

Unix domain sockets



Tom Ryder

tom@sanctum.geek.nz

<https://sanctum.geek.nz/>

Proxies abound—1/3

- Over the last 10 years or so, a typical way of building internet servers has developed.
- One **reverse proxy** listening on a WAN address...
- ...securing and directing requests to **internal HTTP services** listening on IPv4 localhost (127.0.0.1).



Proxies abound—2/3

- The reverse proxy manages the **trust**:
 - HTTPS...
 - ACLs...
 - Traffic management...
 - All the stuff you need on the hostile internet.
- The internal services *trust* the traffic they're passed.
- They are not reachable from the internet.



Proxies abound—3/3

- Nginx listening on the internet: **203.0.113.1:443**
- Securing and routing requests to:
 - **127.0.0.1:3000**—Grafana
 - **127.0.0.1:8096**—Jellyfin
 - **127.0.0.1:6379**—NetBox
 - **127.0.0.1:9090**—Prometheus



Any port in a storm—1/6

- But why those ports? Why not port 80 (HTTP)?
- There's no process.
- The developers *just pick one*.
- Sometimes there's a conflict...



Any port in a storm—2/6

- Numbers are seldom standardized in `/etc/services`.
- The service needs to run as root (or have special capabilities) to listen on a port number less than 1024.
- Only one service can listen on a port.



Any port in a storm—3/6

One way to get around this addressing issue is to use multiple IPv4 local listeners, like this:

- **127.0.0.2:80**—Grafana
- **127.0.0.3:80**—Jellyfin
- **127.0.0.4:80**—NetBox
- **127.0.0.5:80**—Prometheus



Any port in a storm—4/6

You can even give them cute names in `/etc/hosts...`

`grafana.internal.` ↔ `127.0.0.2`

`jellyfin.internal.` ↔ `127.0.0.3`

`netbox.internal.` ↔ `127.0.0.4`

`prometheus.internal.` ↔ `127.0.0.5`

(Yes, `*.internal.` is the `standard.`)



Any port in a storm—5/6

...and **systemd** lets you listen on low ports:

```
[Service]
ExecStart=/usr/bin/exampled 127.0.0.2:80
User=example
AmbientCapabilities=CAP_NET_BIND_SERVICE
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
```



Any port in a storm—6/6

- So, that all works OK, and makes the output of `ss -ltu` readable.
- That is, if you're using **legacy IP (v4)**.
- In **standard IP (v6)**, guess how many local loopback addresses you get?
- Out of all those undecillions...



One!

∴ 1

(Geoff Huston is trying to fix it...)



More like *not-working*—1/2

- So, are we stuck with all these random ports?
- Well, let's start thinking a bit here...
- What really *is* computer networking?
- Communicating *between computers*.
- But all our processes are on the *same* computer...



More like *not-working*—2/2

- So...the addressing, the assigning, the routing, the flow control, the error correction...
- ...*all wasted*?
- We're not really...networking?



Dulce et Decorum Est

- Remember how people say of Unix and Linux:

“Everything’s a file!”

- Well...except for: network interfaces, TCP/UDP ports, signals, direct memory access, `ioctl`...
- Let’s do something about the first two.

In my domain—1/4

- Use **Unix domain sockets (UDS)** instead of network connections.
- These are bound as a *file* in the *filesystem*:
 - /run/EXAMPLED/server.sock
 - /run/user/1000/EXAMPLED/server.sock
- You have lots of them already: `ss -x`



```
~$ ss -x | head
```

Netid	State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port	Process
u_str	ESTAB	0	0	/run/systemd/journal/stdout	29820661 * 29820660	
u_str	ESTAB	0	0	*	31618719 * 31618720	
u_str	ESTAB	0	0	*	32268572 * 32268573	
u_str	ESTAB	0	0	*	19520328 * 19520329	
u_str	ESTAB	0	0	/run/dovecot/anvil-auth-penalty	32633603 * 32633602	
u_str	ESTAB	0	0	*	5539753 * 5539754	
u_dgr	ESTAB	0	0	/run/systemd/journal/dev-log	12459 * 0	
u_str	ESTAB	0	0	/run/systemd/journal/stdout	19520329 * 19520328	
u_dgr	ESTAB	0	0	/run/systemd/journal/socket	12461 * 0	

```
~$ █
```

In my domain—2/4

Unix domain sockets on Linux come in three flavors:

- **SOCK_STREAM**: Maps nicely to **TCP** (HTTP, SSH...)
- **SOCK_DGRAM**: Maps nicely to **UDP** (DNS, WireGuard...)
- **SOCK_SEQPACKET**: Custom protocol: best of both worlds (out of scope here)



In my domain—3/4

For a listening internal service (e.g. NetBox):

- In networking, you **bind** to a hostname and a TCP or UDP port: `127.0.0.1:8080`
- With a UDS, you **bind** to a path:
`/run/exampleled/server.socket`



In my domain—4/4

For a connecting reverse proxy (e.g. Nginx):

- In networking, you **connect** to a hostname and a TCP or UDP port: `127.0.0.1:8080`
- With a UDS, you **connect** to a path: `/run/exampleled/server.socket`



Advantages—1/4

Faster—double the throughput, half the CPU usage.

- Doesn't actually hit the disk (/run is a tmpfs).
- Skips network stuff it doesn't need:
 - addressing
 - routing
 - checksums
 - flow control



Advantages—2/4

Clearer—no more random high listen ports.

- Listening sockets can be arranged in `/run`, in a filesystem hierarchy.
- Give local services *names*, not *numbers*!
- Don't make sysadmins reach for `nc`...



Advantages—3/4

Safer—specify which users can connect.

- *Any user* can connect to local network listeners.
- UDS listeners respect Unix owner/group permissions.
- Your service might not even need its own authentication!

```
$ chgrp -c mygroup server.sock  
$ chmod -c g+rw server.sock
```



Advantages—4/4

More versatile—Files as servers is handy:

- Bind-mount (alias) a socket in a chroot? Works like any other file.
- Open private keys as root, and pass them safely to a less privileged process? `SCM_RIGHTS`!
- Tell the server your system username, for custom auth? `SO_PEERCRECRED`!



But *how?*—1/2

- Network applications are getting *much* better at supporting Unix listeners and connections now.
 - **Nginx** connects and listens natively.
 - **Redis**—dead to us, use **Valkey**—was an early adopter.
- Check your server's documentation! It could be as simple as a config change:

```
-bind=127.0.0.1:8080  
+bind=/run/exampleled/server.sock
```



But *how?*—2/2

- But perhaps you're *writing* the software.
- Some languages' core libraries make it easier than others.
 - Python's `socket` is totally workable.
- If you are writing a network server, please consider supporting UDS!



“But the software won’t do it!”

Use `socat`, or wrap it in `ip2unix`:

```
[Service]
ExecStart=/usr/bin/ip2unix \
  -r in,path=/run/EXAMPLED/server.socket \
  /usr/bin/EXAMPLED 127.0.0.1:8080
```

(Even better: *fix the software!*)



“But I’m debugging it in a browser!”

- Use **cURL**'s `--unix-socket`:

```
curl --unix-socket /run/examp1ed/server.socket \  
http://localhost/
```

- Use **socat** for a temporary network listener:

```
socat TCP-LISTEN:8080, fork \  
/run/examp1ed/server.socket
```



“But it’s a Docker container!”

Put the socket in a **shared volume**:

```
services:  
  exampled:  
    image: exampled  
    volumes:  
      # HOST_PATH:CONTAINER_PATH  
      - /run/exampled/server.sock:/run/server.sock  
      # Ensure write permissions  
    user: "1000:1000"
```



“But there are remote clients too!”

- **Check:** Can the server have *multiple listeners*?
- Listen on a UDS for *local* connections, and an internet socket for *remote* ones.
- Stick them together with [socat](#) if you have to...
- If you need TLS: [stunnel](#), [Nginx](#), [Caddy](#)...



Questions?

- [unix\(7\) manual page](#)
- [socat](#)—stick sockets together
- [ip2unix](#)—shoehorn UDS support

Email: tom@sanctum.geek.nz

Website: <https://sanctum.geek.nz/>

Fediverse: [@tejr@mastodon.sdf.org](https://mstdn.social/@tejr)

