 함수는 애플리케이션 데이터를 송신 버퍼에 복사함으로써 궁극적으로 하부 프로토콜(예를 들면, TCP/IP)에 의해 데이터가 전송되도록 한다.
- 10 send() 함수는 첫번째 인자로 사용한 소켓의 특성(블로킹, 논블로킹)에 따라 두 종류의 성공적인 리턴을 할 수 있다.
- 11 recv() 함수는 수신 버퍼에 도착한 데이터를 애플리케이션 버퍼로 복사한다.
- 12 recv() 함수는 상황에 따라 두 종류의 성공적인 리턴을 할 수 있다.
- 13 애플리케이션 프로토콜은 애플리케이션 수준(application-level)에서 주고받는 데이터의 형식과 의미 그리고 처리 방식 등을 정의한 프로토콜이다.
- 14 TCP와 같이 메시지 경계 구분을 하지 않는 프로토콜을 사용할 경우, 애플리케이션 수준에서 이를 처리해야 한다.
- 15 서로 다른 바이트 정렬 방식을 사용하는 시스템 사이에 데이터를 교환할 때는 바이트 정렬 방식을 통일해야 한다.
- 16 멤버 정렬(member alignment)이란 구조체(공용체, 클래스 포함) 멤버의 시작 주소에 대한 제약 사항을 의미한다. 양쪽 프로그램이 동일한 구조체 멤버 정렬 방식을 사용하지 않을 경우 문제가 될 수 있다.

* 연습 문제

- 1 TCPServer 예제를 수정하여, 받은 데이터를 영문자열로 간주하고 모두 대문자로 바꾼 후 다시 클라이언트로 보내도록 하라.
- 2 TCPServer 예제를 수정하여, 시작할 때 명령행 인자(command line argument)로 포트 번호를 입력할 수 있도록 하라.
- 3 TCPClient 예제를 수정하여, 시작할 때 명령행 인자로 접속할 서버의 IP 주소(또는 도메인 이름)와 포트 번호를 입력할 수 있도록 하라.
- 4 listen() 함수의 backlog 값에 따른 연결 큐(connection queue)의 길이를 추정할 수 있는 프로그램을 작성하라. 자신의 시스템에서 backlog 값이 SOMAXCONN인 경우 최대 몇 개의 클라이언트가 연결을 성공할 수 있는가?
- 5 FileSender, FileReceiver 예제를 수정하여, 전송 상황을 % 단위로 표시하도록 하라. 또한 파일 전송이 끝나면 총 소요 시간을 표시하도록 하라.
- 6 이어받기 기능을 위한 애플리케이션 프로토콜을 설계하고, FileSender, FileReceiver 예제를 수정하여 구현하라. 단, 구현을 간단히 하기 위해 파일 이름이 같으면 같은 파일로 간주한다(이어받기란 중간에 전송이 실패한 파일을 재전송할 경우, 마지막으로 성공한 바이트 다음 위치부터 전송하도록 하는 기능이다).
- 7 명령행 인자로 IP 주소(또는 도메인 이름)와 두 개의 포트 번호(최소값, 최대값)를 입력받아, 특정 컴퓨터의 연결 대기 중인(listening) TCP 포트를 모두 알아내는 프로그램을 작성하라. 단, 속도가 빠르게 구현할 필요는 없다(힌트: 정해진 범위의 포트에 대해 소켓을 생성하고 connect() 함수를 호출하여 성공 여부를 확인한다).

이와 같은 프로그램을 포트 스캐너(port scanner)라고 부른다. 이 문제는 가장 간단한 형태의 포트 스캐닝 구현을 요구한다. 강력한 기능의 포트 스캐너 구현에 관심이 있다면 공개 소프트웨어인 nmap (<http://www.insecure.org/nmap/>) 소스 코드를 다운로드해서 연구해보기 바란다.