Running multiple PgBouncer instances with systemd

PETER EISENTRAUT (/BLOG/AUTHOR/PETER-EISENTRAUT) AUGUST 13, 2020

Since PgBouncer runs as a single process, it is not straightforward to make use of multiple CPUs on a host. When you are running on real hardware, this is wasteful, since single-CPU server machines don't exist anymore, as far as I can tell. When using virtualization, you could just provision a single CPU per virtual host and run multiple virtual hosts with a single PgBouncer instance each. But there are also ways to run multiple instances of PgBouncer on a single host, so let's explore that. Since version 1.12, PgBouncer supports the `SO_REUSEPORT` socket option, which allows running multiple instances of PgBouncer on the same host on the same port and they will share the connections.

First, let's set up the basics of how to run a single instance of PgBouncer via systemd. PgBouncer ships with a sample service unit configuration file at `etc/pgbouncer.service` in the source tree that you can use to get started. Copy that file to `/etc/systemd/system/pgbouncer.service` and run `sudo systemctl daemon-reload` . Then you can run `sudo systemctl start pgbouncer` .

This is the unit file I'm using for this example. It is a slightly adjusted and abbreviated version of the file in the source tree.

```
[Unit]
Description=connection pooler for PostgreSQL
After=network.target

[Service]
Type=notify
User=postgres
ExecStart=/usr/local/bin/pgbouncer /etc/pgbouncer/pgbouncer.ini
ExecReload=/bin/kill -HUP $MAINPID
KillSignal=SIGINT

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

And this is the PgBouncer configuration file I'm using:

```
[databases]
;; add yours here

[pgbouncer]
listen_addr = localhost
listen_port = 6432
unix_socket_dir = /tmp

so_reuseport = 1
```

Now if you want to run multiple instances, you can obviously just make copies of these files and run them as entirely separate services. But here we want to make use of the systemd template system. So create a file `/etc/systemd/system/pgbouncer@.service` that looks like this:

```
[Unit]
Description=connection pooler for PostgreSQL (%i)
After=network.target

[Service]
Type=notify
User=postgres
ExecStart=/usr/local/bin/pgbouncer /etc/pgbouncer/pgbouncer.ini
ExecReload=/bin/kill -HUP $MAINPID
KillSignal=SIGINT

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

(Remember to `daemon-reload` at appropriate times. I'm not going to repeat that every time.)

Then use

```
systemctl start pgbouncer@1
systemctl start pgbouncer@2
# etc.
```

and you can start as many as you want. The specific numbers here are irrelevant, they just serve as identifiers. (They don't even have to be numbers.)

The problem with that setup is that while the TCP/IP listen sockets will happily share between all instances, per the `so_reuseport` setting, this does not apply to the Unix-domain sockets. So you would either have to turn off Unix-domain sockets, or create separate configuration files for each instance with different directories for the Unix-domain sockets, which would remove some of the elegance of this approach.

I recommend a new approach using socket activation, which is supported as of PgBouncer 1.14.

Create a file `/etc/systemd/system/pgbouncer@.socket` :

```
[Unit]
Description=sockets for PgBouncer

[Socket]
ListenStream=6432
ListenStream=%i
ListenStream=/tmp/.s.PGSQL.%i

ReusePort=true

[Install]
WantedBy=sockets.target
```

This gives us:

1. A TCP/IP socket on port 6432 for normal use. (You could also use 5432.)
2. Per-instance TCP/IP and Unix-domain sockets, for administration and monitoring.

The `ReusePort` setting corresponds to the `so_reuseport` setting in `pgbouncer.ini` . (When you use systemd socket activation, the listen socket settings in `pgbouncer.ini` are ignored.)

Then change `pgbouncer@.service` like this:

```
[Unit]
Description=connection pooler for PostgreSQL (%i)
After=network.target
Requires=pgbouncer@%i.socket

[Service]
Type=notify
User=postgres
ExecStart=/usr/local/bin/pgbouncer /etc/pgbouncer/pgbouncer.ini
ExecReload=/bin/kill -HUP $MAINPID
KillSignal=SIGINT

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

and then you can start

```
systemctl start pgbouncer@50001
systemctl start pgbouncer@50002
systemctl start pgbouncer@50003
systemctl start pgbouncer@50004
```

or

```
systemctl enable pgbouncer@50001
# etc.
```

to enable them for system start.

Here, the 50001+ numbers are the administration port numbers. You can use any numbers you want that are not used by anything else.

Now, for production traffic, you can connect like this as normal:

```
psql -h somehost -p 6432 -d yourdb ...
```

(must be TCP/IP). For monitoring and administration, you connect to each instance like this:

```
psql -p 50001 -d pgbouncer -U pgbouncer -c 'SHOW ...'
```

As alluded to at the beginning, there are many ways to set up and deploy PgBouncer. It depends on available hardware, deployment tools, and architecture requirements. If you have other ideas, please leave a comment or join the discussion on Gitter (https://gitter.im/pgbouncer/pgbouncer).

Share this (/#facebook) (/#x) (/#linkedin) (/#email) (https://www.addtoany.com/share#url=https%3A%2F%2Fenterprisedb.com%2Fblog%2Frunning-multiple-pgbouncer-instances-systemd&title=Running%20multiple%20PgBouncer%20instances%20with%20systemd)

Recent Comments

Add New Comment

Please **Log In (/accounts/login?fromURI=https://enterprisedb.com/blog/running-multiple-pgbouncer-instances-systemd)** to comment on this post.

More Blogs

PostgreSQL Contributor Story: Mark Wong **(/blog/postgresql-contributor-story-mark-wong)**

Earlier this year we started a program ("Developer U") to help colleagues who show promise for PostgreSQL Development to become contributors. Meet Mark Wong, wearer of many hats.

January 14, 2026

The Cost of Downtime and How to Reduce Its Impact on Your Organization **(/blog/cost-of-downtime)**

For today's enterprises, system downtime can significantly harm revenue, reputation, and overall resilience. Even a brief disruption can translate into financial losses ranging from thousands to millions of dollars. When...

January 08, 2026

Software Bill of Materials (SBOM): The Traceability You Need **(/blog/what-is-software-bill-of-materials)**

Modern applications are built from thousands of components, including open source libraries, third-party packages, plugins, and custom code. While this speeds up development, it also creates hidden risks in the...

January 08, 2026