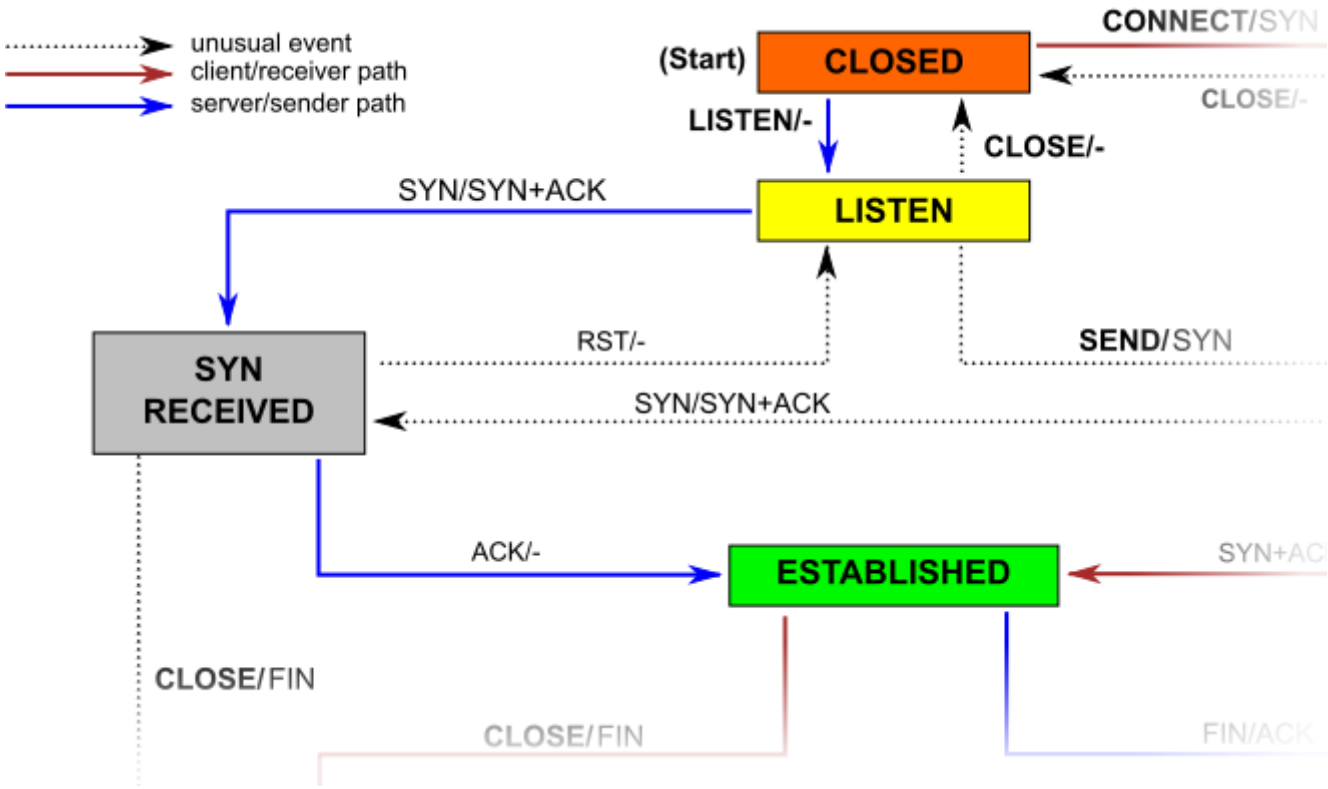
How TCP backlog works in Linux

#Linux (/#tech;Linux) #TCP/IP (/#tech;TCP/IP)

January 1, 2014 (updated March 14, 2015)

When an application puts a socket into LISTEN state using the `listen` (http://linux.die.net/man/2/listen) syscall, it needs to specify a backlog for that socket. The backlog is usually described as the limit for the queue of incoming connections.



Because of the 3-way handshake used by TCP, an incoming connection goes through an intermediate state SYN RECEIVED before it reaches the ESTABLISHED state and can be returned by the `accept` (http://linux.die.net/man/2/accept) syscall to the application (see the part of the TCP state diagram (http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg) reproduced above). This means that a TCP/IP stack has two options to implement the backlog queue for a socket in LISTEN state:

1. The implementation uses a single queue, the size of which is determined by the `backlog` argument of the `listen` syscall. When a SYN packet is received, it sends back a SYN/ACK packet and adds the connection to the queue. When the corresponding ACK is received, the connection changes its state to ESTABLISHED

and becomes eligible for handover to the application. This means that the queue can contain connections in two different state: SYN RECEIVED and ESTABLISHED. Only connections in the latter state can be returned to the application by the `accept` syscall.

2. The implementation uses two queues, a SYN queue (or incomplete connection queue) and an accept queue (or complete connection queue). Connections in state SYN RECEIVED are added to the SYN queue and later moved to the accept queue when their state changes to ESTABLISHED, i.e. when the ACK packet in the 3-way handshake is received. As the name implies, the `accept` call is then implemented simply to consume connections from the accept queue. In this case, the `backlog` argument of the `listen` syscall determines the size of the accept queue.

Historically, BSD derived TCP implementations use the first approach. That choice implies that when the maximum backlog is reached, the system will no longer send back SYN/ACK packets in response to SYN packets. Usually the TCP implementation will simply drop the SYN packet (instead of responding with a RST packet) so that the client will retry. This is what is described in section 14.5, *`listen` Backlog Queue* in W. Richard Stevens' classic textbook *TCP/IP Illustrated, Volume 3*.

Note that Stevens actually explains that the BSD implementation does use two separate queues, but they behave as a single queue with a fixed maximum size determined by (but not necessary exactly equal to) the `backlog` argument, i.e. BSD logically behaves as described in option 1:

> The queue limit applies to the sum of […] the number of entries on the incomplete connection queue […] and […] the number of entries on the completed connection queue […].

On Linux, things are different, as mentioned in the man page (http://linux.die.net/man/2/listen) of the `listen` syscall:

> The behavior of the `backlog` argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for *completely* established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using `/proc/sys/net/ipv4/tcp_max_syn_backlog`.

This means that current Linux versions use the second option with two distinct queues: a SYN queue with a size specified by a system wide setting and an accept queue with a size specified by the application.

The interesting question is now how such an implementation behaves if the accept queue is full and a connection needs to be moved from the SYN queue to the accept queue, i.e. when the ACK packet of the 3-way handshake is received. This case is handled by the

`tcp_check_req` function in `net/ipv4/tcp_minisocks.c`. The relevant code reads:

```
        child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
        if (child == NULL)
                goto listen_overflow;
```

For IPv4, the first line of code will actually call `tcp_v4_syn_recv_sock` in `net/ipv4/tcp_ipv4.c`, which contains the following code:

```
        if (sk_acceptq_is_full(sk))
                goto exit_overflow;
```

We see here the check for the accept queue. The code after the `exit_overflow` label will perform some cleanup, update the `ListenOverflows` and `ListenDrops` statistics in `/proc/net/netstat` and then return `NULL`. This will trigger the execution of the `listen_overflow` code in `tcp_check_req`:

```
listen_overflow:
        if (!sysctl_tcp_abort_on_overflow) {
                inet_rsk(req)->acked = 1;
                return NULL;
        }
```

This means that unless `/proc/sys/net/ipv4/tcp_abort_on_overflow` is set to 1 (in which case the code right after the code shown above will send a RST packet), the implementation basically does… nothing!

To summarize, if the TCP implementation in Linux receives the ACK packet of the 3-way handshake and the accept queue is full, it will basically ignore that packet. At first, this sounds strange, but remember that there is a timer associated with the SYN RECEIVED state: if the ACK packet is not received (or if it is ignored, as in the case considered here), then the TCP implementation will resend the SYN/ACK packet (with a certain number of retries specified by `/proc/sys/net/ipv4/tcp_synack_retries` and using an exponential backoff (http://en.wikipedia.org/wiki/Exponential_backoff) algorithm).

This can be seen in the following packet trace for a client attempting to connect (and send data) to a socket that has reached its maximum backlog:

```
  0.000  127.0.0.1 -> 127.0.0.1  TCP 74 53302 > 9999 [SYN] Seq=0 Len=0
  0.000  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
  0.000  127.0.0.1 -> 127.0.0.1  TCP 66 53302 > 9999 [ACK] Seq=1 Ack=1 Len=0
  0.000  127.0.0.1 -> 127.0.0.1  TCP 71 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 L
  0.207  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
  0.623  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
  1.199  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
  1.199  127.0.0.1 -> 127.0.0.1  TCP 66 [TCP Dup ACK 6#1] 53302 > 9999 [ACK]
  1.455  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
  3.123  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
  3.399  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
  3.399  127.0.0.1 -> 127.0.0.1  TCP 66 [TCP Dup ACK 10#1] 53302 > 9999 [ACK]
  6.459  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
  7.599  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
  7.599  127.0.0.1 -> 127.0.0.1  TCP 66 [TCP Dup ACK 13#1] 53302 > 9999 [ACK]
 13.131  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
 15.599  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
 15.599  127.0.0.1 -> 127.0.0.1  TCP 66 [TCP Dup ACK 16#1] 53302 > 9999 [ACK]
 26.491  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
 31.599  127.0.0.1 -> 127.0.0.1  TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 L
 31.599  127.0.0.1 -> 127.0.0.1  TCP 66 [TCP Dup ACK 19#1] 53302 > 9999 [ACK]
 53.179  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
106.491  127.0.0.1 -> 127.0.0.1  TCP 71 [TCP Retransmission] 53302 > 9999 [PS
106.491  127.0.0.1 -> 127.0.0.1  TCP 54 9999 > 53302 [RST] Seq=1 Len=0
```

Since the TCP implementation on the client side gets multiple SYN/ACK packets, it will assume that the ACK packet was lost and resend it (see the lines with `TCP Dup ACK` in the above trace). If the application on the server side reduces the backlog (i.e. consumes an entry from the accept queue) before the maximum number of SYN/ACK retries has been reached, then the TCP implementation will eventually process one of the duplicate ACKs, transition the state of the connection from SYN RECEIVED to ESTABLISHED and add it to the accept queue. Otherwise, the client will eventually get a RST packet (as in the sample shown above).

The packet trace also shows another interesting aspect of this behavior. From the point of view of the client, the connection will be in state ESTABLISHED after reception of the first SYN/ACK. If it sends data (without waiting for data from the server first), then that data will be retransmitted as well. Fortunately TCP slow-start (http://en.wikipedia.org/wiki/Slow-start) should limit the number of segments sent during this phase.

On the other hand, if the client first waits for data from the server and the server never reduces the backlog, then the end result is that on the client side, the connection is in state ESTABLISHED, while on the server side, the connection is considered CLOSED. This means that we end up with a half-open connection (http://en.wikipedia.org/wiki/Half-open_connection)!

There is one other aspect that we didn't discuss yet. The quote from the `listen` man page suggests that every SYN packet would result in the addition of a connection to the SYN queue (unless that queue is full). That is not exactly how things work. The reason is the following code in the `tcp_v4_conn_request` function (which does the processing of SYN packets) in `net/ipv4/tcp_ipv4.c`:

```
        /* Accept backlog is full. If we have already queued enough
         * of warm entries in syn queue, drop request. It is better than
         * clogging syn queue with openreqs with exponentially increasing
         * timeout.
         */
        if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
                goto drop;
        }
```

What this means is that if the accept queue is full, then the kernel will impose a limit on the rate at which SYN packets are accepted. If too many SYN packets are received, some of them will be dropped. In this case, it is up to the client to retry sending the SYN packet and we end up with the same behavior as in BSD derived implementations.

To conclude, let's try to see why the design choice made by Linux would be superior to the traditional BSD implementation. Stevens makes the following interesting point:

> The backlog can be reached if the completed connection queue fills (i.e., the server process or the server host is so busy that the process cannot call `accept` fast enough to take the completed entries off the queue) or if the incomplete connection queue fills. The latter is the problem that HTTP servers face, when the round-trip time between the client and server is long, compared to the arrival rate of new connection requests, because a new SYN occupies an entry on this queue for one round-trip time. […]
>
> The completed connection queue is almost always empty because when an entry is placed on this queue, the server's call to `accept` returns, and the server takes the completed connection off the queue.

The solution suggested by Stevens is simply to increase the backlog. The problem with this is that it assumes that an application is expected to tune the backlog not only taking into account how it intents to process newly established incoming connections, but also in function of traffic characteristics such as the round-trip time. The implementation in Linux effectively separates these two concerns: the application is only responsible for tuning the backlog such that it can call `accept` fast enough to avoid filling the accept queue); a system administrator can then tune `/proc/sys/net/ipv4/tcp_max_syn_backlog` based on traffic characteristics.

caimaoy • 6 months ago

Excellent

3 • Reply • Share ›

Yuwei • 2 years ago

This is a great article for understanding the socket `listen` and `accept` mechanism. However may I ask you a question here is I have a sample code here: https://gist.github.com/3bd...

First, I run it on my macOS and use `nc` to connect to the server, the behavior is:
1. accept the first `nc` connection
2. the second `nc` connection is hanging and resending the SYN packet

Then, I run it on a Linux machine to do the same thing:
1. accept the first connection
2. accept the second connection!
3. the third connection hangs and resending the SYN packet, the server at the same time, resending the SYN/ACK.

My question is, why the server can accept 2 connections on Linux and only on on macOS. Am I get the misunderstanding of the backlog QUEUE here?

1 • Reply • Share ›

SelfBoot ➚ Yuwei • a year ago

Did you have found the answer? If you have, please let me know. Thanks.

PS: I also propose the question on https://stackoverflow.com/q...

• Reply • Share ›

Yuwei ➚ SelfBoot • a year ago

actually not yet.. Maybe I'll post my tcpdump output here when I free to see there is anything helpful.

21 • Reply • Share ›

SelfBoot ➚ Yuwei • a year ago

@Andreas Veithen May you explain this question, I'm stuck here too.

^ v • Reply • Share ›

junjian ➚ SelfBoot • a year ago

I think this is because that no connection would be ever accepted hence a single connection would make the queue get stuck.

^ v • Reply • Share ›

Good Be3r • 4 months ago

Hi,

I found your article (very well explain by the way) looking for an explaination about recv-q and send-q displayed with netstat (sadly not ss) for the sockets in LISTENING state.

This informations (since kernel 2.6.18) should match SYN backlog and maximum SYN backlog but seems to be always set to zero for both, no matter what is set in the listen() function...

Do you know a way to actually change this values ?

^ v • Reply • Share ›

Philip Champon • 5 months ago

I wonder, how do TcpExtTCPBacklogDrop and TcpExtTCPPrequeueDropped fit into this conversation? I've got an abundance of both, on some very busy servers, which are using jumbo frames. I tried tweaking tcp_max_syn_backlog, somaxconn, I haven't successfully updated the application, which seems to default to a listen queue of 50... That's next on the todo list.

^ v • Reply • Share ›

SelfBoot • a year ago

May I ask what tool did you use for the packet trace? How did you get something like this:

```
0.000 127.0.0.1 -> 127.0.0.1 TCP 74 53302 > 9999 [SYN]
Seq=0 Len=0
0.000 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK]
Seq=0 Ack=1 Len=0
```

^ v • Reply • Share ›

Andreas Veithen Mod ➚ SelfBoot • a year ago

tcpdump

1 • Reply • Share ›

Guy Cheung • a year ago

Great article for me.

• Reply • Share ›

Beney Kim • a year ago

Great article for me. Is there any other article explaining how linux network system work?

• Reply • Share ›

Sergii Mikhtoniuk • 3 years ago

Thanks for a great article, it helped me a lot.

Is it just me, or the new implementation using two queues completely breaks the backpressure?

Imagine your service is under heavy load and cannot keep up with incoming requests. For me the most natural thing to do in this case is to start refusing some of the connections, allowing your load balancer to redispatch the requests to another server in the pool.

Implementation with two queues essentially takes away this ability from applications. Kernel SYN-ACKs all connections for you unconditionally, even if your listen backlog is already full.

The way tcp_abort_on_overflow option works makes no sense to me. When it's on, kernel resets the connection only _after_ it moves to ESTABLISHED state on the client side. So by the time RST is sent client may have already started transmitting the data. I think this is a huge problem. Load balancers such as HAProxy do not even allow redispatching requests to another server if connection to the first server was successfully established.

With tcp_abort_on_overflow option off the situation is even worse. Unconditional SYN-ACK means that client almost never will hit the connect timeout. So all the time spent by connection waiting for a free slot in listen backlog while kernel just ignores it is accounted as "server timeout", which can be really high depending on application.

Tuning tcp_max_syn_backlog to achieve this kind of behaviour doesn't seem like a good idea.

• Reply • Share ›

Andreas Veithen Mod ➚ Sergii Mikhtoniuk • 3 years ago

I think you are basically right, but I'm not sure if in practice the scenario you describe is actually relevant. The reason is that in order to gracefully handle servers

The reason is that in order to gracefully handle servers that become unhealthy or overloaded you would probably use a load balancing algorithm such as least outstanding requests. In that case the desired back pressure would actually build up, even if the target system doesn't start rejecting connections. WDYT?

 • Reply • Share ›

Sergii Mikhtoniuk ➚ Andreas Veithen
• 3 years ago

Thanks Andreas, my case is probably too specific. We use SmartStack service discovery so there's a bunch of HAProxies running locally on every service host and routing the requests. Unlike conventional reverse proxy setup there's no centralized place to know how many connections are currently established to a backend server. With mixed and irregular workload coming from frontend services it's hard to define a reasonable outbound connection limits for frontend hosts without risking either underutilizing the backends or opening too many and having latency problems because of excessive queuing. I have not found a better approach to avoid all these problems other than active back pressure from backend hosts.

My overall conclusion was that the tcp_abort_on_overflow option is broken and there