beanstalkc Tutorial

Release 0.3

Andreas Bolka

Feb 09, 2018

Contents

Welcome, dear stranger, to a tour de force through beanstalkd's capabilities. Say hello to your fellow travel companion, the beanstalkc client library for Python. You'll get to know each other fairly well during this trip, so better start off on a friendly note. And now, let's go!

1 Getting Started

You'll need beanstalkd listening at port 14711 to follow along. So simply start it using:

```
beanstalkd -l 127.0.0.1 -p 14711
```

Besides having beanstalkc installed, you'll typically also need PyYAML. If you insist, you can also use beanstalkc without PyYAML. For more details see *Appendix A* of this tutorial.

To use beanstalkc we have to import the library and set up a connection to an (already running) beanstalkd server:

```
>>> import beanstalkc
>>> beanstalk = beanstalkc.Connection(host='localhost', port=14711)
```

If we leave out the `host` and/or `port` parameters, `'localhost'` and `11300` would be used as defaults, respectively. There is also a `connect_timeout` parameter which determines how long, in seconds, the socket will wait for the server to respond to its initial connection attempt. If it is `None` then there will be no timeout; it defaults to `1`.

2 Basic Operation

Now that we have a connection set up, we can enqueue jobs:

```
>>> beanstalk.put('hey!')
1
```

Or we can request jobs:

```
>>> job = beanstalk.reserve()
>>> job.body
'hey!'
```

Once we are done with processing a job, we have to mark it as done, otherwise jobs are re-queued by beanstalkd after a “time to run” (120 seconds, per default) is surpassed. A job is marked as done, by calling `delete`:

```
>>> job.delete()
```

`reserve` blocks until a job is ready, possibly forever. If that is not desired, we can invoke `reserve` with a timeout (in seconds) how long we want to wait to receive a job. If such a `reserve` times out, it will return `None`:

```
>>> beanstalk.reserve(timeout=0) is None
True
```

If you use a timeout of 0, `reserve` will immediately return either a job or `None`.

Note that beanstalkc requires job bodies to be strings, conversion to/from strings is left up to you:

```
>>> beanstalk.put(42)
Traceback (most recent call last):
...
AssertionError: Job body must be a str instance
```

There is no restriction on what characters you can put in a job body, so they can be used to hold arbitrary binary data. If you want to send images, just `put` the image data as a string. If you want to send Unicode text, just use `unicode.encode` to convert it to a string with some encoding.

3 Tube Management

A single beanstalkd server can provide many different queues, called “tubes” in beanstalkd. To see all available tubes:

```
>>> beanstalk.tubes()
['default']
```

A beanstalkd client can choose one tube into which its job are put. This is the tube “used” by the client. To see what tube you are currently using:

```
>>> beanstalk.using()
'default'
```

Unless told otherwise, a client uses the 'default' tube. If you want to use a different tube:

```
>>> beanstalk.use('foo')
'foo'
>>> beanstalk.using()
'foo'
```

If you decide to use a tube, that does not yet exist, the tube is automatically created by beanstalkd:

```
>>> beanstalk.tubes()
['default', 'foo']
```

Of course, you can always switch back to the default tube. Tubes that don't have any client using or watching, vanish automatically:

```
>>> beanstalk.use('default')
'default'
>>> beanstalk.using()
'default'
>>> beanstalk.tubes()
['default']
```

Further, a beanstalkd client can choose many tubes to reserve jobs from. These tubes are "watched" by the client. To see what tubes you are currently watching:

```
>>> beanstalk.watching()
['default']
```

To watch an additional tube:

```
>>> beanstalk.watch('bar')
2
>>> beanstalk.watching()
['default', 'bar']
```

As before, tubes that do not yet exist are created automatically once you start watching them:

```
>>> beanstalk.tubes()
['default', 'bar']
```

To stop watching a tube:

```
>>> beanstalk.ignore('bar')
1
>>> beanstalk.watching()
['default']
```

You can't watch zero tubes. So if you try to ignore the last tube you are watching, this is silently ignored:

```
>>> beanstalk.ignore('default')
1
>>> beanstalk.watching()
['default']
```

To recap: each beanstalkd client manages two separate concerns: which tube newly created jobs are put into, and which tube(s) jobs are reserved from. Accordingly, there are two separate sets of functions for these concerns:

- `use` and `using` affect where `put` places jobs;
- `watch` and `watching` control where `reserve` takes jobs from.

Note that these concerns are fully orthogonal: for example, when you `use` a tube, it is not automatically `watch`-ed. Neither does `watch`-ing a tube affect the tube you are `using`.

4 Statistics

Beanstalkd accumulates various statistics at the server, tube and job level. Statistical details for a job can only be retrieved during the job's lifecycle. So let's create another job:

```
>>> beanstalk.put('ho?')
2

>>> job = beanstalk.reserve()
```

Now we retrieve job-level statistics:

```
>>> from pprint import pprint
>>> pprint(job.stats())
{'age': 0,
 ...
 'id': 2,
 ...
 'state': 'reserved',
 ...
 'tube': 'default'}
```

If you try to access job stats after the job was deleted, you'll get a `CommandFailed` exception:

```
>>> job.delete()
>>> job.stats()
Traceback (most recent call last):
...
CommandFailed: ('stats-job', 'NOT_FOUND', [])
```

Let's have a look at some numbers for the `'default'` tube:

```
>>> pprint(beanstalk.stats_tube('default'))
{...
 'current-jobs-ready': 0,
 'current-jobs-reserved': 0,
 'current-jobs-urgent': 0,
 ...
 'name': 'default',
 ...}
```

Finally, there's an abundant amount of server-level statistics accessible via the `Connection`'s `stats` method. We won't go into details here, but:

```
>>> pprint(beanstalk.stats())
{...
 'current-connections': 1,
 'current-jobs-buried': 0,
 'current-jobs-delayed': 0,
 'current-jobs-ready': 0,
 'current-jobs-reserved': 0,
 'current-jobs-urgent': 0,
 ...}
```

5 Advanced Operation

In “Basic Operation” above, we discussed the typical lifecycle of a job:

```
   put            reserve               delete
  -----> [READY] ---------> [RESERVED] --------> *poof*



(This illustration was taken from beanstalkd's protocol
documentation. It is originally contained in ``protocol.txt``,
part of the beanstalkd distribution.) #doctest:+SKIP
```

But besides `ready` and `reserved`, a job can also be `delayed` or `buried`. Along with those states come a few transitions, so the full picture looks like the following:

```
   put with delay               release with delay
  ----------------> [DELAYED] <------------.
                        |                   |
                        | (time passes)     |
                        |                   |
   put                  v     reserve       |       delete
  -----------------> [READY] ---------> [RESERVED] --------> *poof*
                       ^  ^                |  |
                       |   \  release      |  |
                       |    ``-------------'   |
                       |                      |
                       | kick                 |
                       |                      |
                       |       bury           |
                    [BURIED] <---------------'
                       |
                       |  delete
                        ``--------> *poof*



(This illustration was taken from beanstalkd's protocol
documentation. It is originally contained in ``protocol.txt``,
part of the beanstalkd distribution.) #doctest:+SKIP
```

Now let’s have a practical look at those new possibilities. For a start, we can create a job with a delay. Such a job will only be available for reservation once this delay passes:

```
>>> beanstalk.put('yes!', delay=1)
3

>>> beanstalk.reserve(timeout=0) is None
True
```

```
>>> job = beanstalk.reserve(timeout=1)
>>> job.body
'yes!'
```

If we are not interested in a job anymore (e.g. after we failed to process it), we can simply release the job back to the tube it came from:

```
>>> job.release()
>>> job.stats()['state']
'ready'
```

Want to get rid of a job? Well, just "bury" it! A buried job is put aside and is therefore not available for reservation anymore:

```
>>> job = beanstalk.reserve()
>>> job.bury()
>>> job.stats()['state']
'buried'

>>> beanstalk.reserve(timeout=0) is None
True
```

Buried jobs are maintained in a special FIFO-queue outside of the normal job processing lifecycle until they are kicked alive again:

```
>>> beanstalk.kick()
1
```

You can request many jobs to be kicked alive at once, `kick`'s return value will tell you how many jobs were actually kicked alive again:

```
>>> beanstalk.kick(42)
0
```

6 Inspecting Jobs

Besides reserving jobs, a client can also "peek" at jobs. This allows to inspect jobs without modifying their state. If you know the `id` of a job you're interested, you can directly peek at the job. We still have job #3 hanging around from our previous examples, so:

```
>>> job = beanstalk.peek(3)
>>> job.body
'yes!'
```

Note that this peek did *not* reserve the job:

```
>>> job.stats()['state']
'ready'
```

If you try to peek at a non-existing job, you'll simply see nothing:

```
>>> beanstalk.peek(42) is None
True
```

If you are not interested in a particular job, but want to see jobs according to their state, beanstalkd also provides a few commands. To peek at the same job that would be returned by `reserve` (i.e. the next ready job) use `peek-ready`:

```
>>> job = beanstalk.peek_ready()
>>> job.body
'yes!'
```

Note that you can't release, or bury a job that was not reserved by you. Those requests on unreserved jobs are silently ignored:

```
>>> job.release()
>>> job.bury()

>>> job.stats()['state']
'ready'
```

You can, though, delete a job that was not reserved by you:

```
>>> job.delete()
>>> job.stats()
Traceback (most recent call last):
...
CommandFailed: ('stats-job', 'NOT_FOUND', [])
```

Finally, you can also peek into the special queues for jobs that are delayed:

```
>>> beanstalk.put('o tempores', delay=120)
4
>>> job = beanstalk.peek_delayed()
>>> job.stats()['state']
'delayed'
```

. . . or buried:

```
>>> beanstalk.put('o mores!')
5
>>> job = beanstalk.reserve()
>>> job.bury()

>>> job = beanstalk.peek_buried()
>>> job.stats()['state']
'buried'
```

7 Job Priorities

Without job priorities, beanstalkd operates as a FIFO queue:

```
>>> _ = beanstalk.put('1')
>>> _ = beanstalk.put('2')

>>> job = beanstalk.reserve() ; print job.body ; job.delete()
1
>>> job = beanstalk.reserve() ; print job.body ; job.delete()
2
```

If need arises, you can override this behaviour by giving different jobs different priorities. There are three hard facts to know about job priorities:

1. Jobs with lower priority numbers are reserved before jobs with higher priority numbers.
2. beanstalkd priorities are 32-bit unsigned integers (they range from 0 to 2**32 - 1).
3. beanstalkc uses 2**31 as default job priority (`beanstalkc.DEFAULT_PRIORITY`).

To create a job with a custom priority, use the keyword-argument `priority`:

```
>>> _ = beanstalk.put('foo', priority=42)
>>> _ = beanstalk.put('bar', priority=21)
>>> _ = beanstalk.put('qux', priority=21)

>>> job = beanstalk.reserve() ; print job.body ; job.delete()
bar
>>> job = beanstalk.reserve() ; print job.body ; job.delete()
qux
>>> job = beanstalk.reserve() ; print job.body ; job.delete()
foo
```

Note how `'bar'` and `'qux'` left the queue before `'foo'`, even though they were enqueued well after `'foo'`. Note also that within the same priority jobs are still handled in a FIFO manner.

8 Fin!

```
>>> beanstalk.close()
```

That's it, for now. We've left a few capabilities untouched (touch and time-to-run). But if you've really read through all of the above, send me a message and tell me what you think of it. And then go get yourself a treat. You certainly deserve it.

9 Appendix A: beanstalkc and YAML

As beanstalkd uses YAML for diagnostic information (like the results of `stats()` or `tubes()`), you'll typically need PyYAML. Depending on your performance needs, you may want to supplement that with the libyaml C extension.

If, for whatever reason, you cannot use PyYAML, you can still use beanstalkc and just leave the YAML responses unparsed. To do that, pass `parse_yaml=False` when creating the `Connection`:

```
>>> beanstalk = beanstalkc.Connection(host='localhost',
...                                   port=14711,
...                                   parse_yaml=False)

>>> beanstalk.tubes()
'---\n- default\n'

>>> beanstalk.stats_tube('default')
'---\nname: default\ncurrent-jobs-urgent: 0\n...'

>>> beanstalk.close()
```

This possibility is mostly useful if you don't use the introspective capabilities of beanstalkd (`Connection#tubes`, `Connection#watching`, `Connection#stats`, `Connection#stats_tube`, and `Job#stats`).

Alternatively, you can also pass a function to be used as YAML parser:

```
>>> beanstalk = beanstalkc.Connection(host='localhost',
...                                   port=14711,
...                                   parse_yaml=lambda x: x.split('\n'))

>>> beanstalk.tubes()
['---', '- default', '']

>>> beanstalk.stats_tube('default')
['---', 'name: default', 'current-jobs-urgent: 0', ...]

>>> beanstalk.close()
```

This should come in handy if PyYAML simply does not fit your needs.