Programming with TCP/IP – Best Practices

Matt Muggeridge
TCP/IP for OpenVMS Engineering

> "Be liberal in what you accept, and
> conservative in what you send"
>
> Source: RFC 1122, section 1.2.2 [Braden, 1989a]

Overview

A seasoned network programmer appreciates the many complexities and pitfalls associated with meeting the requirements of robustness, scalability, performance, portability, and simplicity for an application that may be deployed in a heterogeneous environment and a wide range of network configurations. The TCP/IP programmer controls only the end-points of the network connection, but must provide for all contingencies, both predictable and unpredictable. Therefore, an extensive knowledge base is required. The TCP/IP programmer must understand the relationship among network API calls, protocol exchange, performance, system and network configuration, and security.

This article is intended to help the intermediate TCP/IP programmer who has a basic knowledge of network APIs in the design and implementation of a TCP/IP application in an OpenVMS environment. Special attention is given to writing programs that support configurations where multiple NICs are in use on a single host, known as a multihomed configuration, and to using contemporary APIs that support both IPv4 and IPv6 in a protocol-independent manner. Key differences between UDP and TCP applications are identified and code examples are provided.

This article is not the most definitive source of information for TCP/IP programmers. There are many more topics that could be covered, as is evident by the number of expansive text books, web-sites, newsgroups, RFCs, and so on.

The information in this article is organized according to the structure of a network program. First, the general program structure is introduced. Subsequent sections describe each of the phases: Establish Local Context, Connection Establishment, Data Transfer, and Connection Shutdown. The final section is dedicated to important general topics.

Structure of a Network Program

All network programs are structured in a similar way, regardless of the complexity of the service they provide. They consist of two peer applications: one is designated as the server and the other is the client (see Figure 1). Each application creates a local end-point (*socket*), and associates (*binds*) a local name with it that is identified by the three-tuple: *protocol, local IP address,* and *port number*. The named end-point can be referenced by a peer application to form a connection that is uniquely identified in terms of the named end-points. Once connected, data transfer occurs. Finally, the connection is shut down.

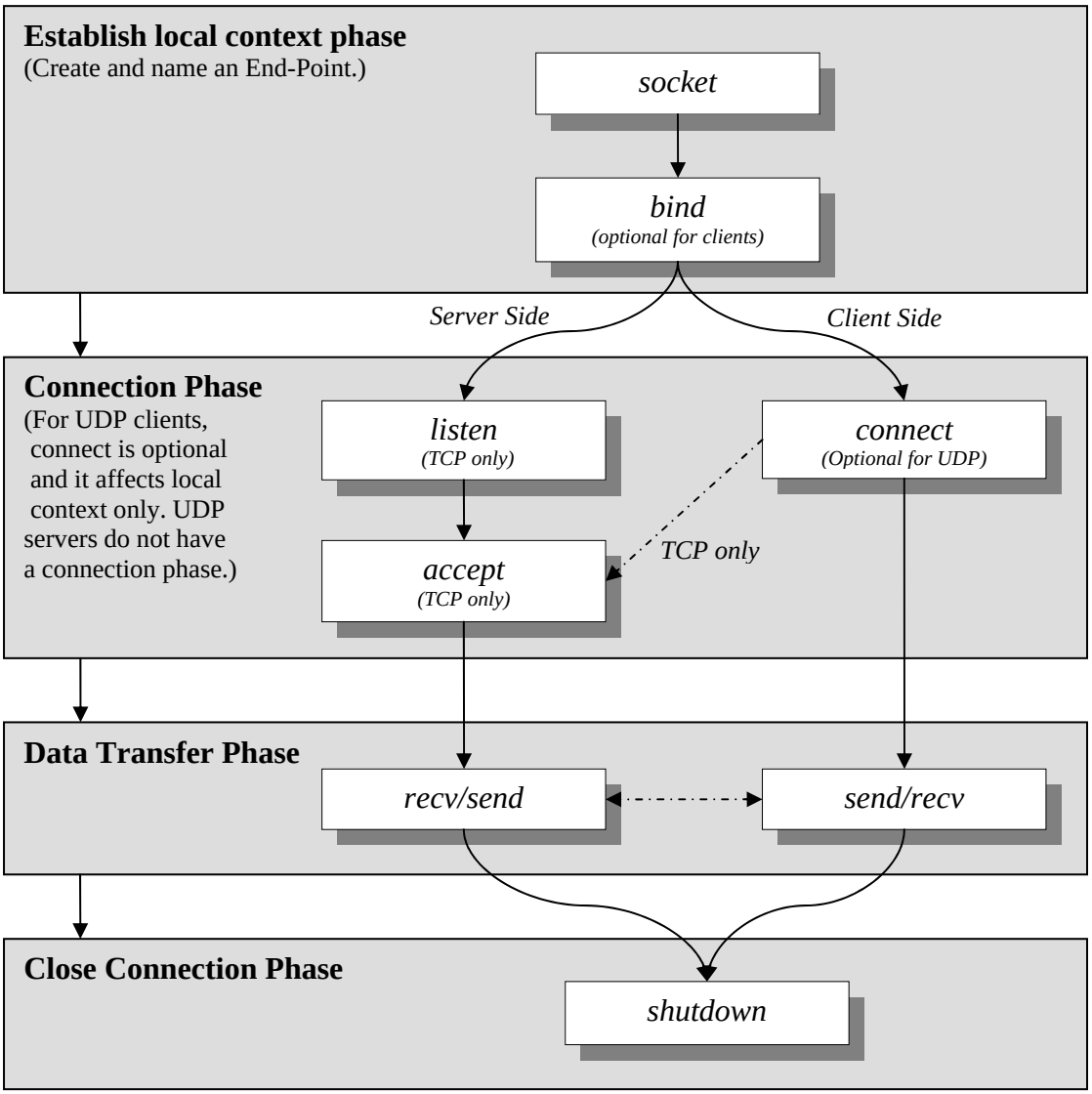

Figure 1 Structure of a Network Program

You have to take special measures to support multihomed configurations for UDP applications. In addition, by using the modern API's, network programs can readily support both IPv4 and IPv6. Writing applications that are largely independent of the version of the internet protocol (IPv4 or IPv6) requires the use of simple address conversion APIs.

The structure of any network program is independent of the API. Here it is described in terms of the BSD API. Most TCP/IP applications use the BSD sockets API, which was introduced with BSD V4.2 in 1983. OpenVMS programmers also have the option of using the $QIO system services, which may be preferable, especially when designing an event-driven application.

The remainder of this document is divided into sections that match the structure of a network program, as shown in Figure 1.

Establishing Local Context Phase

Establishing local context begins with deciding on the most appropriate transport protocol: TCP, UDP, or both. In addition, requirements for establishing local context differ based on whether the application is the server or the client. For instance, a server should be able to accept incoming connections directed to any IPv4 or IPv6 address configured on the system. A server design (TCP or UDP) may require multiple threads to accept incoming connections on the same port and address for either or both TCP and UDP. A TCP server must avoid the TCP TIME_WAIT state so that it can be restarted instantly. The client (UDP or TCP) must be aware that a server may be identified by a list of addresses and should connect to the more preferred address. These points are discussed in more detail in the following subsections and are summarized in the list below:

- Select the Appropriate Protocol – UDP or TCP
- Providing for UDP Behaviors
- Creating an End-Point
- Naming an Endpoint
- Servers Explicitly Bind to a Local End-Point
- Clients Implicitly Bind to Their End-Point
- UDP Servers Enable Ancillary Data
- Servers Reuse Port and Address
- UDP Servers Enable Ancillary Data
- Management of Local Context Phase

Select the Appropriate Protocol – UDP or TCP

One of the earliest decisions a TCP/IP programmer must make is whether the application will use a *datagram* or *stream* socket type. This decision determines whether the transport protocol will be UDP or TCP, because UDP uses the datagram socket type and TCP uses the stream socket type.

The four socket types are compared in Table 1. Only the *datagram* and *stream* socket types are discussed in this article. The *raw* socket provides access to underlying communication protocols and is not intended for general use. The *sequenced* socket is not implemented by TCP/IP Services for OpenVMS. The SCTP protocol (RFC 2960) uses sequenced sockets.

Table 1. Socket Types and Characteristics

	RAW	*DATAGRAM*	*STREAM*	*SEQUENCED*
Bidirectional	✓	✓	✓	✓
Reliable			✓	✓
Sequenced			✓	✓
No Duplicates			✓	✓
Record Boundaries		✓		✓

The User Datagram Protocol (UDP) is connectionless, and supports broadcasting and multicasting of datagrams. UDP uses the *datagram* socket service, which is not reliable; therefore, datagrams may be lost. Also, datagrams may be delivered out of sequence or duplicated. However, record boundaries are preserved; a `recvfrom()` call will result in the same unit of data that was sent using the corresponding `sendto()`.

In a UDP application, it is the responsibility of the programmer to ensure reliability, sequencing, and detection of duplicate datagrams. The UDP broadcast and multicast services are not well suited to a WAN environment, because routers will often block broadcast and multicast traffic. Also because WANs are generally less reliable, a UDP application in a WAN environment may suffer from the greater processing overhead required to cope with data loss, which may in turn flood the WAN with retransmissions. UDP is particularly suited to applications that rely on short request-reply communications in a LAN environment, such as DNS (the Domain Name System) or applications that use a polling mechanism such as the OpenVMS Load Broker and Metric Server.

The Transmission Control Protocol (TCP) is connection-oriented; provides reliable, sequenced service; and transfers data as a stream of bytes. Because TCP is connection-oriented, it has the additional overhead associated with connection setup and tear-down. For applications that transfer large amounts of data, the cost of connection overhead is negligible. However, for short-lived connections that transfer small amounts of data, the connection overhead can be considerable and can lead to performance bottlenecks. Examples of TCP applications that are long-lived or transfer large amounts of data include Telnet and FTP.

Providing for UDP Behaviors

UDP is designed to be an inherently unreliable and simple protocol. Do not expect errors to be returned when datagrams are lost, arrive out of sequence, dropped, or duplicated. You may find it necessary to overcome these behaviors. It is the responsibility of the UDP application to detect these conditions, and it must take the appropriate action according to the application's needs. At some point, you may be duplicating the behavior of the TCP protocol in the application, in which case you should reconsider your choice of protocol.

Creating an End-Point

In a BSD-based system, the socket defines an end-point. It is a local entity and is used to establish local context only. The end-point is created using the BSD `socket()` function. See Example 1.

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

Where the arguments are:
      domain – may be either AF_INET (IP version 4) or AF_INET6 (IP version 6)
      type – field may be either SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
      protocol – set to zero, because the protocol is implied by the type argument
```

Example 1 Creating an End-Point

Naming an Endpoint

An end-point is uniquely identified by its name. The name is defined by the *protocol, local IP address,* and *local port number* using the `bind()` function, as shown in Example 2.

The server-side application must `bind()` a name to its socket so that clients can reference the service. It is not recommended for the client-side application to call `bind()`. When a client does

not explicitly call `bind()`, the kernel will implicitly bind a name of its choosing when the application calls either `connect()` for TCP, or `sendto()` for UDP.

```
int bind(int socket, struct sockaddr *address, int address_len);

Where the arguments are:
      socket – value returned by calling the socket() function
      address – socket address structure
      address_len – length of socket address structure

Note that the "address" structure and "address_len" value are best initialized by
calling getaddrinfo().
```

Example 2 Naming an Endpoint

Servers Explicitly Bind to a Local End-Point

A service is identified by its *protocol, local IP address,* and *local port number.* The application advertises its service as either TCP or UDP on a specific local port number. (When a service binds to a local port number below 1024, the process requires one of the following privileges: SYSPRV, BYPASS, or OPER.) A server application should be capable of accepting connections on all IP addresses configured on the host, including IPv4 and IPv6 addresses.

Binding to all addresses is easiest to achieve by binding to the special address known as INADDR_ANY (IPv4) or IN6ADDR_ANY_INIT (IPv6). However, by using the protocol-independent APIs, the differences between IPv6 and IPv4 become less relevant. The server can readily be programmed to accept incoming TCP connections (or UDP datagrams) sent to any interface configured with an IPv4 or IPv6 address.

Use `getaddrinfo()` to return the list of all available socket addresses, (see Example 3). This function accepts hostnames as alias names, or in numeric format as IPv4 or IPv6 strings. In a multihomed environment, this may be a long list. For instance, a system configured with IPv4 and IPv6 addresses will return a socket address for each of the following protocol combinations: TCP/IPv6, UDP/IPv6, TCP/IP and UDP/IP.
Example 4 demonstrates the method for establishing local context for each of the socket addresses configured on a system, independent of IPv6 or IPv4.

Clients Implicitly Bind to Their End-Point

Whereas a server must explicitly `bind()` to its local end-point so that its service may be accessible, a client does not advertise a service. Hence, a client is able to use any local IP address and local port number. This is achieved by the client skipping the `bind()` call, (see the client path in Figure 1). Instead, when a TCP client issues `connect()`, or a UDP client issues `sendto()`, an implicit binding is made. The bound IP address is determined from the routing table and the order of addresses configured on an interface. The local port number is dynamically assigned and is referred to as an *ephemeral* port.

Note that the ephemeral port numbers are selected from a range specified by the following `sysconfig inet` attributes [Hewlett-Packard Company, 2003c]:

`ipport_userreserved` (specifies the maximum ephemeral port number)
`ipport_userreserved_min` (specifies the minimum ephemeral port number)

These values can be modified with the following command:

```
$ sysconfig –r inet ipport_userreserved=65535 ipport_userreserved_min=50000
```

Servers Reuse Port and Address
You should be aware of the following limitations with respect to servers binding to their local port and local IP address.

By default, the server's local port number and local address can be bound only once. Subsequent attempts to bind another instance of the server to the same local port and local address will fail, even if it is using a different protocol. This is a problem for servers that must advertise UDP and TCP services on the same port.

If a server performs an *active close*, the TCP state machine [Stevens, 1994] forces it into the TIME_WAIT state which, amongst other things, prevents an application from binding to the same local IP address and local port number. An active close is performed by the peer application that first issues the `shutdown()`, which causes TCP to send a FIN packet (see Figure 3). The peer that receives the FIN packet performs a *passive close* and is not subject to the TIME_WAIT state. The TIME_WAIT state lasts for at least twice the maximum segment lifetime (`sysconfig inet` attribute `tcp_msl` [Hewlett-Packard Company, 2003c].). By default this is 60 seconds. For the duration of the TIME_WAIT state, a subsequent attempt to `bind()` to the same local address

```
int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);

Where the arguments are:
      nodename – string representing nodename as an alias or numeric format
      servname – string representing service name or port number
      hints – filters addresses with matching fields in addrinfo structure
      res – linked list of returned addresses
```

Example 3 Obtaining Addresses with Protocol Independent API

and local port number will return an error (EADDRINUSE). The service would be unavailable for at least 60 seconds. You can prevent the server from going through the TIME_WAIT state by having the client perform the active close, which causes no problems because the client will obtain a new ephemeral port for each invocation. Despite having a well-designed client, a server will still perform an active close if it exits unexpectedly or is forced to issue the active close for some other reason, and you must avoid this situation.

```
int sd[MAX_SOCKS]; /* one per TCP/IPv6, UDP/IPv6, TCP/IP, UDP/IP */
char *port, *addr = NULL;
struct addrinfo *res, hints;

port = argv[1]; /* port number as a string – must not be NULL */
if(argc == 3) addr = argv[2]; /* hostname – NULL implies ANY address */

memset(&hints, '\0', sizeof(hints));
hints.ai_flags = AI_PASSIVE; /* if usrreq.addr NULL, sets sockaddr to ANY */

err = getaddrinfo(usrreq.addr, usrreq.port, &hints, &res);
if(err) {
   if(err == EAI_SYSTEM) perror("getaddrinfo");
   else printf("getaddrinfo error %d - %s", err, gai_strerror(err));
   return 1;
}

i = 0;
for(aip = res; aip; aip = aip->ai_next) {
   if(aip->ai_family != AF_INET && aip->ai_family != AF_INET6) continue;

   /* create a socket for this protocol */
   sd[i] = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
   if(sd[i] < 0) {perror("socket"); return sd[i];}

   err = socket_options(sd[i], aip); /* set SO_REUSEADDR, SO_REUSPORT etc. */
   if(err == -1) {perror("socket_options"); return 1;}

   err = bind(sd[i], res->ai_addr, res->ai_addrlen);
   if(err == -1) {perror("bind"); return 1;}

  /** perform other per-socket work here – e.g. maybe create threads etc **/

   if(i == NUM_ELT(sd)) {printf("Insufficient socket elements\n"); break;}
   i++;
}
freeaddrinfo(res);
```

Example 4 Server Establishes Context for All Addresses

To overcome these issues, you must modify the server's socket to allow it to rebind to the same address and port number multiple times and without delay. This is implemented as a call to `setsockopt()`, as shown in Example 5.

```
int on = 1;

/* allow server to reuse address when binding */
err = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
if(err < 0) {perror("setsockopt SO_REUSEADDR"); return err;}

/* allows UDP and TCP to reuse port (and address) when binding */
err = setsockopt(sd, SOL_SOCKET, SO_REUSEPORT, (char *)&on, sizeof(on));
if(err < 0) {perror("setsockopt SO_REUSEPORT"); return err;}
```

Example 5 Setting Socket Options to Reuse Port and Address

UDP Servers Enable Ancillary Data

In a multihomed environment, a UDP server requires special care when replying to a request. It may reply to a client using any appropriate interface, setting the outgoing source address to that interface. That is, the reply source address does not have to match the request's destination address.

This creates problems in environments protected by a firewall that monitors source and destination addresses. If a packet that has a reply source address that does not match the request's destination address, the firewall interprets this as address spoofing and drops the packet. Also, a client using a connected UDP socket will only receive a datagram with a source/destination address pair matching what it specified in the `connect()` call. Therefore, the server socket must be enabled to receive the destination source address and the server must reply using that address as the reply source address. Sample code that enables a socket to receive the destination address information is shown in Example 6. This is an area where there are differences between IPv6 and IPv4, so they must be treated individually.

```
int err, on = 1, len = sizeof(on);

/* UDP should reply using dst address of the request */
if(ai->ai_protocol == IPPROTO_UDP) {
   if(ai->ai_family == AF_INET) { /* must be IPv4 - enable recvdstaddr */
      err = setsockopt(sd, IPPROTO_IP, IP_RECVDSTADDR, (char *)&on, len);
      if(err < 0) {perror("setsockopt IP_RECVDSTADDR"); return err;}
   }
   else { /* must be IPv6 - enable recvpktinfo and pktinfo */
      err = setsockopt(sd, IPPROTO_IPV6,IPV6_RECVPKTINFO, (char *)&on, len));
      if(err < 0) {perror("setsockopt IP_RECVPKTINFO"); return err;}

      err = setsockopt(sd, IPPROTO_IPV6,IPV6_PKTINFO,(char *)&on, len);
      if(err < 0) {perror("setsockopt IP_PKTINFO"); return err;}
   }
}
```

Example 6 UDP Servers Enable Ancillary Data

Management of Local Context Phase
The number of sockets that can be opened is limited by the OpenVMS `CHANNELCNT sysgen` parameter, with one channel per socket. In addition, starting with TCP/IP Services for OpenVMS V5.4, a new `sysconfig net` subsystem attribute `ovms_unit_maximum` can extend the limit. The ephemeral port range is defined by the `sysconfig inet` attributes `ipport_userreserved and ipport_userreserved_min.` The `sysconfig` attributes are discussed in more detail in [Hewlett-Packard Company, 2003c].

Connection Phase

The connection phase has a different meaning depending on whether TCP or UDP is being used. In the case of TCP, the establishment of a connection results in a protocol exchange between the peers and, if successful, each peer maintains state information about that connection. Similarly, when a TCP connection is shut down, it results in a protocol exchange that affects a change in state of each peer. (See [Stevens, 1994] for more information about the TCP state machine.) Before establishing a connection, a TCP server application must issue `listen()` and `accept()` calls. The TCP client application initiates the connection by calling `connect()`.

UDP, on the other hand, is a connectionless protocol and the connection phase is optional. However, a UDP socket may be connected, which serves only to establish additional local context about the peer's address. That is, a UDP connect request does not result in any protocol exchange between peers. When a UDP socket is connected, the application will receive notifications generated by incoming ICMP messages and it will receive datagrams only from the peer that it has connected to. In other words, if a UDP socket is *not* connected, it is unable to receive notifications from ICMP packets and it will receive datagrams from any address. In fact, it is common for the UDP client to connect the server address, while the UDP server never connects the client address. A UDP `connect()` is similar to binding, where `bind()` associates a *local address* with the socket; `connect()` associates the *peer address* with the socket.

Because connecting TCP sockets is different from connecting UDP sockets, they are treated separately in the following subsections:

- TCP Connection Phase
- Optional UDP Connection Phase
- `connect()` and Address Lists
- Resolve Host Name Prior to Every Connection Attempt
- Server Controls Idle Connection Duration with Keepalive
- TCP Server's Listen Backlog
- Management of Connection Phase

TCP Connection Phase
TCP connection establishment defines a protocol exchange, often referred to as the "three-way handshake," between client and server. The API calls and resulting protocol exchange are shown in Figure 2.

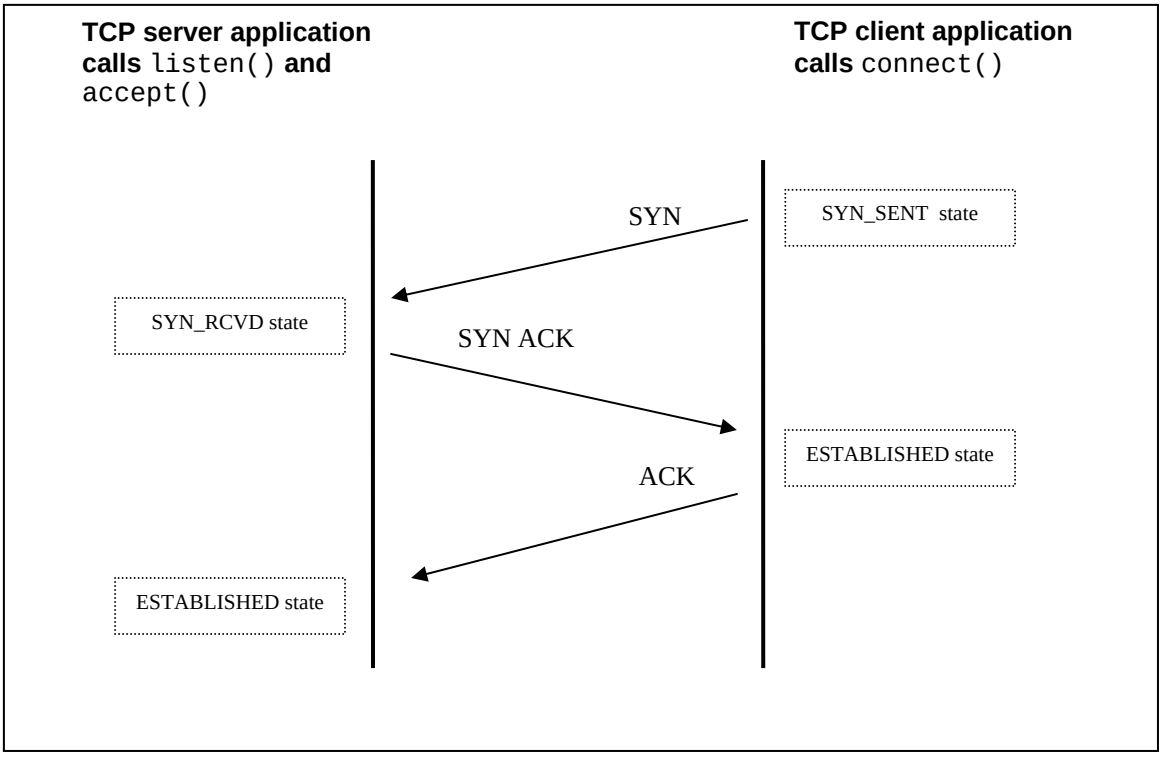

Figure 2 TCP Connection Establishment - "Three-Way Handshake"

Before a server can receive a connection, it must first issue `listen()` and `accept()`. These are illustrated in Example 7.

```
int listen(int socket, int backlog);

Where the arguments are:
      socket – value returned by calling the socket() function
      backlog – maximum number of outstanding connection requests
```

```
int accept(int socket, struct sockaddr *fromaddr, int *fromlen);

Where the arguments are:
      socket – value returned by calling the socket() function
      fromaddr – socket address of the peer we're accepting the connection from
      fromlen – length of the socket address

The return value is a new socket descriptor that can be used for data transfer between
the client and server.

Note that if the peers address details need to be recorded, they are best decoded with
getnameinfo().
```

Example 7 TCP Server Connection Phase

Once the server is ready to accept incoming connections the TCP client initiates the connection by calling `connect()`:

```
int connect(int socket, struct sockaddr *toaddr, int *tolen);

Where the arguments are:
      socket – value returned by calling the socket() function
      toaddr – socket address describing the peer to connect to
      tolen – length of the socket address

Note that if the toaddr and tolen fields may be initialized by calling getaddrinfo().
```

Example 8 TCP or UDP Client Connect Phase

Optional UDP Connection Phase

Unlike the TCP client where the `connect()` call is required, a UDP client may optionally call `connect()`. The API for connecting a UDP socket is the same as that used to connect a TCP socket, as shown in Example 8. During a UDP `connect()` the destination address is bound to the socket; therefore, when sending a UDP datagram, it is an error to specify the destination address with each datagram. To transmit data over the connected UDP socket, use the `sendto()`function with a NULL destination address, or use the `send()` function.
A connected UDP socket does not change the behavior of UDP as a connectionless protocol. There is no protocol exchange when a UDP socket is connected or shut down, and there is no

state machine. Connecting a UDP socket affects the local context only. A connected UDP socket allows the kernel to deliver errors to the user application as the result of received ICMP messages. Unconnected UDP sockets do not receive errors as a result of an ICMP message. For example, when a connected UDP socket attempts to send a datagram to a host without a bound service, the ICMP "port unreachable" message is returned and the kernel reports this to the application as a "connection refused" error. The client application is alerted that the service is not running.

As a result of connecting a UDP socket, the programmer must not specify the destination address with each datagram, because it is already bound to the socket. Unlike `bind()`, which binds a local name (IP address and port) to a socket, the UDP `connect()`call binds the remote name (IP address and port) to the UDP socket.

Because a UDP server must accept incoming datagrams from many remote clients and a connected UDP socket limits the communication to one peer at a time, the UDP server should not use connected sockets. Furthermore, a multihomed UDP server may reply with a source address that differs from the client's destination address. If the client is using a connected UDP socket, then datagrams that do not match the address in the connected UDP socket will not be delivered to the client. See Example 20 for programming a UDP server in a multihomed environment.

`connect()` and Address Lists

Host names are often stored in a name server as DNS aliases. It is not unusual for a DNS alias to be represented by multiple IP addresses, usually for the purpose of offering higher availability or load sharing [Muggeridge, 2003]. When an application resolves a host name, the resolver will return the list of addresses associated with that host name. The address list is usually sorted with the most desirable address at the top of the list.

The client should call `getaddrinfo()` (see Example 9) to retrieve the list of IP addresses and then cycle through this list, attempting to connect to each address until a successful connection is established. (RFC 1123 Sec 2.3. [Braden, 1989b])

Resolve Host Name Prior to Every Connection Attempt

Addresses are not permanent – they can change or become unavailable. For example:

- A system administrator may add or remove addresses during a migration exercise.
- High availability configurations using the Load Broker/Metric Server are designed to dynamically update the DNS alias with a modified address list [Muggeridge, 2003].

Therefore, it is highly recommended that applications never cache IP addresses. When the client connects or reconnects, it should resolve the server's DNS alias each time, using `getaddrinfo()`. This makes the client resilient to changes in the servers' address list. (See Example 9).

Keep in mind that DNS may also be caching bad addresses. Even if your application performs the name-to-address conversion again, it may receive the same obsolete list. Some strategies for ensuring the DNS alias lists are current include making an address highly available with failSAFE IP, or dynamically keeping the DNS alias address list current using Load Broker/Metric Server. For more information, on this refer to [Muggeridge, 2003].

Server Controls Idle Connection Duration with Keepalive

Because a server assigns resources for every connection, it should also control when to release the resources if the connection remains idle. A polling mechanism used to ensure the peer is still connected can be used to keep the connection alive. TCP has a "keepalive" mechanism built into the protocol; however, UDP does not.

```
char *srv_addr, *srv_port;
struct addrinfo *srv_res, *ai, hints;

srv_addr = argv[1];  /* server address */
srv_port = argv[2];  /* sever port     */

memset(&hints, '\0', sizeof(hints));
hints.ai_family   = usrreq.family;
hints.ai_socktype = usrreq.type;

/* get remote address info */
err = getaddrinfo(srv_addr, srv_port, &hints, &srv_res);
if(err) {
   if(err == EAI_SYSTEM) perror("getaddrinfo");
   else printf("getaddrinfo error %d - %s", err, gai_strerror(err));
   return 1;
}
for(ai = srv_res; ai; ai = ai->ai_next) {
   /* AF_INET and AF_INET6 only */
   if(ai->ai_family != AF_INET && ai->ai_family != AF_INET6) continue;

   sd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
   if(sd < 0) {perror("socket"); continue;}  /* try next socket */

   err = socket_options(sd, ai);
   if(err) {perror("sockopt"); continue;}  /* try next socket */

   /* use connected UDP sockets if userreq.connected is set */
   if(ai->ai_protocol == IPPROTO_TCP || usrreq.connected) {
       err = connect(sd, ai->ai_addr, ai->ai_addrlen);
       if(err == -1) {perror("connect"); continue;} /* try next socket */
   }
   break; /* use first successful connection */
}
if(err == -1) {printf("No connection"); return 1;}

/** data transfer phase **/
```

Example 9 Client Connects to Each Server Address Until Success

TCP is a wonderfully robust protocol that can recover from lengthy network outages, but this can result in zombie connections on the server. A zombie connection is one that is maintained by just one of the peers after the other peer has exited. For example, a cable modem may be powered off before the client application shuts down the connection. Because the modem has been powered off, the TCP client cannot notify the server that it is shutting the connection. As a result, the server-side application unwittingly maintains the context of the connection. This is not unusual with home networks. Without any notification of the client being disconnected, the TCP server will maintain its connection indefinitely.

There are a number of ways to solve this problem. At the application level, a keepalive message can be transferred between peers. When a peer stops responding for a configurable number of keepalives the connection should be closed. Alternatively, the system manager can enable a system-wide keepalive mechanism that will affect all TCP connections. This is controlled using

the following `sysconfig inet` atrributes: `tcp_keepidle`, `tcp_keepcnt`, and `tcp_keepintvl` [Hewlett-Packard Company, 2003c].

These system configuration parameters are useful when an application does not provide a mechanism for closing zombie connections. An application must be restarted to pick up changes in the keepalive `sysconfig` attributes.

Because UDP provides no way to determine the availability of its peer, you can implement a keepalive mechanism at the application level for this purpose.

TCP Server's Listen Backlog

The TCP server's listen backlog is a queue of connection requests for connections that have not been accepted by the application. When the connection has been accepted by the application, the request is removed from the backlog queue. The length of the backlog is set by the `listen()` call. If this backlog queue becomes full, new connection requests are silently ignored, which may lead to clients suffering from timeouts on their new connection attempts.

When a TCP application issues a successful `connect()` request, it results in a "three-way handshake" (shown in Figure 2). When the `connect()` request is unsuccessful, you have to provide for all potential failures.

Note from Figure 2 that the client and server enter the ESTABLISHED state at different times. Therefore, if the final ACK is lost, it is possible for the client to believe it has established a connection, while the server remains in SYN_RCVD state. The server must receive the final ACK before it believes the connection is established.

Consider the impact that this protocol exchange has on the peer applications. For example, after the first SYN is received, the TCP server tracks the connection request by adding the peer details to an internal socket queue *(so_q0)*. When the final ACK is received, the peer details are moved from *so_q0* to another internal queue *(so_q)*. The connection state is freed from *so_q* only when the application's `accept()` call completes. (For more details see [Wright, 1995].)

The socket queues will grow under any of the following conditions:

- The rate of incoming SYN packets (connection requests) is greater than the completion rate of `accept()`.
- The final ACK is slow in arriving, or an acknowledgement (SYN ACK or ACK) is lost.
- The final ACK never arrives (as in the case of a SYN flood attack).

If the condition persists, the socket queues will eventually become full. Subsequent SYN packets will be silently dropped (that is, the TCP server does not respond with SYN ACK segments). Eventually, the client-side application will time out. The client timeout will occur after approximately `tcp_keepinit/2` seconds (75 seconds by default).

The length of this socket queue is controlled by several attributes. You can specify the queue length in the `listen()` call. This can be overridden with the `sysconfig` attributes `sominconn` and `somaxconn`, but the server application must be restarted to use these system configuration changes. Restarting a busy server may not be practical, so it is better to treat this as a sizing concern, and ensure that the `accept()` call is able to complete in a timely fashion, given the rate of requests and the length of the listen queue.

Management of Connection Phase
The state of a connection can be viewed using the commands:

- `netstat`
- `tcpip show device`

These commands are described in [Hewlett Packard, 2003b].

The following `sysconfig` attributes of the `socket` subsystem affect the connection phase: `sominconn, somaxconn, sobacklog_drops, sobacklog_hiwat, somaxconn_drops, tcp_keepalive_default, tcp_keepcnt, tcp_keepidle, tcp_keepinit, tcp_keepintvl` [Hewlett-Packard, 2003c].

Data Transfer Phase

The data transfer phase provides the means of transferring data between the peer applications and, as with the connection establishment phase, there are different choices to be made depending on whether TCP or UDP is being used. The topics covered in this section include:

- Data Transfer APIs
- Avoid MSG_PEEK When Receiving Data
- Avoid Out-Of-Band Data
- TCP Data Transfer – Stream of Bytes
- UDP Data Transfer – Datagram Units
- UDP Datagram Size
- Understand Buffering
- Management of Data Transfer Phase

Data Transfer APIs
Several functions can be used for data transfer. The choice of function depends on whether the socket is connected or unconnected. A TCP socket must be connected, whereas a UDP socket may be connected or unconnected (see Example 8). A connected socket may transmit data with `send()` or `write()` and receive data with `recv()` or `read()`. The `send()` and `recv()` functions support a "flags" argument that `write()` and `read()` do not. Unconnected sockets require functions that support a "destination address" in the API. These functions include `sendto()` or `sendmsg()`, and `recvfrom()` or `recvmsg()`. The list of examples that demonstrate these API calls are:

- Example 10 Connected Socket Data Transfer APIs
- Example 11 Unconnected UDP Sockets Data Transfer APIs
- Example 12 UDP Data Transfer
- Example 13 TCP Receive Algorithm
- Example 14 UDP Receive Algorithm
- Example 15 Retrieving and Modifying Socket Options

For connected sockets the `recv()` and `send()` functions are shown in Example 10.

For unconnected sockets, the destination address of the peer must be sent with each message. Similarly, when receiving each message, the peer's address is available to the application. The `sendto()` and `recvfrom()` functions support the peer's address, as show in Example 11.

Note that for `sendto()`, the peer's destination address is specified by the arguments `dstaddr` and `dstlen`, which should be initialized using `getaddrinfo()`. For `recvfrom()`, the peer's address is available in the `fromaddr` argument and can be resolved with `getnameinfo()`.

```
int recv(int socket, char *buffer, int length, int flags);

int send(int socket, char *message, int length, int flags);

Where the arguments are:
      socket – value returned by calling the accept() function
      message – buffer containing data to be sent
      length – length of the message to send
      flags – sender may control transmission of the message.
```

Example 10 Connected Socket Data Transfer APIs

```
int sendto(int socket, char *message, int length, int flags,
           struct sockaddr *dstaddr, int dstlen);

int recvfrom(int socket, char *message, int length, int flags, struct sockaddr
*fromaddr, int *fromlen);

Where the arguments are:
      socket – socket descriptor for data transfer
      message – buffer containing data to be sent
      length – length of the message to send
      flags – sender may control transmission of the message
      dstaddr – socket address of destination
      dstlen – length of dstaddr
      fromaddr – socket address of peer from where the data was sent
      fromlen – length of fromaddr
```

Example 11 Unconnected UDP Sockets Data Transfer APIs

For UDP sockets, (regardless of whether they are connected or unconnected) the `sendmsg()` and `recvmsg()` routines may be used. These routines provide a special interface for sending and receiving *ancillary data*[1], as shown in Example 12. A socket may be enabled to receive ancillary data with `setsockopt()`. A special use of the received ancillary data is shown in Example 17 through Example 21.

Note that the `sendmsg()` and `recvmsg()` functions are particularly important in a UDP server application in a mulithomed configuration; see section "UDP and Multihoming" on page 21.

[1] UDP over IPv4 ignores ancillary data for `sendmsg()`.

```
int sendmsg(int socket, const struct msghdr *message, int flags);

int recvmsg(int socket, struct msghdr *message, int flags);

Where the arguments are:
      socket – value returned by calling the accept() function
      message – describes the data to be sent and ancillary data
      flags – sender may control transmission of the message.
```

Example 12 UDP Data Transfer

The `msghdr` structure contains a field, `msg_control`, for ancillary data. IPv4 currently ignores this field for transmission. IPv6 uses the ancillary data as described in RFC 3542 [Stevens et. al., 2003].

The `recv()`, `recvfrom()`, and `recvmsg()` APIs support a `flags` argument that can be used to control message reception. One of the options is MSG_PEEK, which allows the application to peek at the incoming message without removing it from the socket's receive buffer. Another option is MSG_OOB, which supports the processing of out-of-band data. These features are not recommended, as explained in the next two sections.

Avoid MSG_PEEK When Receiving Data

With today's modern networks and high-performing large memory systems, MSG_PEEK is an unnecessary receive option. The MSG_PEEK function looks into the socket receive buffer, but does not remove any data from it. Keep in mind that the objective of any network application is to keep data flowing through the network. Because the MSG_PEEK function does not remove data from the receive buffer, it will cause the receive window to start closing, which applies back pressure on the sender and can result in an inefficient use of the network. In any case, after a MSG_PEEK, it is still necessary to read the data from the socket. So an application may as well have done that in the first place and "peeked" inside its own buffer.

Avoid Out-Of-Band Data

Out-of-band (OOB) data provides a mechanism for urgently delivering a single byte of data to the peer. The receiving application is notified of OOB data and it may be read out of sequence. It is typically used for signaling. However, out-of-band data cannot be relied upon and is dependent on the implementation of the protocol stack. In many BSD implementations, if the out-of-band data is not read by the application before new out-of-band data arrives, then the new OOB data overwrites the unread OOB data. Instead of using OOB data for signaling, a better approach is to create a dedicated connection for signaling.

TCP Data Transfer – Stream of Bytes

Possibly the most common oversight of TCP/IP programmers is that they fail to realize the significance of TCP sending a stream of bytes. In essence, this means that TCP guarantees to deliver no more than one byte at a time; no matter how much data the user application sends. The amount of data that TCP transmits is affected by a wide variety of protocol events such as: send window size, slow start, congestion control, Nagle algorithm, delayed ACKS, timeout events and so on [Stevens, 1994], [Snader, 2000]. In other words, data is delivered to the peer in differently sized chunks that are independent of the amount of data that is written to TCP with each `send()` call. For a TCP application, this means that, when receiving data, the algorithm must loop on a `recv()` call. See Example 13.

```
while((nbytes = recv(sd, streambuf, sizeof(streambuf), 0)) > 0)
{
   /* assemble the TCP byte stream into a message buffer */
   if(msg_assembled(message, streambuf, nbytes)) perform_action(message);
}
if(nbytes == -1) perror("recv");
```

Example 13 TCP Receive Algorithm

UDP Data Transfer – Datagram Units

UDP delivers datagram units in the same way they were sent, preserving record boundaries. Sample code showing how a UDP application may receive data is shown in Example 14.

```
if((nbytes = recvmsg[1](sd, &mhdr, 0)) == -1) perror("recvmsg");
```

Example 14 UDP Receive Algorithm

UDP Datagram Size

For robustness and responsible memory usage, do not assume the maximum size of a datagram when allocating buffer space within the application data structures. Avoid sending datagrams much larger than the maximum transmission unit, (MTU), because lost IP segments will require retransmission of the entire datagram. Also, limit the datagram size to be less than the socket option SO_SENDBUF.

Example 15 describes the APIs used to retrieve and modify the socket options. The maximum size of a datagram can be determined by calling `getsockopt()` with the `level` and `option_name` arguments set to SOL_SOCKET and SO_SENDBUF, respectively.

```
int getsockopt(int socket, int level, int option_name,
               void *option_value, socklen_t *option_len);

int setsockopt(int socket, int level, int option_name,
               char *option_value, socklen_t option_len);

Where the arguments are:
      socket – socket handle as returned by the socket() call.
      level – protocol layer for socket option
      option_name – socket option
      option_value – address of buffer to store the size of the socket send buffer
      option_len – address of integer to store the length of data returned
```

Example 15 Retrieving and Modifying Socket Options

The maximum size of a socket buffer can be controlled by system-wide variables. These can be viewed and modified with the `sysconfig` utility. For example, to view these values you can use the following command [Hewlett-Packard Company, 2003c]:

```
$ sysconfig –q inet udp_sendspace udp_recvspace
```

To set the `udp_recvspace` buffer to 9216 bytes, use:

```
$ sysconfig –r inet udp_recvspace 9216
```

Changing these attribute settings will override programs that use the `setsockopt()` function to modify the size of their respective socket buffers.

Understand Buffering
A TCP or UDP application writes data to the kernel. The kernel stores the data in the socket send buffer. A successful write operation means that the data has been successfully written to the socket send buffer; it does not indicate that the kernel has sent it yet.

The procedure then involves two steps:

1. The data from the send buffer is transmitted and arrives at the peer's socket receive buffer. In the case of TCP, the delivery of data to the peer's socket receive buffer is guaranteed by the protocol, because TCP acknowledges that it has received the data. UDP provides no such guarantees and silently discards data if necessary.
   At this point, the data has not yet been delivered to the peer application.
2. The receiving application is notified that data is ready in its socket's receive buffer and the application reads the data from the buffer.

Because of this buffering, data can be lost if the receiving application exits (or the node crashes) while data remains in the receive socket buffer. It is up to the application to guarantee that data has arrived successfully at the peer application. TCP has completed its responsibility when it notifies the application that the data is ready.

Management of Data Transfer Phase
The following `sysconfig` attributes of the `socket` subsystem affect the data transfer phase: `tcp_sendspace, tcp_recvspace, udp_sendspace, tcp_recvspace, tcp_nodelack` [Hewlett-Packard, 2003c].

Connection Shutdown Phase

A connection is bidirectional; consequently, each side of the connection may be shut down independently. The following topics are described for the connection shutdown phase:

- TCP Orderly Release
- Management of Connection Shutdown

The `shutdown()` function is shown in Example 16.

```
int shutdown(int socket, int how);

Where the arguments are:
      socket – socket created for data transfer
      how – describes the direction to shutdown
```

Example 16 Connection Shutdown API

TCP Orderly Release

An application may not know how much data it will receive from a peer; therefore, each peer should signal when it has finished sending data, as in a telephone conversation in which both parties say "goodbye" to indicate they have nothing more to say. Similarly, a receiving application should not exit before it has received the signal indicating the last of the data. TCP applications can make use of the *half-close* to signal that a peer has finished sending data. When both peers have signaled this, the socket may be closed and the application can exit.

When a TCP application issues a `shutdown()` on the sending side of the socket, it results in the protocol exchange as shown in Figure 3. The TCP FIN packet is queued behind the last data. Because a connection is bidirectional, it requires a total of four packets to shut down both directions of a connection. The side that first issues the `shutdown()`on the sending side of the socket performs an *active close.* The side that receives the FIN performs a *passive close.* The difference between these is important, because the side issuing an *active close* must also wait in the TIME_WAIT state for 2 x MSL[2]. Some socket resources persist during the TIME_WAIT state. Because it is more critical to conserve server resources than client resources (see page 28), it is better practice to ensure the client issues the active close. See Servers Reuse Port and Address, on page 6, which discusses avoidance of the TIME_WAIT delay.

It is possible to `shutdown()` the receive side of a socket, but this is of little use, because shutting down the receive side does not result in a protocol exchange. In practice, the send direction of the socket is the more appropriate to shut down. Further attempts to send data on that socket will return an error. The peer reads the data until there is no more data in the receive socket buffer. When TCP processes the FIN packet, it closes that side of the connection and a subsequent `recv()` will return an error. This signals the receiver that the peer has no more data to send.

UDP applications do not provide for any protocol exchange when `shutdown()` is called. Instead, the programmer must design a message exchange that signals the end of transmission.

[2] MSL = Maximum Segment Lifetime

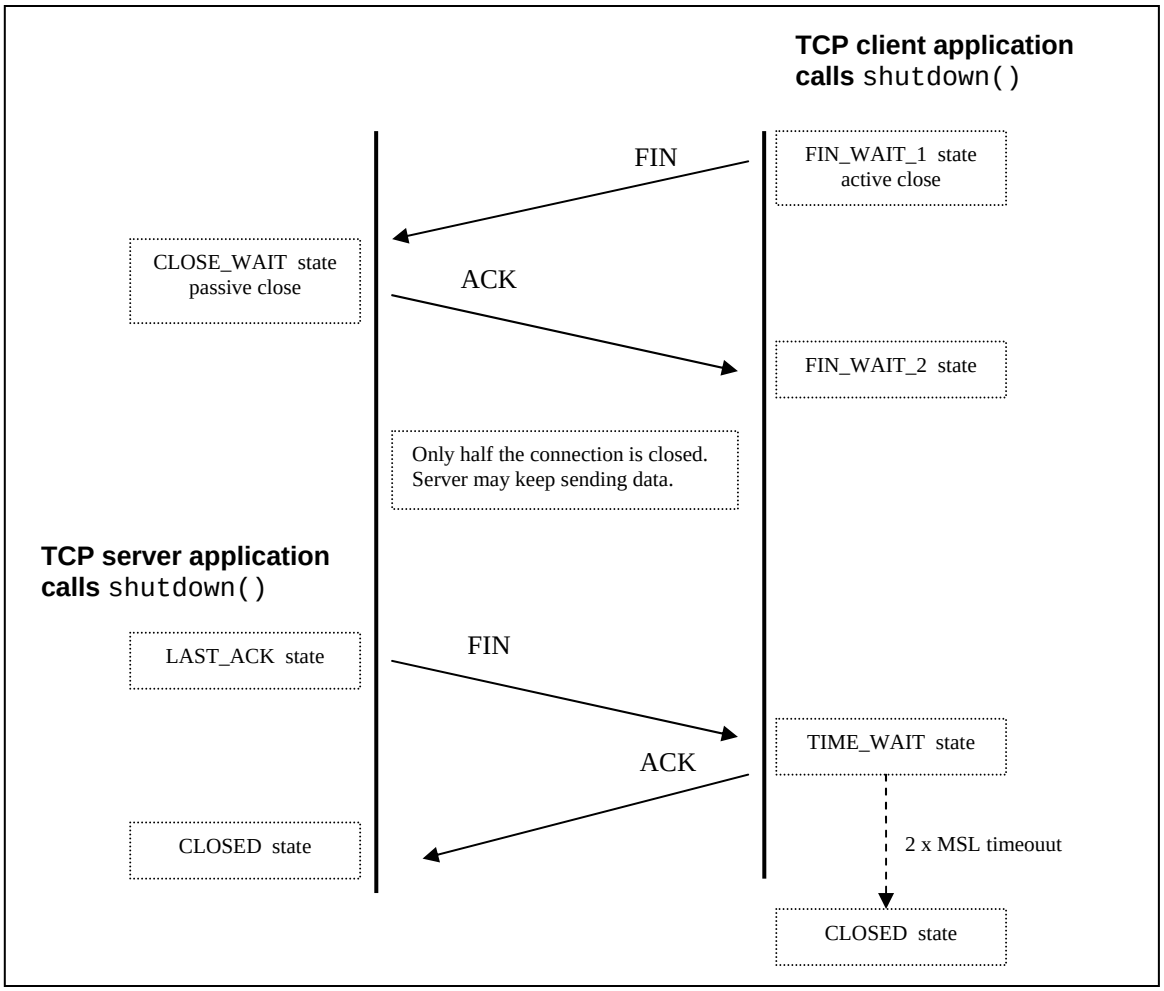

Figure 3. Closing a Connection - *4-Way Handshake*

Management of Connection Shutdown

The following `sysconfig` attributes of the `inet` subsystem affect the connection shutdown phase: `tcp_keepinit, tcp_msl`, [Hewlett-Packard, 2003c]. The network manager may also shut down a connection using the `TCPIP DISCONNECT` command, [Hewlett-Packard, 2003b].

Miscellaneous

A range of topics do not readily fit into the specific phases described in Figure 1, Structure of a Network Program. These are treated in the subsections below, including:

- UDP and Multihoming
- Concurrency In Server Applications
- Conservation of Server Resources
- Simplifying Address Conversion

UDP and Multihoming

Multihomed hosts are becoming more common. Recent features in TCP/IP Services for OpenVMS make it more desirable to configure a system with multiple interfaces. Depending on the configuration, this can have the following advantages:

- Load-balancing of outgoing connections over interfaces configured in the same subnet
- Increased data throughput
- IP addresses can be protected when using failSAFE IP
- Multiple subnets per host

With these benefits, additional concerns arise for the TCP/IP programmer:

- Name to address translation can return a list of addresses (see "connect() and Address Lists," page 12).
- UDP servers ought to set the source address in the reply to be the same as the destination address in the request (see next subsection).

Each UDP datagram is transmitted with an IP source address that is determined dynamically[3]. The algorithm for choosing the IP source address is performed in two steps:

1. The routing protocol selects the most appropriate outbound interface
2. The subnet's primary address configured on that interface is assigned as the IP source address for the outbound datagram.

In a multihomed environment with multiple interfaces configured in the same subnet, this can result in successive datagrams with different IP source addresses. This becomes even more likely in a configuration that uses failSAFE IP, where the primary IP address may change during a failover or recovery event [Muggeridge, 2003].

An extract from RFC 1122 section 4.1.3.5 [Braden, 1989a] says:

```
A request/response application that uses UDP should use
a source address for the response that is the same as
the specific destination address of the request.
```

An extract from RFC 1123, section 2.3 [Braden, 1989b] says:

```
When the local host is multihomed, a UDP-based request/response
application SHOULD send the response with an IP source address
that is the same as the specific destination address of the UDP
request datagram.
```

If this recommendation is ignored, the application will be subject to the following types of failures.

1. Firewalls may be configured to pass traffic with specific source and destination addresses. In a single-homed host this does not present a problem, because a client will send a datagram to the server's IP address and the server will reply using that IP address in its source address field. When a second interface is configured with an address in the same subnet, the IP layer can choose either address as the reply source address. To solve this problem, either the firewall needs to be reconfigured or the addresses need to be reconfigured. If the UDP server application is written to always set the reply to the RECVDSTADDR then it will be more robust to changes in the network configuration.
2. Clients that use connected UDP sockets will only receive UDP datagrams from the server if its address matches the value stored in the UDP's connected socket.

The implementation differs depending on whether it is an AF_INET (IPv4) or AF_INET6 (IPv6) socket. However, the algorithm is essentially the same:

1. Enable the socket to receive ancillary data.

[3] Unless the server binds to a specific IP address, which is not recommended as described in "Servers Explicitly Bind to a Local End-Point," page 5.

2. Receive the ancillary data that describes the destination address.
3. Reply using the source address that was received in the ancillary data.

Because IPv6 and IPv4 differ in the type of ancillary data that is needed, some general type definitions will help with making the implementation protocol-independent. These type definitions are shown in Example 17.

```
#include <sys/socket.h>
#include <net/in.h>

typedef union _recvdstaddr_u { /* ancillary data - IPv4 recvdstaddr */
   struct cmsghdr cm;
   char   ia[_CMSG_SPACE(sizeof(struct in_addr))];
} recvdstaddr_t;

typedef union _recvpktinfo_u { /* ancillary data - IPv6 recvpktinfo */
   struct cmsghdr cm;
   char   pktinfo[_CMSG_SPACE(sizeof(struct in6_pktinfo))];
} recvpktinfo_t;

typedef union _ancillary_u  { /* ancillary data - general control structure */
   recvdstaddr_t recvdstaddr; /* IPv4 */
   recvpktinfo_t recvpktinfo; /* IPv6 */
} ancillary_t;
```

Example 17 Ancillary Data Type Definitions for IPv4 and IPv6

Assuming the socket has been enabled to receive ancillary data (see Example 6) the sample library function in Example 18 can be called to receive IPv4 or IPv6 ancillary data.

```
int udp_recvdstaddr(int sd, void *buf, int buflen,
                    struct sockaddr_storage *from,
                    ancillary_t *recvdstaddr, unsigned int *controllen)
{
   struct msghdr mhdr; /* structure used to receive ancillary data */
   struct iovec iov[1]; /* data buffer to */
   int nbytes = 0, buflen = sizeof(buf);

   mhdr.msg_name = (char *)from; /* IP address of sender */
   mhdr.msg_namelen = sizeof(*from);
   mhdr.msg_iov = iov;  /* vector for scatter read */
   mhdr.msg_iovlen = 1; /* one buffer for scatter read */
   mhdr.msg_control = (char *)recvdstaddr;  /* ancillary rx data */
   mhdr.msg_controllen = *controllen;
   mhdr.msg_flags = 0;

   iov[0].iov_base = buf; /* data we receive from the peer */
   iov[0].iov_len = buflen;

   /* read data and ancillary data */
   if((nbytes = recvmsg(sd, &mhdr, 0)) == -1)
   {perror("recvmsg"); return -1;}

  *controllen = mhdr.msg_controllen;
   return nbytes;
}
```

Example 18 Receiving Ancillary Data

The implementation for sending a datagram with a specified source address varies depending on whether it is AF_INET (IPv4) or AF_INET6 (IPv6). The IPv6 socket interface simplifies this, (see RFC 3542 [Stevens et. al., 2003]). However, IPv4 ignores any ancillary data associated with the `sendmsg()` call, so it is necessary to create a separate reply socket and bind it to the desired local source address. A library function for doing this is shown in Example 19.

Notice that IPv6 readily supports ancillary data for UDP transmit, so the AF_INET6 case (Example 20) is straight forward (just a few lines of code). The AF_INET case, however, requires more than 20 lines of code and an additional five system calls (including the `udp_reply_sock()` routine) to perform the same action.

```
int udp_reply_sock(struct sockaddr_in *sin) /* needed for AF_INET only */
{
   int rsock, on = 1;

   if((rsock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
   {perror("reply socket"); return -1;}

   if(setsockopt(rsock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) == -1)
   {perror("reply setsockopt"); return -1;}

   if(setsockopt(rsock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on)) == -1)
   {perror("reply setsockopt"); return -1;}

   if(bind(rsock, (struct sockaddr *)sin, sizeof(*sin)) == -1)
   {perror("bind"); return -1;}

   return rsock;
}
```

Example 19 Receiving Destination Address from UDP

```
int udp_sendsrcaddr(int sd, void *buf, int buflen, struct sockaddr_storage *dst,
                  ancillary_t *srcaddr, unsigned int controllen,
                  unsigned int localport)
{
   struct msghdr mhdr;  /* ancillary data message header */
   struct cmsghdr *cmsg = NULL;
   struct iovec iov[1]; /* vector used to reference data buffer */
   struct sockaddr_in sin;
   int rsock;
   int nbytes;

   /* init message header */
   mhdr.msg_name = (char *)dst; /* destination IP address */
   mhdr.msg_namelen = dst->ss_len;
   mhdr.msg_iov = iov;  /* vector for gather write */
   mhdr.msg_iovlen = 1; /* one buffer for gather write */
   mhdr.msg_control = (char *)srcaddr;  /* buffer pointing to ancillary data */
   mhdr.msg_controllen = controllen;
   mhdr.msg_flags = 0;

   /* init data buffer */
   iov[0].iov_base = buf; /* data we receive from the peer */
   iov[0].iov_len = buflen;

   switch(dst->ss_family) {
   case AF_INET:
      /* sndmsg() for IPv4 ignores ancillary data, so must bind to new socket */
      sin.sin_len = dst->ss_len;
      sin.sin_family = dst->ss_family;
      sin.sin_port = htons(localport);

      /* assumes RECVDSTADDR is only element of ancillary data */
      cmsg = CMSG_NXTHDR(&mhdr, cmsg);
      if(cmsg) { /* if there is ancillary data to send then copy it */
               /* save IPv4 source address to sin */
       memcpy(&sin.sin_addr, CMSG_DATA(cmsg), sizeof(struct in_addr));

       /* create a new socket and bind to the local address in sin */
       if((rsock = udp_reply_sock(&sin)) == -1)
       {printf("reply_sock failed\n"); return 0;}
      }
      else rsock = sd; /* no ancillary data - use same socket */

      /* send data using replysock */
      if((nbytes = sendmsg(rsock, &mhdr, 0)) == -1)
      {perror(errno); return -1; }

      if(cmsg) close(rsock); /* finished with newly created reply sock */
      break;
   case AF_INET6:
      if(controllen == 0) mhdr.msg_control = NULL;

      /* call sendmsg to force IP source address */
      if((nbytes = sendmsg(sd, &mhdr, 0)) == -1)
      {perror(errno); return -1;}
      break;
   }
   return nbytes;
}
```


Example 20 UDP Reply with Source Address Set to Request's Destination Address

A UDP echo server might make use of these library functions in the following way:

```
len = sizeof(recvdstaddr);

if((bytes = udp_recvdstaddr(sd, buf, buflen, &rmtaddr, &recvdstaddr, &len)) <= 0)
{perror ("udp_recvdstaddr"); return -1;}

if(udp_sendsrcaddr(sd, buf, bytes, &rmtaddr, &recvdstaddr, len, port) == -1)
{perror("udp_sendsrcaddr"); return -1;}
```

Example 21 UDP Echo Server Using Correct Source Address

Concurrency In Server Applications

A server application must typically respond to many incoming connections in a concurrent manner. Concurrent handling of connections can be achieved by several different methods:

- Within a single-threaded application by using either `select()` or `$QIO()`
- Multithreaded application using pthreads
- Multiple processes, which may require additional interprocess communication (IPC)

The major difference between `select()` and `$QIO` is that `select()` will not return until one of its socket descriptors is ready for I/O. When `select()` returns, the application must poll each descriptor to determine which one caused it to return. In contrast, an asynchronous `$QIO()` will return immediately after having queued an AST routine that is called back when data is available. The argument passed to the AST routine uniquely describes the connection, so there is no need to perform further polling.

To measure the order of program complexity, consider an application with *"n"* connections. Because an algorithm using `select()` must poll each descriptor, this results in an algorithm with a complexity of *O(n)*. For asynchronous `$QIO()`, the argument passed to the AST routine describes the channel that has become ready. Hence, the `$QIO()` algorithm has a complexity of *O(1)*, which is far more efficient when *"n"* is large. Also, because the AST interrupts the mainline of processing, the `$QIO()` solution provides two code paths within the process (an AST code path and a process priority code path), although only one of these code paths will be active at a time.

In high-performance applications that handle many connections, consider using a multithreaded implementation. There are many options for designing a multithreaded solution. The choice of approach depends on many factors, including such concerns as overhead of process creation, program complexity, and type of service. For a discussion on choice of concurrency, see [Comer, 2001].

Separate processes operate similarly to a multithreaded application, except that in a multithreaded application, all threads share the same address space. This allows each thread to access global data directly, usually with the aid of mutex locks. Separate processes each have their private address space; so if there is a need to share data between the processes, a separate IPC mechanism must be implemented.

If the same TCP listen socket descriptor is used by multiple threads, OpenVMS will deliver new incoming requests to each of the listening threads in a round-robin fashion. Separate processes

may also share the same socket descriptor, provided it has first set the SO_SHARE socket option.

Conservation of Server Resources
Typically, a host provides a multitude of disparate services to many clients, with each service competing for system resources. Conserving server resources improves scalability and robustness. This is especially important in environments where the server may be exposed to attacks from poorly written clients or malicious clients.

Simplifying Address Conversion
Legacy IPv4 applications use a variety of BSD API functions to convert between a presentation form of an IP address and its binary format. Because a presentation form of an IP address may be in either *dotted-decimal* or *hostname* format, an application should first try the *dotted-decimal* form, and, if that fails, try to resolve the hostname, (RFC 1123, section 2.1 [Braden, 1989b]). Typical APIs include: `inet_addr(), inet_ntop(), inet_pton(), gethostbyname(), and gethostbyaddr()`.

With the introduction of IPv6, the various forms of IP addresses grew and the legacy APIs proved to be inadequate. Consequently, these APIs have been superceded by new and more powerful protocol-independent APIs. They are `getaddrinfo(), getnameinfo()`, and `freeaddrinfo()` (RFC 3493, section 6.1 [Gilligan et. al., 2003]). A new protocol-indendent structure used to describe socket addresses is `struct socket_storage.` RFC 3493 also describes `inet_pton()` and `inet_ntop()`, but these may also be replaced with `getaddrinfo()` and `getnameinfo()`, respectively.

An additional benefit of the `getaddrinfo()` function is that it returns an initialized socket address structure and other fields (embedded in `struct addrinfo`) that may be used directly in the `socket()` and `bind()` calls. This simplifies the code and makes it independent of the differences between IPv4 and IPv6 addresses. For example, a server application that is willing to accept connections from UDP/IP, TCP/IP, UDP/IPv6, and TCP/IPv6 might use the code in Example 22.

```
int sd[MAX_SOCKS]; /* one per address for: UDP/IP, UDP/IPv6, TCP/IP, TCP/IPv6 */
char *port, *addr = NULL;
struct addrinfo *res, hints;

port = argv[1]; /* port number as a string – must not be NULL */
if(argc == 3) addfrr = argv[2]; /* hostname – NULL implies ANY address */

memset(&hints, '\0', sizeof(hints));
hints.ai_flags = AI_PASSIVE; /* if usrreq.addr NULL, sets sockaddr to ANY */

err = getaddrinfo(usrreq.addr, usrreq.port, &hints, &res);
if(err) {
   if(err == EAI_SYSTEM) perror("getaddrinfo");
   else printf("getaddrinfo error %d - %s", err, gai_strerror(err));
   return 1;
}

i = 0;
for(aip = res; aip; aip = aip->ai_next) {
   if(aip->ai_family != AF_INET && aip->ai_family != AF_INET6) continue;
   /* create a socket for this protocol */
   sd[i] = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
   if(sd[i] < 0) {perror("socket"); return sd[i];}

   err = socket_options(sd[i], aip); /* set SO_REUSEADDR, SO_REUSPORT etc. */
   if(err) {perror("sockopt"); return 1;}
         err = bind(sd[i], res->ai_addr, res->ai_addrlen);
   if(err == -1) {perror("bind"); return 1;}

   if(i == NUM_ELT(sd)) {printf("Insufficient socket elements\n"); break;}
   i++;
}
freeaddrinfo(res);

/*
** TCP sockets prepare to accept incoming connections.
** UDP sockets are ready for data transfer.
*/
```

Example 22 Use of `getaddrinfo()` to Retrieve Address List

Summary

A network programmer is responsible for compensating for any unforeseen events that may affect the successful transfer of data, with the least impact to resources and performance of the systems involved.

To take advantage of a multihomed environment, you must be careful to use a method that ensures that no data is lost; even if a NIC fails during the transaction.

In the current environment of mixed IPv4 and IPv6 implementations, it makes sense to use the APIs that guarantee communication in either world.

With a solid basis of understanding about the way TCP/IP networking operates, you can ensure that your applications make the best use of the available resources in any type of environment.

This article has described a broad variety of best practices for the TCP/IP programmer in terms of the structure of a network program which was described in four phases: establish local context, connection establishment, data transfer, connection shutdown.

Establishing local context deals with selecting the appropriate protocol, creating and naming endpoints and preparing the socket for various functions depending on whether it is a client or server.

The connection establishment phase is different for TCP and UDP, where TCP connection establishment is controlled by a state machine and peers undergo a protocol exchange. A UDP connection affects local context only and merely associates the peer's address with the socket. Before a client attempts to connect to a server, it must resolve the server's hostname using the modern API, `getaddrinfo()`. The impact of a server application not being able to keep up with connection requests was also discussed. Once connected, the peers should monitor the connection with keepalive polls. Where TCP provides a keepalive mechanism, UDP leaves this up to the responsibility of the application.

Data transfer for TCP, a stream socket type, is often misunderstood. It is emphasized that TCP guarantees to deliver no more than one byte at a time and it is up to the receiving application to assemble these bytes into messages. On the other hand, UDP uses a datagram socket type which delivers datagrams in the same way they were sent. However, UDP was designed to be inherently unreliable and simple, so it is the repsonsibiltiy of the UDP application to provide for UDP behaviors.

Shutting down a connection for TCP applications should be done using the orderly release method, where the send side is shut down first, which notifies the peer that no more data will be sent. It is important to realize that the peer that initiates the shut down is forced to close the connection through the TCP TIME_WAIT state.

For More Information

As well as the specific references described below, there are many web sites, newsgroups, and FAQs dedicated to TCP/IP programming. Web-based search engines, such as http://www.google.com, provide a critical tool for locating information for the TCP/IP programmer.

Braden R. T., ed. 1989a., "Requirements for Internet Hosts -- Communication Layers", RFC 1122, (Oct.)

Braden R. T., ed. 1989b., "Requirements for Internet Hosts -- Application and Support", RFC 1123, (Oct.)

Comer D. E., Stevens, D. L., 2001. *Internetworking with TCP/IP Vol III: Client-Server Programming And Applications Linux/POSIX Sockets Version.* Prentice Hall, New Jersey.

Gilligan, R., Thomson, S., Bound, J., McCann, J., and Stevens, W. 2003. "Basic Socket Interface Extensions for IPv6", RFC 3493 (Feb).

Hewlett-Packard Company, 2001. *Compaq TCP/IP Services for OpenVMS, Sockets API and System Services Programming.* (Jan.)

Hewlett-Packard Company, 2003a. *HP TCP/IP Services for OpenVMS, Guide to IPv6.* (Sep.)

Hewlett-Packard Company, 2003b. *HP TCP/IP Services for OpenVMS, Management.* (Sep.)

Hewlett-Packard Company, 2003c. *HP TCP/IP Services for OpenVMS, Tuning and Troubleshooting*, (Sep.).

Muggeridge, M. J., 2003. *Configuring TCP/IP for High Availability.* OpenVMS Technical Journal V2.

Snader, J. C., 2000. *Effective TCP/IP Programming: 44 Tips to Improve Your Network Programs.* Addison Wesley, Reading, Mass.

Stevens, W., Thomas, M., Nordmark, E., and Jinmei, T. 2003. "Advanced Socket Application Program Interface (API) for IPv6", RFC 3542 (May).

Stevens, W. R., 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, Reading, Mass.

Wright, G.R., and Stevens, W. R., 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison Wesley, Reading, Mass.