WinSock Programming

10.1 WinSock

10.1.1 Introduction

The Windows Sockets specification describes a common interface for networked Windows programs. WinSock uses TCP/IP communications and provides for binary and source code compatibility for different network types.

The Windows Sockets API (WinSock API, or WSA) is a library of functions that implement the socket interface by the Berkley Software Distribution of UNIX. WinSock augments the Berkley socket implementation by adding Windows-specific extensions to support the message-driven nature of Windows system.

The basic implementation normally involves:

- Opening a socket. This allows for multiple connections with multiple hosts. Each socket has a unique identifier. It normally involves defining the protocol suite, the socket type and the protocol name. The API call used for this is `socket()`.
- Naming a socket. This involves assigning location and identity attributes to a socket. The API call used for this is `bind()`.
- Associate with another socket. This involves either listening for a connection or actively seeking a connection. The API calls used in this are `listen()`, `connect()`and `accept()`.
- Send and receive between socket. The API calls used in this are `send()`, `sendto()`, `recv()` and `recvfrom()`.
- Close the socket. The API calls used in this are `close()` and `shutdown()`.

10.1.2 Windows Sockets

The main WinSock API calls are:

`socket()`.	Creates a socket.
`accept()`.	Accepts a connection on a socket.
`connect()`.	Establishes a connection to a peer.
`bind()`.	Associates a local address with a socket.
`listen()`.	Establishes a socket to listen for incoming connection.
`send()`.	Sends data on a connected socket.
`sendto()`.	Sends data on an unconnected socket.
`recv()`.	Receives data from a connected socket.
`recvfrom()`.	Receives data from an unconnected socket.

`shutdown()`. Disables send or receive operations on a socket.
`closesocket()`. Closes a socket.

Figure 10.10 shows the operation of a connection of a client to a server. The server is defined as the computer which waits for a connection, the client is the computer which initially makes contact with the server.

On the server the computer initially creates a socket with the `socket()` function, and this is bound to a name with the `bind()` function. After this the server listens for a connection with the `listen()` function. When the client calls the `connection()` function the server then accepts the connection with `accept()`. After this the server and client can send and receive data with the `send()` or `recv()` functions. When the data transfer is complete the `closesocket()` is used to close the socket.

socket()

The `socket()` function creates a socket. Its syntax is:

`SOCKET socket ( int` *af*`, int` *type*`, int` *protocol*`)`

where

af A value of `PF_INET` specifies the ARPA Internet address format specification (others include `AF_IPX` for SPX/IPX and `AF_APPLETALK` for AppleTalk).

type Socket specification, which is typically either `SOCK_STREAM` or `SOCK_DGRAM`. The `SOCK_STREAM` uses TCP and provides a sequenced, reliable, two-way, connection-based stream. `SOCK_DGRAM` uses UDP and provides for connectionless datagrams. This type of connection is not recommended. A third type is `SOCK_RAW`, for types other than UDP or TCP, such as for ICMP.

protocol Defines the protocol to be used with the socket. If it is zero then the caller does not wish to specify a protocol.

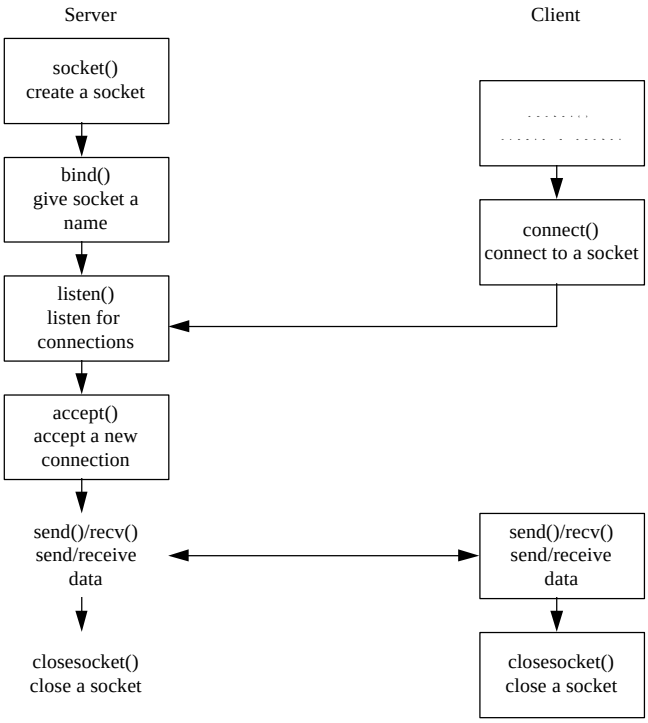

Figure 10.1 WinSock connection

If the `socket` function succeeds then the return value is a descriptor referencing the new socket. Otherwise, it returns `SOCKET_ERROR`, and the specific error code can be tested with `WSAGetLastError`. An example creation of a socket is given next:

```
SOCKET s;

  s=socket(PF_INET,SOCK_STREAM,0);
  if (s == INVALID_SOCKET)
  {
       cout << ”Socket error”
  }
```

bind()

The `bind()` function associates a local address with a socket. It is before calls to the `connect` or `listen` functions. When a socket is created with `socket`, it exists in a name space (address family), but it has no name assigned. The `bind` function gives the socket a local association (host address/port number). Its syntax is:

`int bind(SOCKET` *`s`*`, const struct sockaddr FAR *` *`addr`*`, int` *`namelen`*`);`
where

s A descriptor identifying an unbound socket.

namelen The length of the *addr*.

addr The address to assign to the socket. The `sockaddr` structure is defined as follows:

```
struct sockaddr
{
        u_short    sa_family;
        char       sa_data[14];
};
```

In the Internet address family, the `sockadd_in` structure is used by Windows Sockets to specify a local or remote endpoint address to which to connect a socket. This is the form of the `sockaddr` structure specific to the Internet address family and can be cast to `sockaddr`. This structure can be filled with the `sockaddr_in` structure which has the following form:

```
struct SOCKADDR_IN
{
    short               sin_family;
    unsigned short      sin_port;
    struct              in_addr  sin_addr;
    char                sin_zero[8];
}
```

where

`sin_family` must be set to `AF_INET`.

`sin_port` IP port.

`sin_addr` IP address.

`sin_zero` Padding to make structure the same size as `sockaddr`.

If an application does not care what address is assigned to it, it may specify an Internet address equal to `INADDR_ANY`, a port equal to 0, or both. An Internet address equal to `INADDR_ANY` causes any appropriate network interface be used. A port value of 0 causes the Windows Sockets implementation to assign a unique port to the application with a value between 1024 and 5000.

If no error occur then it returns a zero value. Otherwise, it returns `INVALID_SOCKET`, and the specific error code can be tested with `WSAGetLastError`.

If an application needs to bind to an arbitrary port outside of the range 1024 to 5000 then the following outline code can be used:

```
#include <windows.h>
#include <winsock.h>

int main(void)
{

SOCKADDR_IN    sin;
SOCKET         s;
  s = socket(AF_INET,SOCK_STREAM,0);

  if (s == INVALID_SOCKET)
  {
```

```
      // Socket failed
  }

  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = 0;

  sin.sin_port = htons(100); // port=100

  if (bind(s, (LPSOCKADDR)&sin, sizeof (sin)) == 0)
  {
       // Bind failed
  }
  return(0);
}
```

The Windows Sockets `htons` function converts an unsigned short (`u_short`) from host byte order to network byte order.

connect()

The `connect()` function establishes a connection with a peer. If the specified socket is unbound then unique values are assigned to the local association by the system and the socket is marked as bound. Its syntax is:

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

where

s Descriptor identifying an unconnected socket.
name Name of the peer to which the socket is to be connected.
namelen Name length.

If no error occur then it returns a zero value. Otherwise, it returns `SOCKET_ERROR`, and the specific error code can be tested with `WSAGetLastError`.

listen()

The `listen()` function establishes a socket which listens for an incoming connection. The sequence to create and accept a socket is:

- `socket()`. Creates a socket.
- `listen()`. This creates a queue for incoming connections and is typically used by a server that can have more than one connection at a time.
- `accept()`. These connections are then accepted with accept.

The syntax of `listen()` is:

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

where

s Describes a bound, unconnected socket.
backlog Defines the queue size for the maximum number of pending connections may grow (typically a maximum of 5).

If no error occur then it returns a zero value. Otherwise, it returns `SOCKET_ERROR`, and the specific error code can be tested with `WSAGetLastError`.

```
#include <windows.h>
#include <winsock.h>

int main(void)
{

SOCKADDR_IN    sin;
SOCKET         s;

  s = socket(AF_INET,SOCK_STREAM,0);
  if (s == INVALID_SOCKET)
  {
      // Socket failed
  }

  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = 0;

  sin.sin_port = htons(100); // port=100

  if (bind(s, (struct sockaddr FAR *)&sin, sizeof (sin)) ==
SOCKET_ERROR)
  {
        // Bind failed
     }

  if (listen(s,4)==SOCKET_ERROR)
  {
      // Listen failed
  }
  return(0);
}
```

accept()

The `accept()` function accepts a connection on a socket. It extracts any pending connections from the queue and creates a new socket with the same properties as the specified socket. Finally, it returns a handle to the new socket. Its syntax is:

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

where

s Descriptor identifying a socket that is in listen mode.
addr Pointer to a buffer that receives the address of the connecting entity, as known to the communications layer.
addrlen Pointer to an integer which contains the length of the address *addr*.

If no error occur then it returns a zero value. Otherwise, it returns `INVALID_SOCKET`,

and the specific error code can be tested with `WSAGetLastError`.

```
#include <windows.h>
#include <winsock.h>

int main(void)
{

SOCKADDR_IN    sin;
SOCKET         s;
int            sin_len;

  s = socket(AF_INET,SOCK_STREAM,0);
  if (s == INVALID_SOCKET)
  {
      // Socket failed
  }

  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = 0;
  sin.sin_port = htons(100); // port=100

  if (bind(s, (struct sockaddr FAR *)&sin, sizeof (sin)) ==
SOCKET_ERROR)
  {
       // Bind failed
  }

  if (listen(s,4)<0)
  {
      // Listen failed
  }
  sin_len = sizeof(sin);
  s=accept(s,(struct sockaddr FAR *) & sin,(int FAR *) &sin_len);
  if (s==INVALID_SOCKET)
  {
      // Accept failed
  }
  return(0);
}
```

send()

The `send()` function sends data to a connected socket. Its syntax is:

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

where

s Connected socket descriptor.
buf Transmission data buffer.
len Buffer length.
flags Calling flag.

The *flags* parameter influences the behavior of the function. These can be:

`MSG_DONTROUTE`
Specifies that the data should not be subject to routing.
`MSG_OOB` Send out-of-band data.

If `send()` succeeds then the return value is the number of characters set (which can be less than the number indicated by *len*). Otherwise, it returns `SOCKET_ERRO`, and the specific error code can be tested with `WSAGetLastError`.

```
#include <windows.h>
#include <winsock.h>
#include <string.h>
#define  STRLENGTH 100

int main(void)
{

SOCKADDR_IN    sin;
SOCKET         s;
int sin_len;
char  sendbuf[STRLENGTH];

  s = socket(AF_INET,SOCK_STREAM,0);
  if (s == INVALID_SOCKET)
  {
      // Socket failed
  }
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = 0;
      sin.sin_port = htons(100); // port=100
  if (bind(s, (struct sockaddr FAR *)&sin, sizeof (sin)) ==
SOCKET_ERROR)
  {
       // Bind failed
  }

  if (listen(s,4)<0)
  {
      // Listen failed
  }
  sin_len = sizeof(sin);

  s=accept(s,(struct sockaddr FAR *) & sin,(int FAR *) &sin_len);

  if (s<0)
  {
      // Accept failed
  }

  while (1)
  {
      // get message to send and put into sendbuff
      send(s,sendbuf,strlen(sendbuf),80);
  }
  return(0);
}
```

recv()

The `recv()` function receives data from a socket. It waits until data arrives and its syntax is:

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

where

s Connected socket descriptor.
buf Incoming data buffer.
len Buffer length.
flags Specifies the method by which the data is received.

If `recv()` succeeds then the return value is the number of bytes received (a zero identifies that the connection has been closed). Otherwise, it returns `SOCKET_ERRO`, and the specific error code can be tested with `WSAGetLastError`.

The flags parameter may have one of the following values:

`MSG_PEEK` Peek at the incoming data. Any received data is copied into the buffer, but not removed from the input queue.
`MSG_OOB` Process out-of-band data.

```
#include <windows.h>
#include <winsock.h>

#define  STRLENGTH 100

int main(void)
{

SOCKADDR_IN    sin;
SOCKET         s;
int            sin_len,status;
char           recmsg[STRLENGTH];

  s = socket(AF_INET,SOCK_STREAM,0);

  if (s == INVALID_SOCKET)
  {
      // Socket failed
  }

  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = 0;

       sin.sin_port = htons(100); // port=100

  if (bind(s, (struct sockaddr FAR *)&sin, sizeof (sin)) ==
SOCKET_ERROR)
  {
       // Bind failed
  }

  if (listen(s,4)<0)
  {
      // Listen failed
  }
  sin_len = sizeof(sin);

  s=accept(s,(struct sockaddr FAR *) & sin,(int FAR *) &sin_len);

  if (s<0)
  {
      // Accept failed
  }
```

```
  while (1)
  {
       status=recv(s,recmsg,STRLENGTH,80);

       if (status==SOCKET_ERROR)
       {
          // no socket
          break;
       }

       recmsg[status]=NULL; // terminate string
       if (status)
       {
          // szMsg contains received string
       }
       else
       {
          break;
          // connection broken
       }
  }
  return(0);
}
```

shutdown()

The `shutdown()` function disables send or receive operations on a socket and does not close any opened sockets. Its syntax is:

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

where

s Socket descriptor.
how Flag that identifies operation types that will no longer be allowed. These are:
0 – Disallows subsequent receives.
1 – Disallows subsequent sends.
2 – Disables send and receive.

If no error occur then it returns a zero value. Otherwise, it returns `INVALID_SOCKET`, and the specific error code can be tested with `WSAGetLastError`.

closesocket()

The `closesocket()` function closes a socket. Its syntax is:

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

where

s Socket descriptor.

If no error occur then it returns a zero value. Otherwise, it returns `INVALID_SOCKET`,

and the specific error code can be tested with `WSAGetLastError`.

10.2 TCP/IP services reference

Port	*Service*	*Comment*	*Port*	*Service*	*Comment*
1	TCPmux		7	echo	
9	discard	Null	11	systat	Users
13	daytime		15	netstat	
17	qotd	Quote	18	msp	Message send protocol
19	chargen	ttytst source	21	ftp	
23	telnet		25	smtp	Mail
37	time	Timserver	39	rlp	Resource location
42	nameserver	IEN 116	43	whois	Nicname
53	domain	DNS	57	mtp	Deprecated
67	bootps	BOOTP server	67	bootps	
68	bootpc	BOOTP client	69	tftp	
70	gopher	Internet Gopher	77	rje	Netrjs
79	finger		80	www	WWW HTTP
87	link	Ttylink	88	kerberos	Kerberos v5
95	supdup		101	hostnames	
102	iso-tsap	ISODE	105	csnet-ns	CSO name server
107	rtelnet	Remote Telnet	109	pop2	POP version 2
110	pop3	POP version 3	111	sunrpc	
113	auth	Rap ID	115	sftp	
117	uucp-path		119	nntp	USENET
123	ntp	Network Timel	137	netbios-ns	NETBIOS Name Service
138	netbios-dgm	NETBIOS	139	netbios-ssn	NETBIOS session
143	imap2		161	snmp	SNMP
162	snmp-trap	SNMP trap	163	cmip-man	ISO management over IP
164	cmip-agent		177	xdmcp	X Display Manager
178	nextstep	NeXTStep	179	bgp	BGP
191	prospero		194	irc	Internet Relay Chat
199	smux	SNMP Multiplexer	201	at-rtmp	AppleTalk routing
202	at-nbp	AppleTalk name binding	204	at-echo	AppleTalk echo
206	at-zis	AppleTalk zone information	210	z3950	NISO Z39.50 database
213	ipx	IPX	220	imap3	Interactive Mail Access
372	ulistserv	UNIX Listserv	512	exec	Comsat 513 login
513	who	Whod	514	shell	No passwords used
514	syslog		515	printer	Line printer spooler
517	talk		518	ntalk	
520	route	RIP	525	timed	Timeserver
526	tempo	Newdate	530	courier	Rpc
531	conference	Chat	532	netnews	Readnews
533	netwall	Emergency broadcasts	540	uucp	Uucp daemon
543	klogin	Kerberized ‘rlogin’ (v5)	544	kshell	Kerberized ‘rsh’ (v5)