sockets

sockets - basics

Although the socket concept covers a variety of cases, like Unix sockets (end-point(s) in local interprocess communication) or end-point of a bi-directional communication link in the Berkeley sockets, we will limit ourselves, within the scope of this course, to the most used variety, namely internet sockets. An (internet) **socket** is a logical entity which describes **the end point(s) of a communication link** between two IP entities (entities which implement the Internet Protocol). Sockets are **identified by the IP address and the port number.**

IP addresses come in two flavors, as defined in by the two versions of the Internet Protocol currently in use, version 4 and version 6.

Version 4 IP addresses are 32 bits long and are represented as a sequence of 4 8-bit integers, dot separated, ranging from 0 to 255, like 193.231.200.67.

Version 6 IP addresses are 128 bits long and are represented by a sequence of eight 16-bit integers, separated by columns (:). An example - 2008:0CB8:85A4:0000:0000:8F2C:0371:7714.

port numbers

Port numbers range from 0 to 65535 (2^16 – 1) and are split into 3 categories:

1. **well known ports** - ranging from 0 to 1023 – these ports are under the control of IANA (Internet Assigned Number Authority), a selective list is shown in the table below:

Port number	UDP protocol	TCP protocol	Other
1		TCPMUX	
5	Remote Job Entry (RJE)		
7	Echo		
15	NETSTAT		
20		FTP - data	
21		FTP – control	
22	Secure Shell		
23	Telnet		
25	Simple Mail Transfer Protocol (SMTP)		
41	Graphics		
42	ARPA Host Name Server Protocol		WINS
43		WHOIS	
53	Domain Name System (DNS)		
57		Mail Transfer Protocol (MTP)	
67	BOOTP		
68	BOOTP		
69	TFTP		

79		Finger	
80		HTTP	
107		Remote Telnet	
109		Post Office Protocol 2 (POP2)	
110		POP3	
115		Simple FTP (SFTP)	
118	SQL services		
123	Network Time Protocol (NTP)		
137	NetBIOS Name Service		
138	NetBIOS Datagram Service		
139	NetBIOS Session Service		
143	Internet Message Access Protocol (IMAP)		
156	SQL service		
161	Simple Network Management Protocol (SNMP)		
162	SNMP Trap		
179		Border Gateway Protocol (BGP)	
194		Internet Relay Chat (IRC)	
213	IPX		

2. **registered ports** - ranging from 1024 to 49151 – registered by ICANN, as a convenience to the community, should be accessible to ordinary users. A list of some of these ports is shown below:

Port number	UDP protocol	TCP protocol	Other
1080		SOCKS proxy	
1085	WebObjects		
1098	RMI activation		
1099	RMI registry		
1414		IBM WebSphere MQ	
1521		Oracle DB default listener	
2030	Oracle services for Microsoft Transaction Server		
2049	Network File System		
2082		CPanel default	
3306	MySQL DB system		
3690	Subversion version control system		
3724	World of Warcraft online gaming		
4664		Google Desktop Search	
5050		Yahoo Messenger	
5190		ICQ and AOL IM	
5432	PostgreSQL DB system		

5500		VNC remote desktop protocol	
5800		VNC over HTTP	
6000/6001		X11	
6881-6887	BitTorrent		
6891-6900	Windows Live Messenger – File transfer		
6901	Windows Live Messenger – Voice		
8080		Apache Tomcat	
8086/8087	Kaspersky AV Control Center		
8501	Duke Nukem 3D		
9043		WebSphere Application Server	
14567	Battlefield 1942		
24444	NetBeans IDE		
27010/27015	Half-Life, Counter-Strike		
28910	Nintendo Wi-Fi Connection		
33434	traceroute		

3. **dynamic (private) ports**, ranging from 49152 to 65535

posix sockets

The **socket API**s (Application Programming Interface) are rooted into the POSIX socket API. POSIX stands for Portable Operating System Interface – a common name for a set of IEEE standards used to define APIs. This family of standards dates back to 1988 and is identified as IEE 1003 or ISO/IEC 9945.

The socket communication is, in general, asymmetric. One of the two communicating entities plays the role of a **server**. The server listens for incoming requests at a certain port. This port number is public, and together with the IP address identifies the server. The actual communication is initiated by the **client,** who sends a connection request to the server. If the connection request Is accepted, the server creates (in general) another socket, which is dedicated to the communication with that particular client. The closure of this communication link can be initiated by either the client or by the server.

To create a client socket, two calls are necessary. The first one creates a file descriptor (fd) which is basically a number which identifies an I/O channel (not different from the file descriptor resulted from a fopen() call which opens a file).

The prototype of this call is the following:

```
int socket(int family, int type, int protocol);
```

The **family** parameter specifies the address family of the socket and may take one of the following values, the list itself depending on the implementation platform:

- `AF_APPLETALK`
- `AF_INET` – most used, indicates an IP version 4 address

- `AF_INET6` – indicates an IP version 6 address
- `AF_IPX`
- `AF_KEY`
- `AF_LOCAL`
- `AF_NETBIOS`
- `AF_ROUTE`
- `AF_TELEPHONY`
- `AF_UNSPEC`

The **type** parameter specifies the socket stream type and may take the following values:

- `SOCK_DGRM` – the transport level protocol is UDP
- `SOCK_STREAM` – the transport level protocol is TCP
- `SOCK_RAW` – used to generate/receive packets of a type that the kernel doesn't explicitly support

The value of the **protocol** parameter is set to 0, except for raw sockets.

The function `socket()` returns an integer. In case of success, it is the identifier of a file descriptor (fd), and if the call fails, it is an error code.

The second call connects the client to the server. Here is the signature of the `connect()` call.

```
int connect(int sock_fd, struct sockaddr * server_addr, int addr_len);
```

The `sockaddr` structure varies depending on the protocol selected. For reference purposes, let's display it, together with another associated structure – `sockaddr_in`, both used in the context of IPv4.

```
struct sockaddr {
        ushort  sa_family;
        char    sa_data[14];
};
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Both structures have the same size and have compatible content, therefore casting is allowed.

The `sock_fd` parameter identifies the local socket and the `server_addr` parameter contains information about the server we try to connect to. `addr_len` is just the size (in bytes) of this structure.

The `connect()` call returns 0, in case of success or a negative error indicator, in case of failure.

To create a server socket, four calls are necessary. Here are the prototypes of these calls:

```
int socket(int family, int type, int protocol);
int bind(int sock_fd, struct sockaddr * my_addr, int addr_len);
int listen(int sock_fd, int backlog);
int accept(int sock_fd, struct sockaddr * client_addr, int * addr_len);
```

The `bind()` function merely associates the socket to a specified port, while the `listen()` function sets the server in listening mode.

A few remarks. Why not binding the client socket to a particular port, as well? Well, nobody stops us from invoking the bind() function on a client socket, but this is not exactly relevant. While the server port has to be known, because the client must know both the IP address (or the URL, if that is the case) and the port of the server, it is not important to know the port of the client. The assignment of a port to a client socket is done by the operating system, and this solution is quite satisfactory.

The `accept()` call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. In case of an error, the return value is an error code. The first parameter is the fd of the local listening socket, and the second one is a reference pointer to the address of the client at the other end of the connection. The third parameter is a pointer to the size of this structure.

the select() function

How do we detect if some sort of activity occurs on a file descriptor (fd) of interest?. There are two specialized functions, namely poll() and select(), which are triggered when some I/O activity occurs at one of the fd's of interest.

We present now the function select() while the poll function is presented in the next paragraph.

Here is the signature of the select function, according to [SELE]:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

The function allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation.

The function returns the number of fd's which are “ready” or a negative number, in case of error.

The nfds is set equal to the highest fd + 1, *readfds*, *writefds* and *excepfds* are bit masks specifying that a bit which is set denotes an fd of interest. A few macros – FD_ZERO, FD_SET, FD_CLR and FD_ISSET are used to manage these bit masks.

The *timeout* value is specified in milliseconds and its expiration leads to the trigger of select, in case no I/O activity was detected during the time specified.

the poll() function

The poll() function is an alternative to the select() function. One of the weaknesses of the select()

function is the way the in which it stores the fd's of interest. - as a bit mask and therefore has a fixed size which sometimes is pretty big.

The signature of the poll() function is presented below:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Using the function requires the include of <poll.h>.

The struct pollfd is described below:

```
struct pollfd {
  int fd;           /* file descriptor */
  short events;     /* requested events */
  short revents;    /* returned events */
};
```

From a functional stand point, the poll() function waits for one of the file descriptors of interest to become ready to perform some I/O operation.

The caller specifies the number of items in the array *fds* in the *nfds* parameter.

The *timeout* parameter specifies the number of milliseconds that the poll() function should be waiting for some fd to become ready.

The main differences between poll() and select() are explained well by Richard Stevens [DIFSP]:

“The basic difference is that select()'s fd_set is a bit mask and therefore has some fixed size. It would be possible for the kernel to not limit this size when the kernel is compiled, allowing the application to define FD_SETSIZE to whatever it wants (as the comments in the system header imply today) but it takes more work. 4.4BSD's kernel and the Solaris library function both have this limit. But I see that BSD/OS 2.1 has now been coded to avoid this limit, so it's doable, just a small matter of programming. :-) Someone should file a Solaris bug report on this, and see if it ever gets fixed.

With poll(), however, the user must allocate an array of pollfd structures, and pass the number of entries in this array, so there's no fundamental limit. As Casper notes, fewer systems have poll() than select, so the latter is more portable. Also, with original implementations (SVR3) you could not set the descriptor to -1 to tell the kernel to ignore an entry in the pollfd structure, which made it hard to remove entries from the array; SVR4 gets around this. Personally, I always use select() and rarely poll(), because I port my code to BSD environments too. Someone could write an implementation of poll() that uses select(), for these environments, but I've never seen one. Both select() and poll() are being standardized by POSIX 1003.1g.”

the server program

In our example, written in C, the server socket is identified by an integer, called listen_fd, which serves as an identifier for the server socket.

One note: on Windows platforms, the function WSAStartup() has to be called before calling the socket() function. How this can be done, is explained at the link below [WSAS]:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms742213(v=vs.85).aspx

The general structure (main socket related functions) of a server socket program, using the particular case of an IPv4, TCP server, is as follows, if we keep only the socket related functions:

```
// initialization part
WSAStartup(...);     // if working on a Windows platforms
listen_fd = socket(AF_INET, SOCK_STREAM, 0);
err = bind(listen_fd, (sockaddr *)&serv_addr, sizeof(serv_addr));
listen(listen_fd, 16);
// service loop
```

```
while(1) {
   num_ready = select(maxfd + 1, &rset, NULL, NULL, NULL);
   conn_fd = accept(listen_fd, (sockaddr *)&cli_addr, &cli_len);
   // read/write operations ...
}
closesocket(listen_fd);
```

It is worth noting that the accept() call returns the fd of a newly created socket which is dedicated to the communication with that particular client

the client program

In our client example, written in C, the client socket is identified by an integer, called client_fd, which serves as an identifier for the client socket.

Again, on Windows platforms, the function WSAStartup() has to be called before calling the socket() function. How this can be done, is explained at the link below:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms742213(v=vs.85).aspx

The general structure (main socket related functions) of a client socket program, using the particular case of an IPv4, TCP client, is as follows, if we keep only the socket related functions:

```
// initialization part
WSAStartup(...);     // if working on a Windows platforms
client_fd = socket(AF_INET, SOCK_STREAM, 0);
 err = connect(client_fd, (struct sockaddr *)&address, len);
// read/write operations ...
closesocket(client_fd);
```

IN our particular program, the server information (namely the IP address and port number) are specified as command line arguments. Moreover, there is no need to specify a particular port for the client socket, because this is assigned automatically by the operating system.

socket libraries and APIs

High level languages allow a more consistent and coherent use of socket related functionality.

Library support for socket functions is provided by different vendors and contributors, for different languages and platforms. Below are listed some of them.

C/C++ libraries:

- Boost.Asio
- C++ Network Library – a collection of open source libraries for high level network programming
- Qt
- ZeroMQ
- nanomsg
- Apache APR
- Winsock2 (for Windows only)
- C++ Rest SDK

- Glib networking
- libcurl

Java networking support is provided through the classes included in the java.net package. Below we list some of its main classes:

- Socket, with SSLSocket as direct subclass
- ServerSocket

Other classes of interest are used for input and out through sockets, like:

- DataInputStream
- DataOutputStream

Other languages, not necessarily object oriented (at least at their beginnings), like **Php**, provide an extended API for sockets support. In the Php case, we mention functions like:

- socket_accept()
- socket_bind()
- socket_close()
- socket_connect()
- socket_create()
- socket_listen()
- socket_read(), socket_recv()
- socket_select()
- socket_send(), socket_write()

bibliography

[DIFSP] – Differences between pol() and select() - https://stackoverflow.com/questions/970979/what-are-the-differences-between-poll-and-select

[POLL] – the poll() function - http://man7.org/linux/man-pages/man2/poll.2.html

[PORT] – Port numbers - https://www.lifewire.com/popular-tcp-and-udp-port-numbers-817985

[POSO] – Posix sockets API - http://home.deib.polimi.it/agosta/lib/exe/fetch.php?id=teaching%3Apsrete&cache=cache&media=teaching:socket.pdf

[SELE] – the select() function - http://man7.org/linux/man-pages/man2/select.2.html

[WSAS] – the WSAStartup() function - https://msdn.microsoft.com/en-us/library/windows/desktop/ms742213(v=vs.85).aspx