is TCP handshake handled by web application flask or web servers like nginx?

Asked 1 year, 3 months ago Modified 1 year, 3 months ago Viewed 776 times

0

if I run a simple flask application in development mode(so web server and web application is the flask library) on port 5000. is there a way to see SYN, SYN-ack, ack requests that are exchanged between client and server? for example, print them? I don't want to use tools like Wireshark to see the traffic, I want to know who is handling the process.

```
from flask import Flask
app = Flask(__name__)


@app.route('/')
def index():
    return 'hello'
```

when I do `curl localhost:5000/` the curl first sends the SYN? who will answer it? the flask?

what can I do to see all of this process on my own app?? also, I want to see how the keep-alive mechanism of HTTP/1.1 works?

tcp flask

Share Improve this question Follow

asked Feb 4, 2022 at 13:55
kankan256
103 ●4

generally the TCP/IP stack is implemented by the OS, not the individual applications. this allows the system to solve the many multi-application coordination issues that modern networking must contend with. – Frank Thomas Feb 4, 2022 at 14:10

@FrankThomas so how can I send an SYN packet to a computer in my local network manually? is there a tool. and if the handshake is handled by OS, in the case of HTTP/1.1 how does flask tell the OS that keeps the TCP connection to use it for other requests? – kankan256 Feb 4, 2022 at 14:18

1 @kankan256: "Raw sockets" can be used to send arbitrary packets, bypassing the OS TCP stack; so yes, there are several tools that help with building custom packets. See update. – user1686 Feb 4, 2022 at 14:37

1 Answer

Sorted by: Highest score (default)

1

> is TCP handshake handled by web application flask or web servers like nginx?

Neither. For TCP, the handshake is handled entirely by the operating system.

cURL initiates the connection using "sockets API" provided by the OS – usually socket() and connect() functions – while Flask's devel-server receives it using listen() and accept(). By the time the accept() function returns, the entire TCP handshake has already been completed and Flask can use the socket for data.

This means that the process you're trying to see isn't really part of your app at all. Flask only uses Python's `socket.socket()` objects, which then directly call the corresponding OS functions – you would need to use OS tracing tools such as dtrace/bpftrace/systemtap to see the internal kernel calls that happen.

Though note that QUIC (as in HTTP/3) works slightly differently – there is no direct QUIC support in operating systems, so it *is* cURL and Nginx that perform the QUIC handshake, although they still use libraries like ngtcp2 or MsQuic to do all the work. (The endpoints still use the "socket API" to talk to the OS, but for QUIC they create UDP sockets which implement no handshake of their own.)

> how can I send an SYN packet to a computer in my local network manually?

Similarly to packet capture tools, there are also *packet generation* tools which let you craft custom packets (possibly even to the point of making your own TCP stack). One of them is `scapy`, which is written in Python and lets you use the Python REPL to build packets by hand. Another similar option is the `pypacker` module, again in Python.

Such tools generally use "raw sockets", which are quite similar to UDP ones, except the app can include its own IP header. You'll usually need root privileges to use this.

> also, I want to see how the keep-alive mechanism of HTTP/1.1 works? [...] how does flask tell the OS that keeps the TCP connection to use it for other requests?

In general, once a TCP connection is open, it simply **remains open** unless the process deliberately closes it (or unless it exits, in which case the OS closes the connection). The OS doesn't keep track of "requests" – that's an application matter; TCP can transfer data at any time in any direction.

So the entire HTTP/1.1 "keep-alive" mechanism is that the server literally doesn't close the TCP connection after the response is sent, i.e. it doesn't call shutdown() or close(). This allows the client to send more HTTP requests over that connection. There's nothing more to it – neither the client nor server make any effort to *keep* the connection alive, they just avoid terminating it in the first place.

(There is an optional HTTP header which allows HTTP/1.0 clients to request keep-alive, or HTTP/1.1 clients to prevent it.)

Of course the mechanism only works for clients that remain running (and keep the OS socket open) between requests. For example, the `curl` or `wget` CLI tools can be given multiple URLs to download, and after establishing the connection for the first URL, they'll keep it open for the next one (assuming it's from the same server). But the connection always gets closed as soon as the tool exits – running `curl` a second time *has* to make a brand new connection.

Similarly, Python's `requests.Session()` will use 'keep-alive' to automatically re-use the connection across multiple `.get()` or `.post()` calls. If the script exits, though, the connection gets closed with it.

Note however that some *other* protocols also have a "keep-alive" feature that works differently, by sending explicit "I'm still running" packets (e.g. a keepalive packet might be sent every minute). This can be done in one of two ways – either the packets carry actual data generated by the application, *or* they're 0-length TCP packets sent by the OS after the application enables this feature. (The latter are called "TCP keepalives"; they aren't needed to keep a normal connection open between two hosts, but they're sometimes used to prevent *intermediate* systems – like your home NAT router – from forcefully closing it.)

Share Improve this answer Follow

edited Feb 4, 2022 at 14:36

answered Feb 4, 2022 at 14:16

user1686
413k ● 61 ● 867 ● 933