
Brubeck (unmaintained)

Brubeck is a statsd-compatible stats aggregator written in C. Brubeck is currently unmaintained.

List of known maintained forks

- <https://github.com/lukepalmer/brubeck-new>

What is statsd?

Statsd is a metrics aggregator for Graphite (and other data storage backends). This technical documentation assumes working knowledge of what statsd is and how it works; please read the statsd documentation for more details.

Statsd is a good idea, and if you're using Graphite for metrics collection in your infrastructure, you probably want a statsd-compatible aggregator in front of it.

Tradeoffs

- Brubeck is missing many of the features of the original StatsD. We've only implemented what we felt was necessary for our metrics stack.
- Brubeck only runs on Linux. It won't even build on Mac OS X.
- Some of the performance features require a (moderately) recent version of the kernel that you may not have.

Building

Brubeck has the following dependencies:

- A Turing-complete computing device running a modern version of the Linux kernel (the kernel needs to be at least 2.6.33 in order to use multiple recvmsg support)
- A compiler for the C programming language
- Jansson ([libjansson-dev](#) on Debian) to load the configuration (version 2.5+ is required)
- OpenSSL ([libcrypto](#)) if you're building StatsD-Secure support
- libmicrohttpd ([libmicrohttpd-dev](#)) to have an internal HTTP stats endpoint. Build with `BRUBECK_NO_HTTP` to disable this.

Build brubeck by typing:

```
1 ./script/bootstrap
```

Other operating systems or kernels can probably build Brubeck too. More specifically, Brubeck has been seen to work under FreeBSD and OpenBSD, but this is not supported.

Supported Metric Types

Brubeck supports most of the metric types from statsd and many other implementations.

- `g` - Gauges
- `c` - Meters
- `C` - Counters
- `h` - Histograms
- `ms` - Timers (in milliseconds)

Client-sent sampling rates are ignored.

Visit the statsd docs for more information on metric types.

Interfacing

There are several ways to interact with a running Brubeck daemon.

Signals

Brubeck answers to the following signals:

- `SIGINT`, `SIGTERM`: shutdown cleanly
- `SIGHUP`: reopen the log files (in case you're using logrotate or an equivalent)
- `SIGUSR2`: dump a newline-separated list of all the metrics currently aggregated by the daemon and their types.

HTTP Endpoint

If enabled on the config file, Brubeck can provide an HTTP API to poll its status. The following routes are available:

- `GET /ping`: return a short JSON payload with the current status of the daemon (just to check it's up)

-
- `GET /stats`: get a large JSON payload with full statistics, including active endpoints and throughputs
 - `GET /metric/{metric_name}`: get the current status of a metric, if it's being aggregated
 - `POST /expire/{metric_name}`: expire a metric that is no longer being reported to stop it from being aggregated to the backend

Configuration

The configuration for Brubeck is loaded through a JSON file, passed on the commandline.

```
1 ./brubeck --config=my.config.json
```

If no configuration file is passed to the daemon, it will load `config.default.json`, which contains useful defaults for local development/testing.

The JSON file can contain the following sections:

- `server_name`: a string identifying the name for this specific Brubeck instance. This will be used by the daemon when reporting its internal metrics.
- `dumpfile`: a path where to store the metrics list when triggering a dump (see the section on Interfacing with the daemon)
- `http`: if existing, this string sets the listen address and port for the HTTP API
- `backends`: an array of the different backends to load. If more than one backend is loaded, brubeck will function in sharding mode, distributing aggregation load evenly through all the different backends through constant-hashing.
 - `carbon`: a backend that aggregates data into a Carbon cache. The backend sends all the aggregated data once every `frequency` seconds. By default the data is sent to the port 2003 of the Carbon cache (plain text protocol), but the pickle wire protocol can be enabled by setting `pickle` to `true` and changing the port accordingly.

```
1 {  
2   "type" : "carbon",  
3   "address" : "0.0.0.0",  
4   "port" : 2003,  
5   "frequency" : 10,  
6   "pickle: true  
7 }
```

We strongly encourage you to use the pickle wire protocol instead of plaintext, because carbon-relay.py is not very performant and will choke when parsing plaintext under

enough load. Pickles are much softer CPU-wise on the Carbon relays, aggregators and caches.

Hmmm pickles. Now I'm hungry. Lincoln when's lunch?

- **samplers**: an array of the different samplers to load. Samplers run on parallel and gather incoming metrics from the network.
 - **statsd**: the default statsd-compatible sampler. It listens on an UDP port for metrics packets. You can have more than one statsd sampler on the same daemon, but Brubeck was designed to support a single sampler taking the full metrics load on a single port.

```
1 {  
2   "type" : "statsd",  
3   "address" : "0.0.0.0",  
4   "port" : 8126,  
5 }
```

The StatsD sampler has the following options (and default values) for performance tuning:

- ★ **"workers"**: 4 number of worker threads that will service the StatsD socket endpoint. More threads means emptying the socket faster, but the context switching and cache smashing will affect performance. In general, you can saturate your NIC as long as you have enough worker threads (one per core) and a fast enough CPU. Set this to 1 if you want to run the daemon in event-loop mode. But that'd be silly. This is not Node.
- ★ **"multisock"**: **false** if set to true, Brubeck will use the **SO_REUSEPORT** flag available since Linux 3.9 to create one socket per worker thread and bind it to the same address/port. The kernel will then round-robin between the threads without forcing them to race for the socket. This improves performance by up to 30%, try benchmarking this if your Kernel is recent enough.
- ★ **"multimsg"**: 1 if set to greater than one, Brubeck will use the **recvmsg** syscall (available since Linux 2.6.33) to read several UDP packets (the specified amount) in a single call and reduce the amount of context switches. This doesn't improve performance much with several worker threads, but may have an effect in a limited configuration with only one thread. Make it a power of two for better results. As always, benchmark. YMMV.
- **statsd-secure**: like StatsD, but each packet has a HMAC that verifies its integrity. This is hella useful if you're running infrastructure in The Cloud (TM) (C) and you want to send back packets back to your VPN without them being tampered by third parties.

```
1 {
2   "type" : "statsd-secure",
3   "address" : "0.0.0.0",
4   "port" : 9126,
5   "max_drift" : 3,
6   "hmac_key" : "750c783e6ab0b503eaa86e310a5db738",
7   "replay_len" : 8000
8 }
```

The `address` and `port` parts are obviously the same as in `statsd`.

- * `max_drift` defines the maximum time (in seconds) that packets can be delayed since they were sent from the origin. All metrics come with a timestamp, so metrics that drift more than this value will silently be discarded.
- * `hmac_key` is the shared HMAC secret. The client sending the metrics must also know this in order to sign them.
- * `replay_len` is the size of the bloom filter that will be used to prevent replay attacks. We use a rolling bloom filter (one for every drift second), so `replay_len` should roughly be the amount of **unique** metrics you expect to receive in a 1s interval.

NOTE: StatsD-secure doesn't run with multiple worker threads because verifying signatures is already slow enough. Don't use this in performance critical scenarios.

NOTE: StatsD-secure uses a bloom filter to prevent replay attacks, so a small percentage of metrics *will* be dropped because of false positives. Take this into consideration.

NOTE: An HMAC does *not* encrypt the packets, it just verifies its integrity. If you need to protect the content of the packets from eavesdropping, get those external machines in your VPN.

NOTE: StatsD-secure may or may not be a good idea. If you have the chance to send all your metrics inside a VPN, I suggest you do that instead.

Testing

There's some tests in the `test` folder for key parts of the system (such as packet parsing, and all concurrent data access); besides that we test the behavior of the daemon live on staging and production systems.

- Small changes are deployed into production as-is, straight from their feature branch. Deployment happens in 3 seconds for all the Brubeck instances in our infrastructure, so we can roll back into the master branch immediately if something fails.

-
- For critical changes, we multiplex a copy of the metrics stream into an Unix domain socket, so we can have two instances of the daemon (old and new) aggregating to the production cluster and a staging cluster, and verify that the metrics flow into the two clusters is equivalent.
 - Benchmarking is performed on real hardware in our datacenter. The daemon is spammed with fake metrics across the network and we ensure that there are no regressions (particularly in the linear scaling between cores for the statsd sampler).

When in doubt, please refer to the part of the MIT license that says “*THE SOFTWARE IS PROVIDED ‘AS IS,’ WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED*”. We use Brubeck in production and have been doing so for years, but we cannot make any promises regarding availability or performance.

FAQ

- **I cannot hit 4 million UDP metrics per second. I want my money back.**

Make sure receiver-side scaling is properly configured in your kernel and that IRQs are being serviced by different cores, and that the daemon’s threads are not pinned to a specific core. Make sure you’re running the daemon in a physical machine and not a cheap cloud VPS. Make sure your NIC has the right drivers and it’s not bottlenecking. Install a newer kernel and try running with [SO_REUSEPORT](#).

If nothing works, refunds are available upon request. Just get mad at me on Twitter.