

# Re: network programming: how does s.accept() work?

---

*Source:* <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2008-02/msg03585.html>

---

- *From:* Micah Cowan <[micah@xxxxxxxxxxx](mailto:micah@xxxxxxxxxxx)>
  - *Date:* Tue, 26 Feb 2008 10:05:48 -0800
- 

Hrvoje Niksic wrote:

7stud <[bbxx789\\_05ss@xxxxxxxxxxx](mailto:bbxx789_05ss@xxxxxxxxxxx)> writes:

When you surf the Web, say to <http://www.google.com>, your Web browser is a client. The program you contact at Google is a server. When a server is run, it sets up business at a certain port, say 80 in the Web case. It then waits for clients to contact it. When a client does so, the server will usually assign a new port, say 56399, specifically for communication with that client, and then resume watching port 80 for new requests.

Actually the client is the one that allocates a new port. All connections to a server remain on the same port, the one it listens on:

(Hey, I know you! ;) )

Right.

7stud, what you seem to be missing, and what I'm not sure if anyone has clarified for you (I have only skimmed the thread), is that in TCP, connections are uniquely identified by a /pair/ of sockets (where "socket" here means an address/port tuple, not a file descriptor). It is fine for many, many connections, using the same local port and IP address, so long as the other end has either a different IP address or a different port. There is no issue with lots of processes sharing the same socket for various separate connections, because the /pair/ of sockets is what identifies them. See the "Multiplexing" portion of section 1.5 of the TCP spec (<http://www.ietf.org/rfc/rfc0793.txt>).

Reading some of what you've written elsewhere on this thread, you seem to be confusing this address/port stuff with what accept() returns. This is hardly surprising, as unfortunately, both things are called "sockets": the former is called a socket in the various RFCs, the latter

## Re: network programming: how does s.accept() work?

is called a socket in documentation for the Berkeley sockets and similar APIs. What `accept()` returns is a new file descriptor, but the local address-and-port associated with this new thing is still the very same ones that were used for `listen()`. All the incoming packets are still directed at port 80 (say) of the local server by the remote client.

It's probably worth mentioning at this point, that while what I said about many different processes all using the same local address/port combination is true, in implementations of the standard Berkeley sockets API the only way you'd \_arrive\_ at that situation is that all of those different connections that have the same local address/port combination is that they all came from the same `listen()` call (ignoring mild exceptions that involve one server finishing up connections while another accepts new ones). Because while one process has a socket descriptor bound to a particular address/port, no other process is allowed to bind to that combination. However, for listening sockets, that one process is allowed to accept many connections on the same address/port. It can handle all those connections itself, or it can fork new processes, or it can pass these connected sockets down to already-forked processes. But all of them continue to be bound to the same local address-and-port.

Note that, if the server's port were to change arbitrarily for every successful call to `accept()`, it would make it much more difficult to filter and/or analyze TCP traffic. If you're running, say, `tcpdump`, the knowledge that all the packets on a connection that was originally directed at port 80 of `google.com`, will continue to go to port 80 at `google.com` (even though there are many, many, many other connections out there on the web from other machines that are all also directed at port 80 of `google.com`), is crucial to knowing which packets to watch for while you're looking at the traffic.

--

HTH,

Micah J. Cowan

Programmer, musician, typesetting enthusiast, gamer...

<http://micah.cowan.name/>

.