How Do Sockets Work in C?

Asked 16 years, 3 months ago Modified 9 years, 4 months ago

Viewed 8k times

21

I am a bit confused about socket programming in C.

You create a socket, bind it to an interface and an IP address and get it to listen. I found a couple of web resources on that, and understood it fine. In particular, I found an article ~~Network programming under Unix systems~~ to be very informative.

What confuses me is the timing of data arriving on the socket.

How can you tell when packets arrive, and how big the packet is, do you have to do all the heavy lifting yourself?

My basic assumption here is that packets can be of variable length, so once binary data starts appearing down the socket, how do you begin to construct packets from that?

c sockets network-programming

Share

Improve this question

Follow

edited Aug 8, 2015 at 17:26

Enamul Hassan

5,445 ● 23 ● 43 ● 58

asked Sep 8, 2008 at 0:00

kaybenleroll
17k ● 16 ● 56 ● 66

4 Answers

Sorted by: Highest score (default)

18

Short answer is that you have to do all the heavy lifting yourself. You can be notified that there is data available to be read, but you won't know how many bytes are available. In most IP protocols that use variable length packets, there will be a header with a known fixed length prepended to the packet. This header will contain the length of the packet. You read the header, get the length of the packet, then read the packet. You repeat this pattern (read header, then read packet) until communication is complete.

When reading data from a socket, you request a certain number of bytes. The read call may block until the requested number of bytes are read, but it can return fewer bytes than what was requested. When this happens, you simply retry the read, requesting the remaining bytes.

Here's a typical C function for reading a set number of bytes from a socket:

```
/* buffer points to memory block that is bigger than t
read */
/* socket is open socket that is connected to a sender
/* bytesToRead is the number of bytes expected from th
/* bytesRead is a pointer to a integer variable that w
bytes */
/*           actually received from the sender. */
/* The function returns either the number of bytes rea
/*                                0 if the socket was clo
/*                               -1 if an error occurred
socket */
int readBytes(int socket, char *buffer, int bytesToRea
{
    *bytesRead = 0;
    while(*bytesRead < bytesToRead)
    {
        int ret = read(socket, buffer + *bytesRead, by
        if(ret <= 0)
        {
           /* either connection was closed or an error
           return ret;
        }
        else
        {
           *bytesRead += ret;
        }
    }
    return *bytesRead;
}
```

Share Improve this answer

Follow

edited Jul 1, 2012 at 5:40

slashmais
7,137 ● 9 ● 56 ● 80

answered Sep 8, 2008 at 0:27

dfjacobs
1,065 ● 11 ● 17

Shouldn't the early return after the closed connection be "return bytesRead;" instead of "return ret;"? I presume the readBytes function is meant to return the actual number of bytes read. I suppose you could define it to return a non-positive integer if there is an error, but you could also detect that an error occurred by checking to see if readBytes returns a different number of bytes than requested. – A. Levy May 20, 2009 at 2:33

a negative value to flag an error state is common practice in C programming and should be adhered to as much as possible to prevent confusion by the user of the API. – Guss Jan 8, 2011 at 23:58

I modified my example function to return the number of bytes read even if an error occurred during the read. One point to keep in mind with sockets is that a read that returns zero indicates that the connection was closed, and may not necessarily be an error. – dfjacobs Feb 24, 2011 at 7:29

13

So, the answer to your question depends a fair bit on whether you are using UDP or TCP as your transport.

For UDP, life gets a lot simpler, in that you can call recv/recvfrom/recvmsg with the packet size you need (you'd likely send fixed-length packets from the source anyway), and make the assumption that if data is available, it's there in multiples of packet-length sizes. (I.E. You call recv* with the size of your sending side packet, and you're set.)

For TCP, life gets a bit more interesting - for the purpose of this explanation, I will assume that you already know how to use socket(), bind(), listen() and accept() - the latter being how you get the file descriptor (FD) of your newly made connection.

There are two ways of doing the I/O for a socket - blocking, in which you call read(fd, buf, N) and the read sits there and waits until you've read N bytes into buf - or non-blocking, in which you have to check (using select() or poll()) whether the FD is readable, and THEN do your read().

When dealing with TCP-based connections, the OS doesn't pay attention to the packet sizes, since it's considered a continual stream of data, not seperate packet-sized chunks.

If your application uses "packets" (packed or unpacked data structures that you're passing around), you ought to be able to call read() with the proper size argument, and read an entire data structure off the socket at a time. The only caveat you have to deal with, is to remember to properly byte-order any data that you're sending, in case the source and destination system are of different byte endian-ness. This applies to both UDP and TCP.

As far as *NIX socket programming is concerned, I highly recommend W. Richard Stevens' "Unix Network Programming, Vol. 1" (UNPv1) and "Advanced Programming in an Unix Environment" (APUE). The first is a tome regarding network-based programming, regardless of the transport, and the latter is a good all-around programming book as it applies to *NIX based programming. Also, look for "TCP/IP Illustrated", Volumes 1 and 2.

Share Improve this answer Follow

answered Sep 8, 2008 at 3:45

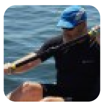
Dominic Eidson
779 ● 4 ● 8

3

When you do a read on the socket, you tell it how many maximum bytes to read, but if it doesn't have that many, it gives you however many it's got. It's up to you to design the protocol so you know whether you've got a partial packet or not. For instance, in the past when sending variable length binary data, I would put an int at the beginning that said how many bytes to expect. I'd do a read requesting a number of bytes greater than the largest possible packet in my protocol, and then I'd compare the first int against however many bytes I'd received, and either process it or try more reads until I'd gotten the full packet, depending.

Share Improve this answer Follow

answered Sep 8, 2008 at 0:05



Paul Tomblin

183k ● 59 ● 323 ● 410

1

Sockets operate at a higher level than raw packets - it's like a file you can read/write from. Also, when you try to read from a socket, the operating system will block (put on hold) your process until it has data to fulfill the request.

Share Improve this answer Follow

answered Sep 8, 2008 at 0:06

Kyle Cronin
79k ● 45 ● 151 ● 167

1 This is only true if the socket is not set to be non-blocking. – Dominic Eidson Sep 24, 2008 at 3:09