Socket Programming HOWTO

- Sockets
  - History
- Creating a Socket
  - IPC
- Using a Socket
  - Binary Data
- Disconnecting
  - When Sockets Die
- Non-blocking Sockets

Abstract

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things operational. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

Sockets

I'm only going to talk about INET (i.e. IPv4) sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM (i.e. TCP) sockets - unless you really know what you're doing (in which case this HOWTO isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that "socket" can mean a number of subtly different things, depending on context. So first, let's make a distinction between a "client" socket - an endpoint of a conversation, and a "server" socket, which is more like a switchboard operator. The client application (your browser, for example) uses "client" sockets exclusively; the web server it's talking to uses both "server" sockets and "client" sockets.

History

Of the various forms of `IPC (Inter Process Communication)`, sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the internet. With good reason --- the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(("www.python.org", 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a "server socket":

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a "server" socket, but one that was only visible within the same machine. `s.bind(('', 80))` specifies that the socket is reachable by any address the machine happens to have.

A second thing to note: low number ports are usually reserved for "well known" services (HTTP, SNMP etc). If you're playing around, use a nice high number (4 digits).

Finally, the argument to `listen` tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a "server" socket, listening on port 80, we can enter the mainloop of the web server:

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = make_client_thread(clientsocket)
    ct.start()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our "server" socket and any active `clientsocket`s using `select`. More about that later. The important thing to understand now is this: this is *all* a "server" socket does. It doesn't send any data. It doesn't receive any data. It just produces "client" sockets. Each `clientsocket` is created in response to some *other* "client" socket doing a `connect()` to the host and port we're bound to. As soon as we've created that `clientsocket`, we go back to listening for more connections. The two "clients" are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

IPC

If you need fast IPC between two processes on one machine, you should look into pipes or shared memory. If you do decide to use AF_INET sockets, bind the "server" socket to `'localhost'`. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

The `multiprocessing` integrates cross-platform IPC into a higher-level API.

Using a Socket

The first thing to note, is that the web browser's "client" socket and the web server's "client" socket are identical beasts. That is, this is a "peer to peer" conversation. Or to put it another way, *as the designer, you will have to decide what the rules of etiquette are for a conversation*. Normally, the `connect`ing socket starts the conversation, by sending in a request, or perhaps a signon. But that's a design decision - it's not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send` and `recv`, or you can transform your client socket into a file-like beast and use `read` and `write`. The latter is the way Java presents its sockets. I'm not going to talk about it here, except to warn you that you need to use `flush` on sockets. These are buffered "files", and a common mistake is to `write` something, and then `read` for a reply. Without a `flush` in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - `send` and `recv` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (`send`) or emptied (`recv`). They then tell you how many bytes they handled. It is *your* responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully; I'll talk more about this later.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that *there is no* `EOT (End of Transfer)` *on a socket.* I repeat: if a socket `send` or `recv` returns after handling 0 bytes, the connection has been broken. If the connection has *not* been broken, you may wait on a `recv` forever, because the socket will *not* tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: *messages must either be fixed length* (yuck), *or be delimited* (shrug), *or indicate how long they are* (much better), *or end by shutting down the connection*. The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```
class MySocket:
    """demonstration class only
      - coded for clarity, not efficiency
```

```
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a

delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You'll need to put that aside and hold onto it, until it's needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

Binary Data

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, network byte order is big-endian, with the most significant byte first, so a 16 bit integer with the value `1` would be the two hex bytes `00 01`. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same `1` would be `01 00`.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl, htonl, ntohs, htons` where "n" means *network* and "h" means *host*, "s" means *short* and "l" means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, most integers have the value 0, or maybe 1. The string `"0"` would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

Disconnecting

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen", or "I'm not listening, good riddance!". Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is

not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server "This client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please* `close` your sockets when you're done.

When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. TCP is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(False)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable POSIX flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list. You'll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a "server" socket, put it in the potential_readers list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

Actually, `select` can be handy even with blocking sockets. It's one way of determining whether you will block - the socket returns as readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets.

□□□ #2
□ □□□ □□□ 2026-10-31 17:00:41 UTC
□ □□□ □□□ 2026-10-31 17:00:44 UTC