8.11 `connect` Function with UDP

We mentioned at the end of Section 8.9 that an asynchronous error is not returned on a UDP socket unless the socket has been connected. Indeed, we are able to call `connect` (Section 4.3) for a UDP socket. But this does not result in anything like a TCP connection: There is no three-way handshake. Instead, the kernel just checks for any immediate errors (e.g., an obviously unreachable destination), records the IP address and port number of the peer (from the socket address structure passed to `connect`), and returns immediately to the calling process.

> Overloading the `connect` function with this capability for UDP sockets is confusing. If the convention that `sockname` is the local protocol address and `peername` is the foreign protocol address is used, then a better name would have been `setpeername`. Similarly, a better name for the `bind` function would be `setsockname`.

With this capability, we must now distinguish between

- An *unconnected UDP socket*, the default when we create a UDP socket
- A *connected UDP socket*, the result of calling `connect` on a UDP socket

With a connected UDP socket, three things change, compared to the default unconnected UDP socket:

1. We can no longer specify the destination IP address and port for an output operation. That is, we do not use `sendto`, but `write` or `send` instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by `connect`.

   > Similar to TCP, we can call `sendto` for a connected UDP socket, but we cannot specify a destination address. The fifth argument to `sendto` (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

2. We do not need to use `recvfrom` to learn the sender of a datagram, but `read`, `recv`, or `recvmsg` instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in `connect`. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket. This limits a connected UDP socket to exchanging datagrams with one and only one peer.

   > Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to `connect` to a multicast or broadcast address.

3. Asynchronous errors are returned to the process for connected UDP sockets.

   The corollary, as we previously described, is that unconnected UDP sockets do not receive asynchronous errors.

Figure 8.14 summarizes the first point in the list with respect to 4.4BSD.

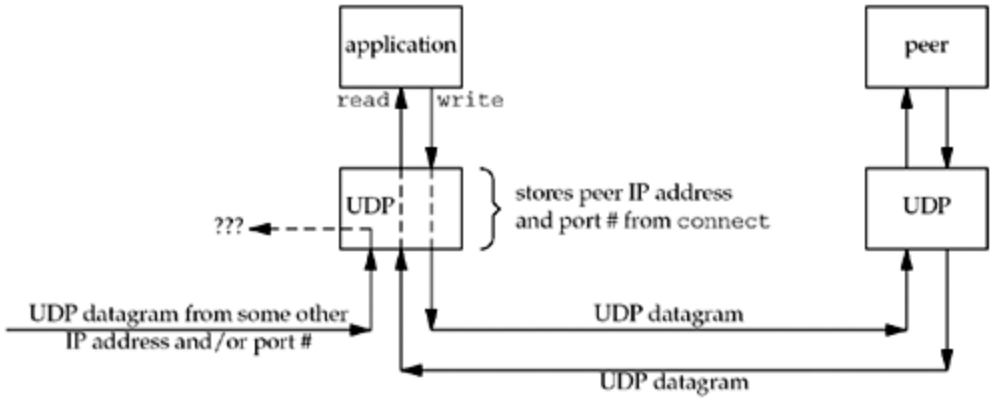
Figure 8.14. TCP and UDP sockets: can a destination protocol address be specified?

Type of socket	`write` or `send`	`sendto` that does not specify a destination	`sendto` that specifies a destination
TCP socket	OK	OK	`EISCONN`
UDP socket, connected	OK	OK	`EISCONN`
UDP socket, unconnected	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

The POSIX specification states that an output operation that does not specify a destination address on an unconnected UDP socket should return `ENOTCONN`, not `EDESTADDRREQ`.

Figure 8.15 summarizes the three points that we made about a connected UDP socket.

Figure 8.15. Connected UDP socket.



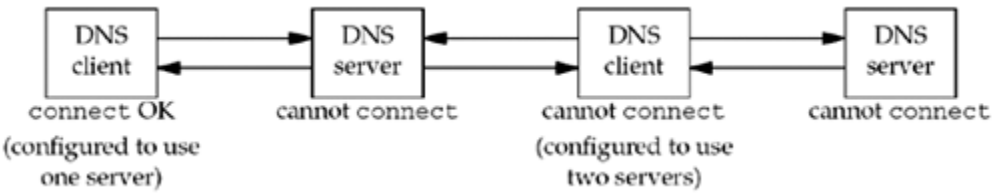
The application calls `connect`, specifying the IP address and port number of its peer. It then uses `read` and `write` to exchange data with the peer.

Datagrams arriving from any other IP address or port (which we show as "???" in Figure 8.15) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is `connected`. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.

In summary, we can say that a UDP client or server can call `connect` only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls `connect`, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call `connect`.

The DNS provides another example, as shown in Figure 8.16.

Figure 8.16. Example of DNS clients and servers and the `connect` function.



A DNS client can be configured to use one or more servers, normally by listing the IP addresses of the servers in the file `/etc/resolv.conf`. If a single server is listed (the leftmost box in the figure), the client can call `connect`, but if multiple servers are listed (the second box from the right in the figure), the client cannot call `connect`. Also, a DNS server normally handles any client request, so the servers cannot call `connect`.

Calling `connect` Multiple Times for a UDP Socket

A process with a connected UDP socket can call `connect` again for that socket for one of two reasons:

- To specify a new IP address and port
- To unconnect the socket

The first case, specifying a new peer for a connected UDP socket, differs from the use of `connect` with a TCP socket: `connect` can be called only one time for a TCP socket.

To unconnect a UDP socket, we call `connect` but set the family member of the socket address structure (`sin_family` for IPv4 or `sin6_family` for IPv6) to `AF_UNSPEC`. This might return an error of `EAFNOSUPPORT` (p. 736 of TCPv2), but that is acceptable. It is the process of calling `connect` on an already connected UDP socket that causes the socket to become unconnected (pp. 787–788 of TCPv2).

> The Unix variants seem to differ on exactly how to unconnect a socket, and you may encounter approaches that work on some systems and not others. For example, calling `connect` with `NULL` for the address works only on some systems (and on some, it only works if the third argument, the length, is nonzero). The POSIX specification and BSD man pages are not much help here, only mentioning that a *null address* should be used and not mentioning the error return (even on success) at all. The most portable solution is to zero out an address structure, set the family to `AF_UNSPEC` as mentioned above, and pass it to `connect`.
>
> Another area of disagreement is around the local binding of a socket during the unconnect process. AIX keeps both the chosen local IP address and the port, even from an implicit bind. FreeBSD and Linux set the local IP address back to all zeros, even if you previously called `bind`, but leave the port number intact. Solaris sets the local IP address back to all zeros if it was

assigned by an implicit bind; but if the program called `bind` explicitly, then the IP address remains unchanged.

Performance

When an application calls `sendto` on an unconnected UDP socket, Berkeley-derived kernels temporarily connect the socket, send the datagram, and then unconnect the socket (pp. 762–763 of TCPv2). Calling `sendto` for two datagrams on an unconnected UDP socket then involves the following six steps by the kernel:

- Connect the socket
- Output the first datagram
- Unconnect the socket
- Connect the socket
- Output the second datagram
- Unconnect the socket

Another consideration is the number of searches of the routing table. The first temporary connect searches the routing table for the destination IP address and saves (caches) that information. The second temporary connect notices that the destination address equals the destination of the cached routing table information (we are assuming two `sendtos` to the same destination) and we do not need to search the routing table again (pp. 737–738 of TCPv2).

When the application knows it will be sending multiple datagrams to the same peer, it is more efficient to connect the socket explicitly. Calling `connect` and then calling `write` two times involves the following steps by the kernel:

- Connect the socket
- Output first datagram
- Output second datagram

In this case, the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when `sendto` is called twice. [Partridge and Pink 1993] note that the temporary connecting of an unconnected UDP socket accounts for nearly one-third of the cost of each UDP transmission.