

httpbis
Internet-Draft
Intended status: Best Current Practice
Expires: June 23, 2016

D. Stenberg
Mozilla
December 21, 2015

TCP Tuning for HTTP
draft-stenberg-httpbis-tcp-01

Abstract

This document records current best practice for using all versions of HTTP over TCP.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 23, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	2
1.1.	Notational Conventions	3
2.	Socket planning	3
2.1.	Number of open files	3
2.2.	Number of concurrent network messages	3
2.3.	Number of incoming TCP SYNs allowed to backlog	4
2.4.	Use the whole port range for local ports	4
2.5.	Lower the TCP FIN timeout	4
2.6.	Reuse sockets in TIME_WAIT state	4
2.7.	TCP socket buffer sizes and Window Scaling	4
2.8.	Set maximum allowed TCP window sizes	5
2.9.	Timers and timeouts	5
3.	TCP handshake	6
3.1.	TCP Fast Open	6
3.2.	Initial Congestion Window	6
3.3.	TCP SYN flood handling	6
4.	TCP transfers	7
4.1.	Packet Pacing	7
4.2.	Explicit Congestion Control	7
4.3.	Nagle's Algorithm	7
4.4.	Keep-alive	7
5.	Re-using connections	7
5.1.	Slow Start after Idle	7
5.2.	TCP-Bound Authentications	8
6.	Closing connections	8
6.1.	Half-close	8
6.2.	Abort	8
6.3.	Close Idle Connections	8
6.4.	Tail Loss Probes	8
7.	IANA Considerations	9
8.	Security Considerations	9
9.	References	9
9.1.	Normative References	9
9.2.	Informative References	9
9.3.	URIs	10
Appendix A.	Acknowledgments	10
	Author's Address	10

[1.](#) Introduction

HTTP version 1.1 [[RFC7230](#)] as well as HTTP version 2 [[RFC7540](#)] are defined to use TCP [[RFC0793](#)], and their performance can depend greatly upon how TCP is configured. This document records the best current practice for using HTTP over TCP, with a focus on improving end-user perceived performance.

These practices are generally applicable to HTTP/1 as well as HTTP/2, although some may note particular impact or nuance regarding a particular protocol version.

There are countless scenarios, roles and setups where HTTP is being using so there can be no single specific "Right Answer" to most TCP questions. This document intends only to cover the most important areas of concern and suggest possible actions.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC2119](#)].

2. Socket planning

Your HTTP server or intermediary may need configuration changes to some system tunables and timeout periods to perform optimally. Actual values will depend on how you are scaling the platform, horizontally or vertically, and other connection semantics. Changing system limits and altering thresholds will change the behavior of your web service and its dependencies. These dependencies are usually common to other services running on the same system, so good planning and testing is advised.

This is a list of values to consider and some general advice on how those values can be modified on Linux systems.

2.1. Number of open files

A modern HTTP server will serve a large number of TCP connections and in most systems each open socket equals an open file. Make sure that limit isn't a bottle neck. In Linux, the limit can be raised like this:

```
fs.file-max = <number of files>
```

2.2. Number of concurrent network messages

Raise the number of packets allowed to get queued when a particular interface receives packets faster than the kernel can process them. In Linux, this limit can be raised like this:

```
net.core.netdev_max_backlog = <number of packets>
```


2.3. Number of incoming TCP SYNs allowed to backlog

The number of new connection requests that are allowed to queue up in the kernel. In Linux, this limit can be raised like this:

```
net.core.somaxconn = <number>
```

2.4. Use the whole port range for local ports

To make sure the TCP stack can take full advantage of the entire set of possible sockets, give it a larger range of local port numbers to use.

```
net.ipv4.ip_local_port_range = 1024 65535
```

2.5. Lower the TCP FIN timeout

High connection completion rates will consume ephemeral ports quickly. Lower the time during which connections are in FIN-WAIT-2/TIME_WAIT states so that they can be purged faster and thus maintain a maximal number of available sockets. The primitives for the assignment of these values were described in [[RFC0793](#)], however significantly lower values are commonly used.

```
net.ipv4.tcp_fin_timeout = <number of seconds>
```

2.6. Reuse sockets in TIME_WAIT state

When running backend servers on a managed, low latency network you might allow the reuse of sockets in TIME_WAIT state for new connections when a protocol complete termination has occurred. There is no RFC that covers this behaviour.

```
net.ipv4.tcp_tw_reuse = 1
```

2.7. TCP socket buffer sizes and Window Scaling

Systems meant to handle and serve a huge number of TCP connections at high speeds need a significant amount of memory for TCP socket buffers. On some systems you can tell the TCP stack what default buffer sizes to use and how much they are allowed to dynamically grow and shrink. Window Scaling is typically linked to socket buffer sizes and on a Linux system can be controlled with the values:

```
net.ipv4.tcp_wmem = <minimum size> <default size> <max size in bytes>  
net.ipv4.tcp_rmem = <minimum size> <default size> <max size in bytes>
```


The minimum and default tend to require less proactive amendment than the maximum value. When deriving maximum values for use, you should consider the BDP (Bandwidth Delay Product) of the target environment and clients. Consider also that 'read' and 'write' values do not require to be synchronised, as the BDP requirements for a load balancer or middle-box might be very different when acting as a sender or receiver.

Allowing needlessly high values beyond the expected limitations of the platform might increase the probability of retransmissions and buffer induced delays within the path. Extensions such as ECN coupled with AQM can help mitigate this undesirable behaviour [[RFC7141](#)].

[RFC7323] covers Window Scaling in greater detail.

2.8. Set maximum allowed TCP window sizes

You may have to increase the largest allowed window size. Window scaling must be accommodated within the maximal values, however it is not uncommon to see the maximum definable higher than the scalable limit; these values can statically defined within socket parameters (SO_RCVBUF, SO_SNDBUF)

```
net.core.rmem_max = <number of bytes>
net.core.wmem_max = <number of bytes>
```

2.9. Timers and timeouts

On a modern shared platform it can be common to plan for both long and short lived connections on the same implementation. However, the delivery of static assets and a 'web push' or 'long poll' service provide very different quality of service promises.

Fail 'fast': TCP resources can be highly contended. For fault tolerance reasons a server needs to be able to determine within a reasonable time frame whether a connection is still active or required. e.g. If static assets typically return in 100s of milliseconds, and users 'switch off' after <10s keeping timeouts of >30s make little sense and defining a 'quality of service' appropriate to the target platform is encouraged. On a shared platform with mixed session lifetimes, applications that require longer render times have various options to ensure the underlying service and upstream servers in the path can identify the session as not failed: HTTP continuations, Redirects, 202s or sending data.

Clients and servers typically have many timeout options, a few notable options are: Connect(client), time to request(server), time

to first byte(client), between bytes(server/client), total connection time(server/client). Some implementations merge these values into a single 'timeout' definition even when statistics are reported individually. All should be considered as the defaults in many implementations are highly underivable, even infinite timeouts have been observed.

3. TCP handshake

3.1. TCP Fast Open

TCP Fast Open (a.k.a. TFO, [[RFC7413](#)]) allows data to be sent on the TCP handshake, thereby allowing a request to be sent without any delay if a connection is not open.

TFO requires both client and server support, and additionally requires application knowledge, because the data sent on the SYN needs to be idempotent. Therefore, TFO can only be used on idempotent, safe HTTP methods (e.g., GET and HEAD), or with intervening negotiation (e.g., using TLS). It should be noted that TFO requires a secret to be defined on the server to mitigate security vulnerabilities it introduces. TFO therefore requires more server side deployment planning than other enhancements.

Support for TFO is growing in client platforms, especially mobile, due to the significant performance advantage it gives.

3.2. Initial Congestion Window

[RFC6928] specifies an initcwnd (initial congestion window) of 10, and is now fairly widely deployed server-side. There has been experimentation with larger initial windows, in combination with packet pacing. Many implementations allow initcwnd to be applied to specific routes which allows a greater degree of flexibility than some other TCP parameters.

IW10 has been reported to perform fairly well even in high volume servers.

3.3. TCP SYN flood handling

TCP SYN Flood mitigations [[RFC4987](#)] are necessary and there will be thresholds to tweak.

[4.](#) TCP transfers

[4.1.](#) Packet Pacing

TBD

[4.2.](#) Explicit Congestion Control

Apple deploying in iOS and OSX [[1](#)].

[4.3.](#) Nagle's Algorithm

Nagle's Algorithm [[RFC0896](#)] is the mechanism that makes the TCP stack hold (small) outgoing packets for a short period of time so that it can potentially merge that packet with the next outgoing one. It is optimized for throughput at the expense of latency.

HTTP/2 in particular requires that the client can send a packet back fast even during transfers that are perceived as single direction transfers. Even small delays in those sends can cause a significant performance loss.

HTTP/1.1 is also affected, especially when sending off a full request in a single `write()` system call.

In POSIX systems you switch it off like this:

```
int one = 1;
setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &one, sizeof(one));
```

[4.4.](#) Keep-alive

TCP keep-alive is likely disabled - at least on mobile clients for energy saving purposes. App-level keep-alive is then required for long-lived requests to detect failed peers or connections reset by stateful firewalls etc.

[5.](#) Re-using connections

[5.1.](#) Slow Start after Idle

Slow-start is one of the algorithms that TCP uses to control congestion inside the network. It is also known as the exponential growth phase. Each TCP connection will start off in slow-start but will also go back to slow-start after a certain amount of idle time.

In Linux systems you can prevent the TCP stack from going back to slow-start after idle by setting


```
net.ipv4.tcp_slow_start_after_idle = 0
```

5.2. TCP-Bound Authentications

There are several HTTP authentication mechanisms in use today that are used or can be used to authenticate a connection rather than a single HTTP request. Two popular ones are NTLM and Negotiate.

If such an authentication has been negotiated on a TCP connection, that connection can remain authenticated throughout the rest of its lifetime. This discrepancy with how other HTTP authentications work makes it important to handle these connections with care.

6. Closing connections

6.1. Half-close

The client or server is free to half-close after a request or response has been completed; or when there is no pending stream in HTTP/2.

Half-closing is sometimes the only way for a server to make sure it closes down connections cleanly so that it doesn't accept more requests while still allowing clients to receive the ongoing responses.

6.2. Abort

No client abort for HTTP/1.1 after the request body has been sent. Delayed full close is expected following an error response to avoid RST on the client.

6.3. Close Idle Connections

Keeping open connections around for subsequent connection reuse is key for many HTTP clients' performance. The value of an existing connection quickly degrades and after only a few minutes the chance that a connection will successfully get reused by a web browser is slim.

6.4. Tail Loss Probes

draft [2]

7. IANA Considerations

This document does not require action from IANA.

8. Security Considerations

TBD

9. References

9.1. Normative References

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, [RFC 793](#), DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", [RFC 7230](#), DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", [RFC 7540](#), DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.

9.2. Informative References

- [RFC0896] Nagle, J., "Congestion Control in IP/TCP Internetworks", [RFC 896](#), DOI 10.17487/RFC0896, January 1984, <<http://www.rfc-editor.org/info/rfc896>>.
- [RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", [RFC 4987](#), DOI 10.17487/RFC4987, August 2007, <<http://www.rfc-editor.org/info/rfc4987>>.
- [RFC6928] Chu, J., Dukkkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", [RFC 6928](#), DOI 10.17487/RFC6928, April 2013, <<http://www.rfc-editor.org/info/rfc6928>>.

- [RFC7141] Briscoe, B. and J. Manner, "Byte and Packet Congestion Notification", [BCP 41](#), [RFC 7141](#), DOI 10.17487/RFC7141, February 2014, <<http://www.rfc-editor.org/info/rfc7141>>.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", [RFC 7323](#), DOI 10.17487/RFC7323, September 2014, <<http://www.rfc-editor.org/info/rfc7323>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", [RFC 7413](#), DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.

[9.3. URIs](#)

- [1] <https://developer.apple.com/videos/wwdc/2015/?id=719>
- [2] <http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01>

[Appendix A. Acknowledgments](#)

This specification builds upon previous work and help from Mark Nottingham, Craig Taylor

Author's Address

Daniel Stenberg
Mozilla

Email: daniel@haxx.se
URI: <http://daniel.haxx.se>

