Running a high-performance pluggable transports Tor bridge

Draft March 7, 2023

David Fifield Linus Nordberg

Abstract

The pluggable transports model in Tor separates the concerns of anonymity and circumvention by running circumvention code in a separate process, which exchanges information with the main Tor process using local interprocess communication. This model leads to problems with scaling, especially for transports, like meek and Snowflake, that do not rely on the existence of numerous, independently administered bridges for blocking resistance, but rather forward all traffic to one or a few centralized bridges. We identify what bottlenecks arise as a bridge scales from 500 to 10,000 simultaneous users, and then from 10,000 to 50,000, and show how to overcome them, based on our experience running a bridge for Snowflake. The key idea is running multiple Tor processes in parallel on the bridge host, with synchronized identity keys.

1 Introduction

Bridges and pluggable transports are how Tor adds blocking resistance (censorship circumvention) to its core function of anonymity. Bridges are relays whose network addresses are not globally known, meant to be difficult for a censor to discover and block by address. Pluggable transports are modular tunneling protocols that encapsulate and disguise an inner protocol, thereby preventing a censor from recognizing the Tor protocol and blocking connections on that basis.

Tor's original blocking resistance design called for a large number of "bridge" relays [2 §5], which clients would connect to directly, using the ordinary TLS-based Tor protocol. The difference between ordinary relays and bridges is that the network addresses of bridges are not made public in the Tor consensus, but rather are distributed one at a time, in some controlled fashion. The blocking resistance of this model depends on keeping bridge addresses secret, because there is nothing to stop a censor from blocking a bridge by its address, once known. When pluggable transports arrived on the scene, many adopted the same strategy with respect to address blocking resistance. obfs2, obfs3, FTE, ScrambleSuit, obfs4—all these change the protocol between client and bridge, but they retain the model of clients making TCP connections to fixed bridge IP addresses that must be kept secret. Because the blocking resistance of this model depends on the existence of a large pool of bridges, it naturally achieves "horizontal" scaling, with user traffic being distributed over hundreds of independently operated hosts in different networks.

But other pluggable transports are not based on a model of secret bridge addresses: meek and Snowflake are currently deployed pluggable transports whose resistance to address-based blocking comes about in a different way. In these transports, the host that is the gateway to the Tor network (the "bridge" proper) is decoupled from the means of accessing it, which is rather via some intermediary (a CDN in meek; a temporary browser proxy in Snowflake). Transports like these do not benefit, in terms of blocking resistance, from having a large number of bridges, and it is therefore convenient to run just one, centralized bridge—whose address need not be secret—to receive all the transport's traffic. This, however, requires attention to the "vertical" scaling of the bridge.

In this paper, we show how to do this vertical scaling of a pluggable transports Tor bridge. The key technique is to run multiple Tor processes on the same host with the same identity keys. This alleviates the largest single bottleneck, namely that of any single Tor process being CPU-bound, but also gives rise to a few complications. Beyond that, there are other resource constraints to consider, such as limits on file descriptors and ephemeral port numbers. The recommendations come from our experience running the Snowflake bridge from December 2021 to February 2023, during which time the average number of simultaneous users grew from 2,000 to around 100,000.

2 Background on pluggable transports

Tor's Pluggable Transports specification [10] describes how Tor interacts with its pluggable transports. It is built around a model of separate processes and interprocess communication. The Tor process spawns a child process, giving it its configuration in the form of environment variables; the pluggable transport process reports status on its standard output stream; and thereafter user traffic is carried over localhost TCP connections.[1] Refer to Figure 1. Client and server transports work similarly, but only the server side concerns us here.

Pluggable transports are enabled in Tor's configuration file, torrc. A sample configuration for a transport called "mypt" looks like this:

```
ServerTransportPlugin mypt exec /usr/local/bin/mypt
ServerTransportListenAddr mypt 0.0.0.0:1234
ExtORPort auto
```

When Tor starts a pluggable transport, it passes configuration information to the subprocess in environment variables. The above torrc causes Tor to execute /usr/local/bin/mypt as a subprocess, with the following environment variables (*eph* is a random ephemeral port):

```
TOR_PT_SERVER_TRANSPORTS=mypt
TOR_PT_SERVER_BINDADDR=mypt-[::]:1234
TOR_PT_EXTENDED_SERVER_PORT=127.0.0.1:eph
TOR_PT_AUTH_COOKIE_FILE=
↪ /var/lib/tor/extended_orport_auth_cookie
```

`TOR_PT_SERVER_TRANSPORTS` tells the pluggable transport what named transports to start (because one executable may support multiple transports). `TOR_PT_SERVER_BINDADDR` is the address at which the pluggable transport should listen for incoming connections (if such a notion makes sense for the transport). `TOR_PT_EXTENDED_SERVER_PORT` is the TCP address of Tor's Extended ORPort [6], the interface between the pluggable transport and the ordinary Tor network. The pluggable transport process receives connections from the Internet, removes the obfuscation layer, and forwards the tunneled stream to the Extended ORPort of the Tor process. The Extended ORPort supports a meta-protocol to tag incoming connections with client IP address and transport names, which is what Tor Metrics uses to break down metrics by country and transport. `TOR_PT_AUTH_COOKIE_FILE` is the path to a file containing an authentication secret that is required when connecting to the Extended ORPort—synchronizing this secret across multiple instances of Tor will be one of the complications to deal with in the next section.

3 Multiple Tor processes

The first and most important bottleneck to overcome is the single-threaded nature of the Tor implementation.[2] A single Tor process is limited to one CPU core: once Tor hits 100% CPU, the performance of the bridge is capped, no matter the speed of the network connection or the number of CPU cores to spare. For us, this started to be a problem at around 6,000 simultaneous users and 10 MB/s of Tor bandwidth.

Fact-check footnote claim about Arti

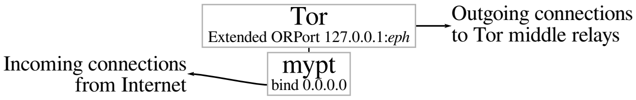

Figure 1: The normal way of running a server pluggable transport. The init system spawns a Tor process, which in turn spawns a pluggable transport process. This model reaches a performance plateau when the Tor process saturates one CPU.

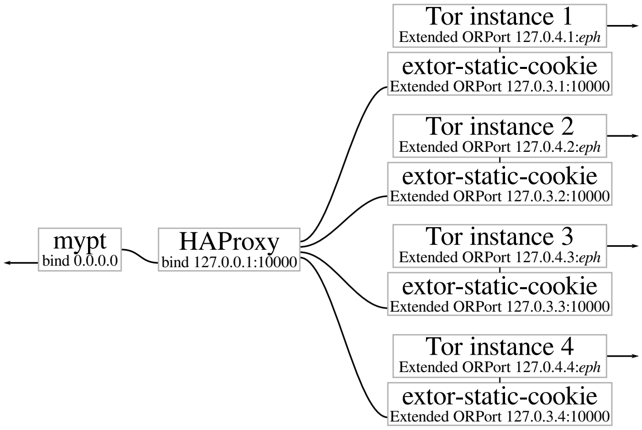

Figure 2: The multi-Tor configuration, which permits more scaling. The init system spawns multiple independent Tor processes and a load balancer to distribute traffic across them. The pluggable transport process is no longer a subprocess of a Tor process, but is spawned by the init system directly, and communicates upstream with the load balancer, which makes the Tor instances' several Extended ORPorts appear as one. Each Tor process spawns an extor-static-cookie process (in the manner of Figure 1), in order to present a predictable Extended ORPort authentication secret through the load balancer.

Our solution is to run multiple Tor processes concurrently, and mediate the pluggable transport's access to them with a load balancer. (We use HAProxy, though any load balancer will do.) Refer to Figure 2. As many Tor processes can be run as are needed to distribute CPU load; we started with 4 and currently use 12. The several Tor instances are independent, in the sense that they do not communicate with one another; but they all share the same long-term identity keys, so they are equally capable of serving as the first hop in a Tor circuit for a client that expects a certain relay fingerprint. The pluggable transport server receives incoming connections, as before, but instead of forwarding connections to the Extended ORPort of a single Tor process, it forwards them to the load balancer, which then chooses one of the many Tor processes. The extor-static-cookie component paired to each Tor process resolves a complication that will be explained in Section 3.1.

---

[1] The Pluggable Transports 2.x and 3.x specifications [8], which descend from Tor's version 1 specification, define an "API" interface for linking pluggable transports directly into an application, in addition to a Tor-like "IPC" interface. Tor does not use or support these later specifications.

[2] We expect that Arti, the in-progress reimplementation of Tor, will be natively multi-threaded, and remove this primary complication.

This multi-instance arrangement requires subverting the usual pluggable transports subprocess model of Section 2. Normally, the operating system's init system (e.g. systemd) starts Tor, and Tor starts the pluggable transport server. Here, we have the init system start the pluggable transport server, the load balancer, and all the instances of Tor as sibling processes. We set up an environment of the pluggable transport server to appear as if it had been started in the normal way, but we make `TOR_PT_EXTENDED_SERVER_PORT` point at the load balancer, rather than any particular instance of Tor. (See Appendix A for sample systemd and other configuration files.)

On Debian or Ubuntu, the tor-instance-create utility [7] is a convenient way to create and manage multiple instances of Tor with independent configurations:

```
tor-instance-create mypt1
tor-instance-create mypt2
tor-instance-create mypt3
tor-instance-create mypt4
```

Each instance exposes an Extended ORPort interface on a distinct, static localhost address. The Extended ORPort addresses are then listed in the load balancer configuration file.

To make the instances all have the same identity keys, start and stop one of them to make it generate keys for itself:

```
systemctl start tor@mypt1
systemctl stop tor@mypt1
```

Then copy that instance's "keys" subdirectory into the data directory of the other instances, fixing permissions as necessary. This operation causes all the instances to be interchangeable, in terms of being able to build circuits under the shared bridge fingerprint.

With multiple instances of Tor created and their identity keys replicated, there are just a few more details to look after.

3.1 Extended ORPort authentication

The Extended ORPort protocol begins with an obligatory authenticated handshake. Both Tor, and the pluggable transport that connects to it, cryptographically verify that the other has access to a secret "cookie" stored in a file [6 §2.1]. Tor regenerates the cookie file every time it is restarted, and shares the path to the file with the pluggable transport in the `TOR_PT_AUTH_COOKIE_FILE` environment variable. This poses a problem for the multi-instance Tor setup. Since every instance generates its cookie file independently, and the pluggable transport cannot predict which Tor instance it will be connected to through the load balancer, it does not know which Extended ORPort authentication secret to use.

Tor does not expose a configuration option to control or disable the regeneration of authentication cookie files. We need a way to expose an Extended ORPort interface with a uniform authentication secret across all Tor instances. To do this, we interpose an adapter, called extor-static-cookie [5], between the load balancer and the Extended ORPort of each of the Tor processes. The adapter acts as an Extended ORPort *client* towards its parent Tor process, and an Extended ORPort *server* towards the load balancer (and the server pluggable transport in turn). As a client, extor-static-cookie communicates with its respective instance of Tor using the authentication cookie specific to that instance. As a server, it uses a static authentication cookie file that is also provided to the server pluggable transport in its `TOR_PT_AUTH_COOKIE_FILE` environment variable. Each Tor process spawns a copy of extor-static-cookie using the normal pluggable transports subprocess machinery of Section 2.

It would be nice if there were a way to disable Extended ORPort authentication cookie regeneration in Tor, or if the Extended ORPort offered an alternative authentication type, one that is easier to coordinate over multiple instances. Apart from its cookie juggling, extor-static-cookie does nothing but add overhead to the communications pipeline. We did consider a few alternative solutions to the problem of Extended ORPort authentication. It would be easy to patch Tor to use a hardcoded cookie, say, but maintaining a forked version of Tor complicates the deployment of security upgrades, which we deemed unacceptably risky. In place of the Extended ORPort, it is possible to use the regular, non-extended ORPort. The regular ORPort does not have an authenticated handshake, but it also does not provide a way to tag connections with the client's IP address, which would mean the loss of country-specific metrics.

3.2 Disabling onion key rotation

Besides its identity key, which never changes, a Tor relay or bridge has medium-term onion keys that are used during circuit construction [3 §4]. Onion keys are rotated on a fixed schedule (every 28 days, as of this writing). A relay's current onion keys appear in the Tor network consensus; when clients make circuits through it, they expect it to use certain onion keys. Immediately after being created, our multiple Tor instances have identical onion keys, because of the copying operation described above. But without further arrangements, the instances would eventually independently rotate their onion keys, which would cause later circuit creation attempts to fail.

To prevent this divergence, we disable onion key rotation. Tor does not expose a configuration option to control this, so we resort to external means. We create preexisting directories at the filesystem paths that Tor uses as the destination of file rename operations during key rotation, secret_onion_key.old and secret_onion_key_ntor.old. (Preexisting *files* are not good enough; they must be directories to stop the rename operation from succeeding.) Tor will log an error every time thereafter it tries and fails to rotate its onion keys, but will otherwise continue running. The hack is effective, but it would be better if there were a supported way to do this in Tor.

4 Further bottlenecks

Distributing Tor processing over many CPU cores is the essential step to enable scaling. As the number of users increases, the bridge will begin to bump into other, less restrictive limitations.

File descriptor limits. Every open socket consumes a file descriptor. Because the pluggable transports model uses sockets not only for external connections but also for interprocess communications, and the number of sockets is proportional to the number of users, it is easy to exceed the operating system's default limit on the number of file descriptors, which manifests in error messages like "too many open files." Tor and HAProxy automatically override the defaults and set sufficiently high limits for themselves, but for the server pluggable transport process you can use, for example, `LimitNOFILE` in a systemd service file to raise the limit (see Section A.2). For us, a limit of 64 thousand was insufficient, but we have not had problems since raising the limit to 1 million.

Ephemeral TCP ports. TCP sockets are distinguished from one another by a four-tuple consisting of the source and destination IP addresses and the source and destination port numbers. When connecting a socket, the operating system chooses a port number from the ephemeral port range to serve as the socket's source port. If all ephemeral ports are already in use, such that the socket's four-tuple would not be unique, the connection fails with an error like "cannot assign requested address."

The baseline mitigation for ephemeral port exhaustion is expanding the range of ephemeral ports. This is common advice for operators of all Tor relays. On Linux, it looks something like this:

```
sysctl -w net.ipv4.ip_local_port_range="15000 64000"
```

But this alone is not enough for a pluggable transports bridge. The bridge's outgoing connections to other Tor relays are not the main problem—the same source port can be used in many sockets, as long as the destination addresses are different. The real crunch comes from the bridge's many local TCP connections, the internal links in Figure 2. If source and destination IP addresses are both 127.0.0.1, source and destination port numbers are all that remain to make TCP sockets distinct.

Part of the solution is using different localhost IP addresses for different listening sockets. Not only 127.0.0.1, but the entire 127.0.0.0/8 range is dedicated to localhost.We use 127.0.0.1 for HAProxy, but 127.0.3.*N* for the *N*th instance of extor-static-cookie, and 127.0.4.*N* for the *N*th Tor instance's Extended ORPort. But this still is not enough; we must also diversify source addresses. In HAProxy we use the `source` option to use 127.0.2.*N* as the source address when connecting to the *N*th instance of extor-static-cookie (see Section A.3); this expands the number of possible simultaneous connections by a factor equal to the number of Tor instances. For the bottleneck connection between the pluggable transport and the load balancer, we added a custom option `orport-srcaddr` to the pluggable transport (Section A.2); that instructs it to choose a random source address in the range 127.0.1.0/24 when connecting to the load balancer, which increases the number of usable source addresses by a factor of 256. The extor-static-cookie adapter also supports the `orport-srcaddr` option (Section A.1), though it is less necessary there, because each instance of extor-static-cookie sees only a fraction of total connections.

Firewall connection tracking. The connection tracking (conntrack) feature of the Linux firewall has a default limit of 262,144 connections in Linux 5.15. When the number of connections reaches that limit, new connections will be dropped. Our experiments showed that the number of connections was getting close to the limit during the busiest times of day, so we doubled the size of the connection tracking table:

```
sysctl -w net.netfilter.nf_conntrack_max=524288
sysctl -w net.netfilter.nf_conntrack_buckets=524288
```

5 Discussion

The multiple-Tor architecture of Section 3 may be useful also for non–pluggable transport relays, such as large exit relays that process many connections, though we hope that in the future Tor or Arti will be able to more naturally scale across CPU cores, and make the multiple-process workaround obsolete.

The pluggable transports model of interprocess communication over TCP sockets is suitable for clients and low- to medium-use servers, but it starts to become awkward for high-use servers, primarily because of ephemeral port exhaustion and the required workarounds. Future designs should consider alternatives, such as Unix domain sockets, or the in-process API of later pluggable transports specifications [8].

Acknowledgements

The basic architecture described in this paper was worked out in a thread on the tor-relays mailing list [4]. We thank Roger Dingledine for confirming that running multiple instances of Tor with synchronized identity keys would be feasible, suggesting a similarity with the "router twins" [1] concept from the early history of Tor, and anticipating the problem of onion key rotation. Silvia Puglisi and Georg Koppen enhanced Tor Metrics to be aware of relays that publish multiple independent descriptors for the same relay fingerprint [9]. Greenhost provided hosting for the Snowflake bridge during the initial transition to the load-balanced configuration. We thank donors and financial supporters: particularly the Open Technology Fund, for a short-term grant to support operational costs when we moved the bridge to a dedicated server; Mullvad VPN, for a donation of hardware for the new server; and OBE.NET, for providing hardware colocation and at-cost bandwidth.

Availability

Step-by-step instructions for installing a load-balanced bridge configuration for Snowflake are available in the Tor anti-censorship team's Snowflake Bridge Installation Guide: https://gitlab.torproject.org/tpo/anti-censorship/team/-/wikis/Survival%20Guides/Snowflake%20Bridge%20Installation%20Guide. The wiki page version as of this writing is a7bcf050 .

Update before submitting

The extor-static-cookie program is available at https://gitlab.torproject.org/dcf/extor-static-cookie.

Move to permanent URL?

References

[1] Roger Dingledine. Router twins. tor-dev mailing list, July 2002. URL: https://lists.torproject.org/pipermail/tor-dev/2002-July/001122.html.

[2] Roger Dingledine and Nick Mathewson. Design of a blocking-resistant anonymity system. Technical Report 2006-11-001, The Tor Project, November 2006. URL: https://research.torproject.org/techreports/blocking-2006-11.pdf.

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*. USENIX, August 2004. URL: https://spec.torproject.org/tor-design.

[4] David Fifield. How to reduce tor CPU load on a single bridge? tor-relays mailing list, December 2021. URL: https://forum.torproject.net/t/1483.

[5] David Fifield. extor-static-cookie, December 2022. URL: https://gitlab.torproject.org/dcf/extor-static-cookie.

Move to permanent URL?

[6] George Kadianakis and Nick Mathewson. Extended ORPort for pluggable transports, October 2021. URL: https://spec.torproject.org/ext-orport-spec.

[7] Peter Palfrader. *tor-instance-create*. Debian, August 2021. URL: https://manpages.debian.org/bullseye/tor/tor-instance-create.8.en.html.

[8] Pluggable Transport Steering Committee. Pluggable transports spec v3.0, January 2022. URL: https://github.com/Pluggable-Transports/Pluggable-Transports-spec/tree/main/releases/PTSpecV3.0.

[9] Silvia Puglisi. userstats-bridge-country graph is undercounting users. URL: https://bugs.torproject.org/tpo/network-health/metrics/website/40047.

[10] The Tor Project. Pluggable transport specification (version 1), February 2022. URL: https://spec.torproject.org/pt-spec.

A Configuration files

A.1 Per-instance torrc files

This is a template for the per-instance torrc configuration file. If using tor-instance-create [7] with instance name prefix “mypt”, the file for instance *N* will be at the path /etc/tor/instances/mypt*N*/torrc. Replace the highlighted text with appropriate values. `NICKNAME` and `EMAIL` will be the same for all instances; `N` is the instance number. It is important to give every instance a distinct nickname, because that is how Tor Metrics disambiguates multiple descriptors with the same relay fingerprint.

`ORPort` likely needs to be exposed for ordinary bridges

The `orport-srcaddr` transport option is not a universal standard; it is a special feature in extor-static-cookie to avoid ephemeral port exhaustion as described in Section 4. When connecting to the Tor process’s Extended ORPort, extor-static-cookie will choose a random source IP address in the range 127.0.5.0/24.

```
BridgeRelay 1
AssumeReachable 1
BridgeDistribution none
ORPort 127.0.0.1:auto # unused
ExtORPort 127.0.4.N:auto
SocksPort 0
ServerTransportPlugin mypt exec extor-static-cookie /etc/extor-static-cookie/static_extended_orport_auth_cookie
ServerTransportListenAddr mypt 127.0.3.N:10000
ServerTransportOptions mypt orport-srcaddr=127.0.5.0/24
Nickname NICKNAME N
ContactInfo EMAIL
```

A.2 Pluggable transport systemd service file

Install this file as /etc/systemd/system/mypt.service. You can then enable it with `systemctl enable mypt` and start it running with `systemctl start mypt`. The service file assumes the existence of a user called “mypt” (`adduser --system mypt`).

The `Environment` variables set up a simulated pluggable transports environment, with `TOR_PT_EXTENDED_SERVER_PORT` pointing at the load balancer. `PORT` is the pluggable transport server’s external listening port. As with extor-static-cookie, the `orport-srcaddr` transport option whose purpose is to conserve ephemeral ports is something special we have implemented, not a part of pluggable transports or any other standard.

Avoid page break in file listing

```
[Unit]
Description=DESCRIPTION

[Service]
Type=exec
Restart=on-failure
User=mypt
StateDirectory=mypt
LogsDirectory=mypt

# Use CAP_NET_BIND_SERVICE if the server needs to bind to a privileged port.
AmbientCapabilities=CAP_NET_BIND_SERVICE
NoNewPrivileges=true
ProtectHome=true
ProtectSystem=strict
PrivateTmp=true
PrivateDevices=true
ProtectClock=true
ProtectKernelModules=true
ProtectKernelLogs=true
LimitNOFILE=1048576

Environment=TOR_PT_MANAGED_TRANSPORT_VER=1
Environment=TOR_PT_SERVER_TRANSPORTS=mypt
Environment=TOR_PT_SERVER_BINDADDR=mypt-[::]:PORT
Environment=TOR_PT_EXTENDED_SERVER_PORT=127.0.0.1:10000
Environment=TOR_PT_AUTH_COOKIE_FILE=/etc/extor-static-cookie/static_extended_orport_auth_cookie
Environment=TOR_PT_SERVER_TRANSPORT_OPTIONS=mypt:orport-srcaddr=127.0.1.0/24
Environment=TOR_PT_STATE_LOCATION=%S/mypt/pt_state
Environment=TOR_PT_EXIT_ON_STDIN_CLOSE=0
```

```
ExecStart=/usr/local/bin/mypt

[Install]
WantedBy=multi-user.target
```

A.3 HAProxy configuration file

The below configuration defines a frontend listener at 127.0.0.1:10000, which forwards to a backend consisting of the multiple Tor instances (actually their extor-static-cookie adapters). Each backend connection uses a different source IP address, to conserve ephemeral ports. There is no need for any kind of backend affinity based on source address or any other feature; simple round-robin balancing is sufficient. This configuration should be added to any defaults already present in /etc/haproxy/haproxy.cfg.

`source` port range may not be necessary with tor 0.4.7.13+

```
frontend tor
        mode tcp
        bind 127.0.0.1:10000
        default_backend tor-instances
        option dontlog-normal
        timeout client 600s
backend tor-instances
        mode tcp
        timeout server 600s
        server mypt1 127.0.3.1:10000 source 127.0.2.1:15000-64000
        server mypt1 127.0.3.2:10000 source 127.0.2.2:15000-64000
        server mypt1 127.0.3.3:10000 source 127.0.2.3:15000-64000
        server mypt1 127.0.3.4:10000 source 127.0.2.4:15000-64000
```