Socket Programming

1. Introduction

In the classic client-server model, the *client* sends out *requests* to the *server*, and the server does some processing with the request(s) received, and returns a *reply (or replies)* to the client. The terms *request* and *reply* here may take on different meanings depending upon the context, and method of operation. An example of a simple and ubiquitous client-server application would be that of a Web-server. A client (Internet Explorer, or Netscape) sends out a request for a particular web page, and the web-server (which may be geographically distant, often in a different continent!) receives and processes this request, and sends out a reply, which in this case, is the web page that was requested. The web page is then displayed on the browser (client).

Figure 1. Client-Server paradigm

Further, servers may be broadly classified into two types based on the way they serve requests from clients. *Iterative* servers can serve only one client at a time. If two or more clients send in their requests at the same time, one of them has to wait until the other client has received service. On the other hand, *concurrent* servers can serve multiple clients at the same time. Typically, this is done by spawning off a new server process on the receipt of a request – the original process goes back to listening to new connections, and the newly spawned off process serves the request received.

We can realize the client-server communication described above with a set of network protocols, like the TCP/IP protocol suite, for instance. The lower-level details, like the operation of the protocol itself must be clear to you by now. In this tutorial, we will look at the issue of **developing applications** for realizing such communication over a network. In order to write such applications, we need to understand sockets.

2. What are sockets?

Sockets (also called Berkeley Sockets, owing to their origin) can simply be defined as end-points for communication. To provide a rather crude visualization, we could imagine the client and server hosts in Figure 1 being connected by a pipe through which data-flow takes place, and each end of the pipe can now be construed as an "end-point". Thus, a socket provides us with an abstraction – or a *logical* end point for communication. There are different types of sockets. Stream sockets, of type `SOCK_STREAM` are used for connection oriented, TCP connections, whereas datagram sockets of type `SOCK_DGRAM` are used for UDP based applications. Apart from these two, there are other socket types defined, like `SOCK_RAW` and `SOCK_SEQPACKET`.

2.1 Socket layer, and the Berkeley Socket API

Figure 2 shows the TCP/IP protocol stack, and shows where the "Socket layer" may be placed. Again, please be advised that this is just a representation to indicate the level at which we operate when we write network programs using sockets. As shown in the figure, sockets make use of the lower level network protocols, and provide the application developer with an interface to the lower level network protocols. A library of system calls are provided by the socket layer, and are termed as the "Socket API". These system calls can be used in writing socket programs. In the sections that ensue, we will study those system calls in detail.

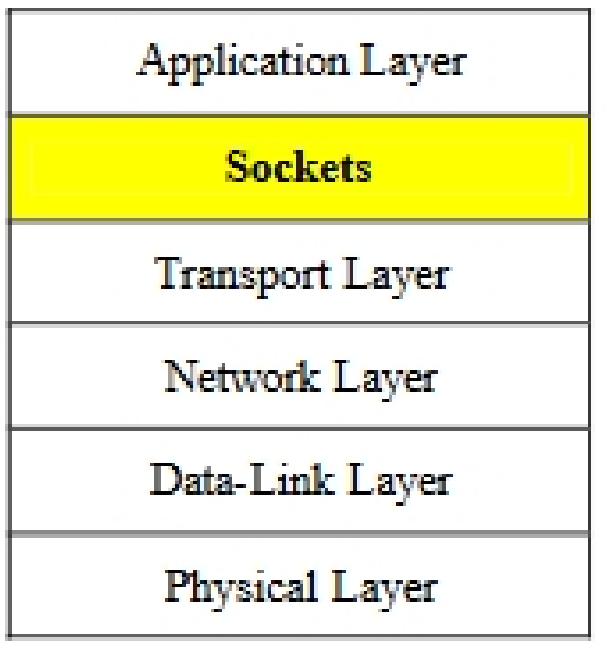

Figure 2. TCP/IP protocol stack

2.2 Programming environment, and the WinSock library

We will study socket programming on the windows environment (Windows 9x/NT/2000/XP). Socket programming on UNIX based environments is similar, and most of the system calls used are the same. Windows adds its own windows-specific extensions. Windows provides the WinSock library which contains the system calls used for socket programming on the windows environment. Compilation issues on windows machines is relegated to Appendix-I.

3 Basic Socket system calls

Figure 3 shows the sequence of system calls between a client and server for a connection-oriented protocol.

3.1 The `socket()` system call

The `socket()` system call creates a socket and returns a handle to the socket created. The handle is of data type `SOCKET`. In UNIX based environments, the socket descriptor is a non-negative integer, but Windows uses the `SOCKET` data type for the same.

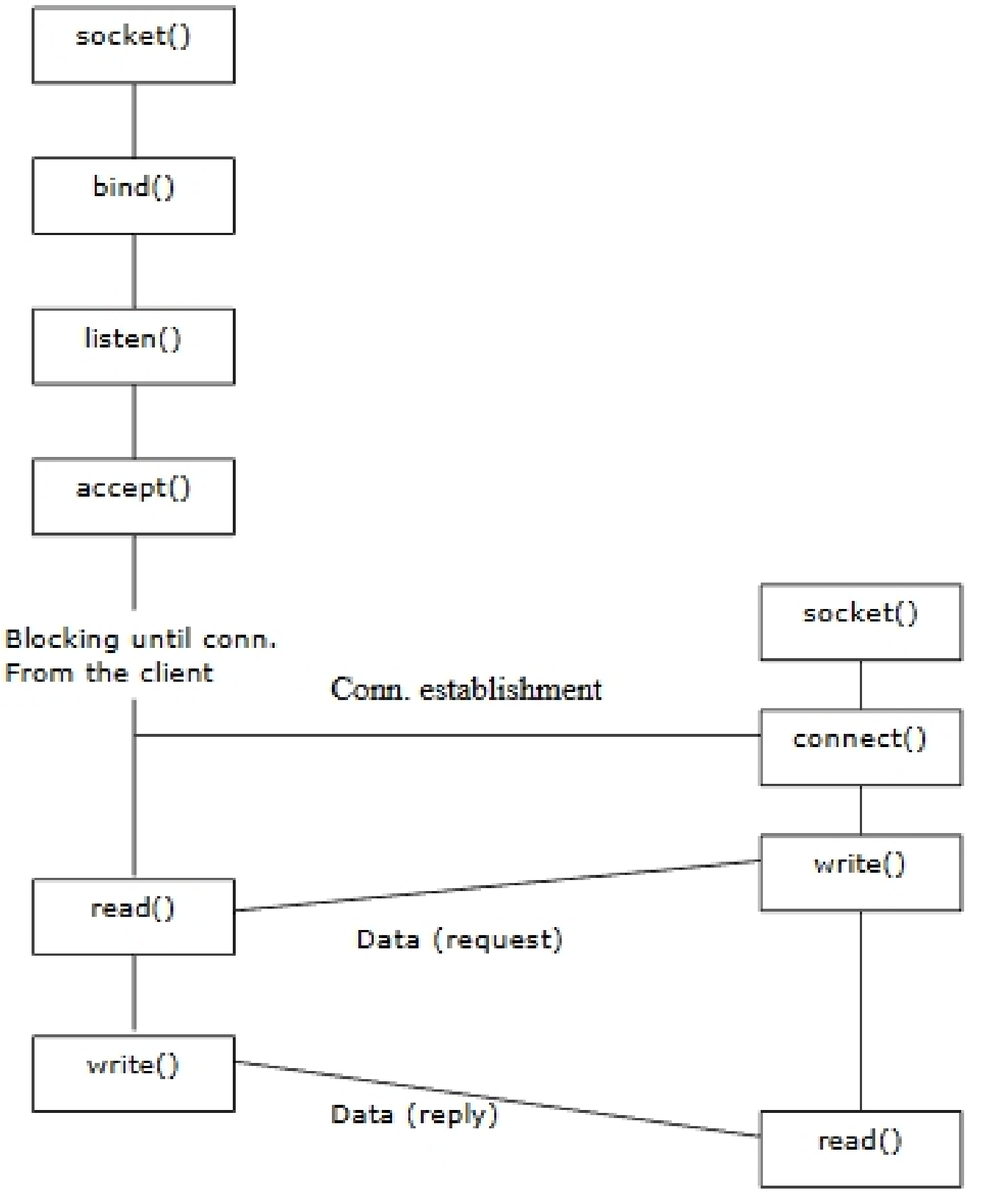

Figure 3. Socket system calls for connection-oriented case

Socket handles may take any value in the range 0 to INVALID_SOCKET-1.

socket() system call syntax