Antenna Documentation

Release 0.1

Mozilla

May 24, 2018

Contents

Breakpad crash collector web app that handles incoming crash reports and saves them to AWS S3.

Uses Python 3, Gunicorn, gevent, Falcon and some other things.

- Free software: Mozilla Public License version 2.0
- Code: https://github.com/mozilla-services/antenna/
- Documentation: https://antenna.readthedocs.io/

CHAPTER 1

Contents

User docs:

Antenna: Breakpad crash report collector

Breakpad crash collector web app that handles incoming crash reports and saves them to AWS S3.

Uses Python 3, Gunicorn, gevent, Falcon and some other things.

- Free software: Mozilla Public License version 2.0
- Documentation: https://antenna.readthedocs.io/
- Bugs: Report a bug

Quickstart

This is a quickstart that uses Docker so you can see how the pieces work. Docker is also used for local development of Antenna.

For more comprehensive documentation or instructions on how to set this up in production, see docs.

1. Clone the repository:

   ```
   $ git clone https://github.com/mozilla-services/antenna
   ```

2. Install docker 1.10.0+ and install docker-compose 1.6.0+ on your machine

3. Download and build Antenna docker containers:

   ```
   $ make build
   ```

   Anytime you want to update the containers, you can run `make build`.

4. Run with a prod-like fully-functional configuration.

(a) Running:

```
$ make run
```

You should see a lot of output. It'll start out with something like this:

```
ANTENNA_ENV="dev.env" /usr/bin/docker-compose up web
antenna_statsd_1 is up-to-date
antenna_localstack-s3_1 is up-to-date
Recreating antenna_web_1
Attaching to antenna_web_1
web_1       | [2016-11-07 15:39:21 +0000] [7] [INFO] Starting gunicorn 19.6.0
web_1       | [2016-11-07 15:39:21 +0000] [7] [INFO] Listening at: http://0.0.
↪0.0:8000 (7)
web_1       | [2016-11-07 15:39:21 +0000] [7] [INFO] Using worker: gevent
web_1       | [2016-11-07 15:39:21 +0000] [10] [INFO] Booting worker with pid:␣
↪10
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: Setting up␣
↪metrics: <class 'antenna.metrics.DogStatsdMetrics'>
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.metrics:␣
↪DogStatsdMetrics configured: statsd:8125 mcboatface
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: BASEDIR=/app
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: LOGGING_
↪LEVEL=DEBUG
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: METRICS_
↪CLASS=antenna.metrics.DogStatsdMetrics
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: DUMP_
↪FIELD=upload_file_minidump
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: DUMP_ID_
↪PREFIX=bp-
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪CLASS=antenna.ext.s3.crashstorage.S3CrashStorage
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: THROTTLE_
↪RULES=antenna.throttler.mozilla_rules
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪ACCESS_KEY=foo
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪SECRET_ACCESS_KEY=*****
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪REGION=us-east-1
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪ENDPOINT_URL=http://localstack-s3:5000
web_1       | [2016-11-07 15:39:21 +0000] [INFO] antenna.app: CRASHSTORAGE_
↪BUCKET_NAME=antennabucket
```

(b) Verify things are running:

In another terminal, you can verify the proper containers are running with:

```
$ docker-compose ps
```

You should see containers with names `web`, `statsd` and `localstack-s3`.

(c) Send in a crash report:

You can send a crash report into the system and watch it go through the steps:

```
$ ./bin/send_crash_report.sh
...
<curl http output>
```

```
...
CrashID=bp-6c43aa7c-7d34-41cf-85aa-55b0d2160622
*  Closing connection 0
```

You should get a CrashID back from the HTTP POST. You'll also see docker logging output something like this:

```
web_1       | [2016-11-07 15:48:45 +0000] [INFO] antenna.breakpad_resource:␣
↪a448814e-16dd-45fb-b7dd-b0b522161010 received with existing crash_id
web_1       | [2016-11-07 15:48:45 +0000] [INFO] antenna.breakpad_resource:␣
↪a448814e-16dd-45fb-b7dd-b0b522161010: matched by is_firefox_desktop;␣
↪returned ACCEPT
web_1       | [2016-11-07 15:48:45 +0000] [INFO] antenna.breakpad_resource:␣
↪a448814e-16dd-45fb-b7dd-b0b522161010 accepted
web_1       | [2016-11-07 15:48:45 +0000] [INFO] antenna.breakpad_resource:␣
↪a448814e-16dd-45fb-b7dd-b0b522161010 saved
```

(d) See the data in localstack-s3:

The `localstack-s3` container stores data in memory and the data doesn't persist between container restarts.

You can use the aws-cli to access it. For example:

```
AWS_ACCESS_KEY_ID=foo AWS_SECRET_ACCESS_KEY=foo \
    aws --endpoint-url=http://localhost:5000 \
        --region=us-east-1 \
        s3 ls s3://antennabucket/
```

If you do this a lot, turn it into a shell script.

(e) Look at runtime metrics with Grafana:

The `statsd` container has Grafana. You can view the statsd data via Grafana in your web browser http://localhost:9000.

To log into Grafana, use username `admin` and password `admin`.

You'll need to set up a Graphite datasource pointed to `http://localhost:8000`.

The statsd namespace set in the `dev.env` file is "mcboatface".

(f) When you're done–stopping Antenna:

When you're done with the Antenna process, hit CTRL-C to gracefully kill the docker web container.

If you want to run with a different Antenna configuration, put the configuration in an env file and then set `ANTENNA_ENV`. For example:

```
$ ANTENNA_ENV=my.env make run
```

See `dev.env` and the docs for configuration options.

5. Run tests:

```
$ make test
```

If you need to run specific tests or pass in different arguments, you can run bash in the base container and then run `py.test` with whatever args you want. For example:

```
$ make shell
app@...$ py.test

<pytest output>

app@...$ py.test tests/unittest/test_crashstorage.py
```

We're using py.test for a test harness and test discovery.

For more details on running Antenna or hacking on Antenna, see the docs.

Antenna configuration

Contents

Introduction

Configuration Antenna is not too hard. You have some basic choices to make. You codify the configuration into an env file. Then set `ANTENNA_ENV` environment variable to the path to the env file and you're done.

Here's an example. This uses Datadog installed on the EC2 node for metrics and also IAM bound to the EC2 node that Antenna is running on so it doesn't need S3 credentials for crashstorage.

```
# Metrics things
METRICS_CLASS=antenna.metrics.DogStatsdMetrics
STATSD_NAMESPACE=mcboatface

# BreakdpadSubmitterResource settings
CRASHSTORAGE_CLASS=antenna.ext.s3.crashstorage.S3CrashStorage

# S3CrashStorage and S3Connection settings
CRASHSTORAGE_BUCKET_NAME=org-myorg-mybucket
```

Application

First, you need to configure the application-scoped variables.

Configuration
These all have sane defaults, so you don't have to configure any of this.

Options

- **BASEDIR** (*str*) – The root directory for this application to find and store things.

  Defaults to `'/home/docs/checkouts/readthedocs.org/user_builds/antenna/checkouts/latest'`.

- **LOGGING_LEVEL** (*str*) – The logging level to use. DEBUG, INFO, WARNING, ERROR or CRITICAL

  Defaults to `'DEBUG'`.

- **METRICS_CLASS** (*<ListOf(everett.manager.parse_class)>*) – ('Comma-separated list of metrics backends to use. Possible options: "antenna.metrics.LoggingMetrics" and "antenna.metrics.DatadogMetrics"',)

  Defaults to `'antenna.metrics.LoggingMetrics'`.

- **SECRET_SENTRY_DSN** (*str*) – Sentry DSN to use. See https://docs.sentry.io/quickstart/#configure-the-dsn for details. If this is not set an unhandled exception logging middleware will be used instead.

  Defaults to `''`.

- **HOST_ID** (*str*) – Identifier for the host that is running Antenna. This identifies this Antenna instance in the logs and makes it easier to correlate Antenna logs with other data. For example, the value could be a public hostname, an instance id, or something like that. If you do not set this, then socket.gethostname() is used instead.

  Defaults to `''`.

Metrics

LoggingMetrics

component `antenna.metrics.`**LoggingMetrics**
Configuration for LoggingMetrics backend.

DogStatsd metrics

component `antenna.metrics.`**DogStatsdMetrics**
Configuration for DatadogMetrics backend.

Options

- **STATSD_HOST** (*str*) – Hostname for the statsd server

  Defaults to `'localhost'`.

- **STATSD_PORT** (*int*) – Port for the statsd server

  Defaults to `'8125'`.

- **STATSD_NAMESPACE** (*str*) – Namespace for these metrics

  Defaults to `''`.

Breakpad crash resource

component `antenna.breakpad_resource.`**`BreakpadSubmitterResource`**

Handles incoming breakpad crash reports and saves to crashstorage

This handles incoming HTTP POST requests containing breakpad-style crash reports in multipart/form-data format.

It can handle compressed or uncompressed POST payloads.

It parses the payload from the HTTP POST request, runs it through the throttler with the specified rules, generates a crash_id, returns the crash_id to the HTTP client and then saves the crash using the configured crashstorage class.

Note: From when a crash comes in to when it's saved by the crashstorage class, the crash is entirely in memory. Keep that in mind when figuring out how to scale your Antenna nodes.

The most important configuration bit here is choosing the crashstorage class.

For example:

```
CRASHSTORAGE_CLASS=antenna.ext.s3.crashstorage.S3CrashStorage
```

Options

- **DUMP_FIELD** (*str*) – the name of the field in the POST data for dumps

  Defaults to `'upload_file_minidump'`.

- **DUMP_ID_PREFIX** (*str*) – the crash type prefix

  Defaults to `'bp-'`.

- **CRASHSTORAGE_CLASS** (*everett.manager.parse_class*) – the class in charge of storing crashes

  Defaults to `'antenna.ext.crashstorage_base.NoOpCrashStorage'`.

- **CONCURRENT_CRASHMOVERS** (*int*) – the number of crashes concurrently being saved to s3

  Defaults to `'2'`.

Throttler

component `antenna.throttler.`**`Throttler`**

Accepts/rejects incoming crashes based on specified rule set

The throttler can throttle incoming crashes using the content of the crash. To throttle, you set up a rule set which is a list of `Rule` instances. That goes in a Python module which is loaded at run time.

If you don't want to throttle anything, use this:

```
THROTTLE_RULES=antenna.throttler.accept_all
```

To set up a rule set, put it in a Python file and define the rule set there. For example, you could have file `myruleset.py` with this in it:

```
from antenna.throttler import Rule

rules = [
    Rule('ProductName', 'Firefox', 100),
    # ...
]
```

then set `THROTTLE_RULES` to the path for that. For example, depending on the current working directory and `PYTHONPATH`, the above could be:

```
THROTTLE_RULES=myruleset.rules
```

FIXME(willkg): Flesh this out.

Options **`THROTTLE_RULES`** (*`antenna.throttler.parse_attribute`*) – Python dotted path to ruleset

Defaults to `'antenna.throttler.mozilla_rules'`.

Crash storage

For crash storage, you have three options one of which is a no-op for debugging.

NoOpCrashStorage

The `NoOpCrashStorage` class is helpful for debugging, but otherwise shouldn't be used.

component `antenna.ext.crashstorage_base.`**`NoOpCrashStorage`**
This is a no-op crash storage that logs crashes it would have stored

It keeps track of the last 10 crashes in `.crashes` instance attribute with the most recently stored crash at the end of the list. This helps when writing unit tests for Antenna.

Filesystem

The `FSCrashStorage` class will save crash data to disk. If you choose this, you'll want to think about what happens to the crash after Antenna has saved it and implement that.

component `antenna.ext.fs.crashstorage.`**`FSCrashStorage`**
Saves raw crash files to the file system.

This generates a tree something like this which mirrors what we do on S3:

```
<FS_ROOT>/
    <YYYYMMDD>/
        raw_crash/
            <CRASHID>.json
        dump_names/
            <CRASHID>.json
        <DUMP_NAME>/
            <CRASHID>
```

Couple of things to note:

1. This doesn't ever delete anything from the tree. You should run another process to clean things up.
2. If you run out of disk space, this component will fail miserably. There's no way to recover from a full disk–you will lose crashes.

FIXME(willkg): Can we alleviate or reduce the likelihood of the above?

When set as the BreakpadSubmitterResource crashstorage class, configuration for this class is in the `CRASHSTORAGE` namespace.

Example:

```
CRASHSTORAGE_FS_ROOT=/tmp/whatever
```

Options **`CRASHSTORAGE_FS_ROOT`** (*`str`*) – path to where files should be stored

Defaults to `'/tmp/antenna_crashes'`.

AWS S3

The `S3CrashStorage` class will save crash data to AWS S3. You might be able to use this to save to other S3-like systems, but that's not tested or supported.

component `antenna.ext.s3.connection.`**`S3Connection`**

Connection object for S3.

Credentials and permissions

When configuring this connection object, you can do one of two things:

1. provide `ACCESS_KEY` and `SECRET_ACCESS_KEY` in the configuration, OR
2. use one of the other methods described in the boto3 docs http://boto3.readthedocs.io/en/latest/guide/configuration.html#configuring-credentials

The AWS credentials that Antenna is configured with must have the following Amazon S3 permissions:

- `s3:ListBucket`

  When Antenna starts up, `S3Connection` will call `HEAD` on the bucket verifying the bucket exists, the endpoint url is good, it's accessible and the credentials are valid. This means that the credentials you use must have "list" permissions on the bucket.

  If that fails, then this will raise an error and will halt startup.

  > **Warning:** This does not verify that it has write permissions to the bucket. Make sure to test your configuration by sending a test crash and watch your logs at startup!

- `s3:PutObject`

  This permission is used to save items to the bucket.

Retrying saves

When saving crashes, this connection will retry saving several times. Then give up. The crashmover coroutine will put the crash back in the queue to retry later. Crashes are never thrown out.

When set as the BreakpadSubmitterResource crashstorage class, configuration for this class is in the `CRASHSTORAGE` namespace.

Example:

```
CRASHSTORAGE_BUCKET_NAME=mybucket
CRASHSTORAGE_REGION=us-west-2
CRASHSTORAGE_ACCESS_KEY=somethingsomething
CRASHSTORAGE_SECRET_ACCESS_KEY=somethingsomething
```

Options

- **CRASHSTORAGE_ACCESS_KEY** (*str*) – AWS S3 access key. You can also specify AWS_ACCESS_KEY_ID which is the env var used by boto3.

  Defaults to `''`.

- **CRASHSTORAGE_SECRET_ACCESS_KEY** (*str*) – AWS S3 secret access key. You can also specify AWS_SECRET_ACCESS_KEY which is the env var used by boto3.

  Defaults to `''`.

- **CRASHSTORAGE_REGION** (*str*) – AWS S3 region to connect to. For example, `us-west-2`

  Defaults to `'us-west-2'`.

- **CRASHSTORAGE_ENDPOINT_URL** (*str*) – endpoint_url to connect to; None if you are connecting to AWS. For example, `http://localhost:4569/`.

  Defaults to `''`.

- **CRASHSTORAGE_BUCKET_NAME** (*str*) – AWS S3 bucket to save to. Note that the bucket must already have been created and must be in the region specified by `region`.

component `antenna.ext.s3.crashstorage.`**S3CrashStorage**

Saves raw crash files to S3.

This will save raw crash files to S3 in a pseudo-tree something like this:

```
<BUCKET>
   v1/
       dump_names/
           <CRASHID>
       <DUMPNAME>/
           <CRASHID>
   v2/
       raw_crash/
           <ENTROPY>/
               <YYYYMMDD>/
                   <CRASHID>
```

When set as the BreakpadSubmitterResource crashstorage class, configuration for this class is in the `CRASHSTORAGE` namespace.

Generally, if the default connection class is fine, you don't need to do any configuration here.

Options **CRASHSTORAGE_CONNECTION_CLASS** (*everett.manager.parse_class*) – S3 connection class to use

Defaults to `'antenna.ext.s3.connection.S3Connection'`.

Putting Antenna in production

Contents

High-level things

Antenna is a WSGI application that uses Gunicorn as the WSGI server.

We use nginx in front of that, but you might be able to use other web servers.

Note: Make sure to use HTTPS–don't send your users' crash reports over HTTP.

Gunicorn configuration

For Gunicorn configuration, see `Dockerfile`. You'll want to set the following:

`GUNICORN_WORKERS`

The number of Antenna processes to spin off. We use 2x+1 where x is the number of processors on the machine we're using.

This is the `workers` Gunicorn configuration setting.

`GUNICORN_WORKER_CONNECTIONS`

This is the number of coroutines to spin off to handle incoming HTTP connections (crash reports). Gunicorn's default is 1000. That's what we use in production.

Note that the Antenna heartbeat insinuates itself into this coroutine pool, so you need 2 at a bare minimum.

This is the `worker-connections` Gunicorn configuration setting.

`GUNICORN_WORKER_CLASS`

This has to be set to `gevent`. Antenna does some `GeventWorker` specific things and won't work with anything else.

This is the `worker-class` Gunicorn configuration setting.

Health endpoints

Antenna exposes several URL endpoints to help you run it at scale.

`/__lbheartbeat__`

Always returns an HTTP 200. This tells the load balancer that this Antenna instance is handling connections still.

`/__heartbeat__`

This endpoint returns some more data. Depending on how you have Antenna configured, this might do a HEAD on the s3 bucket or other things. It returns its findings in the HTTP response body.

`/__version__`

Returns information related to the git sha being run in the HTTP response body.

For example:

```
{
    "commit": "5cc6f5170973c7af2e4bb2097679a82ae5659466",
    "version": "",
    "source": "https://github.com/mozilla-services/antenna",
    "build": "https://circleci.com/gh/mozilla-services/antenna/331"
}
```

`/__broken__`

Intentionally throws an exception that is unhandled. This helps testing Sentry or other error monitoring.

This is a nuissance if abused, so it's worth setting up your webserver to require basic auth or something for this endpoint.

Environments

Will this run in Heroku? Probably, but you'll need to do some Heroku footwork to set it up.

Will this run on AWS? Yes–that's what we do.

Will this run on [insert favorite environment]? I have no experience with other systems, but it's probably the case you can get it to work. If you can't save crashes to Amazon S3, you can always write your own storage class to save it somewhere else.

EC2 instance size and scaling

We're setting up Antenna to run on Amazon EC2 x4.large nodes. Antenna isn't very CPU intensive, but it is very network intensive (it's essentially an upload server) and it queues things in memory.

Our cluster is set to autoscale on network in. When network in hits 600,000,000, then it adds another node to the cluster. Network in is being used as a proxy for number of incoming crashes.

For Socorro, our median incoming crash is 400k and our typical load is between 1,500 crashes/min.

Antenna on two x4.large nodes can handle 1,500 crashes/min without a problem and without scaling up.

What happens after Antenna collects a crash?

Antenna saves the crash to the crash storage system you specify. We save our crashes to AWS S3.

So, yay–we have raw crash data on AWS S3. What happens next?

Currently, we have `s3:PutObject` events for `v2/raw_crash` prefix trigger an AWS Lambda function socorro-pigeon. That takes the crash id andadds it to a RabbitMQ queue. We have a processor that watches that queue, pulls the crash data from AWS S3, processes it and then the crash continues through our crash ingestion pipeline.

You could do something along these lines. You could write your own crash storage class that does other things.

Troubleshooting

Antenna doesn't start up

Antenna won't start up if it's configured wrong.

Things to check:

1. If you're using Sentry and it's set up correctly, then Antenna will send startup errors to Sentry and you can see it there.
2. Check the logs for startup errors. They'll have the string "Unhandled startup exception".
3. Is the configuration correct?

AWS S3 bucket permission issues

At startup, Antenna will try to Head the AWS S3 bucket and if it fails, will refuse to start up. It does this so that it doesn't start up, then get a crash and then fail to submit the crash due to permission issues. At that point, you'd have lost the crash.

If you're seeing errors like:

```
[ERROR] antenna.app: Unhandled startup exception: ... botocore.exceptions.ClientError:
An error occurred (403) when calling the HeadBucket operation: Forbidden
```

it means that the credentials that Antenna is using don't have the right permissions to the AWS S3 bucket.

Things to check:

1. Check the bucket and region that Antenna is configured with. It'll be in the logs when Antenna starts up.
2. Check that Antenna has the right AWS credentials.
3. Try using the credentials that Antenna is using to access the bucket.

Logs are getting lost / StatsD data is getting lost

Depending on how you're collecting logs and StatsD data, it's possible that you might lose this data if Antenna is under so much load that it's saturating the network interface.

You might see evidence of this by seeing lines in the logs saying a crash was saved, but no line indicating it was received. Or vice versa.

You might see evidence of this in StatsD when incoming crashes and saved crashes off by a large number.

Things to check:

1. What's the network out amount for this node? Is it too low?
2. What happens if you increase the capacity for the node? Or if the node is in a cluster, add more nodes to the cluster?

The `save_queue_size > 0` and climbing

This means Antenna is having trouble keeping up with incoming crashes.

Things to check:

1. Increase or decrease the number in the `concurrent_crashmovers` configuration variable.

   Too many will cause a single crash to take longer to save.

   Too few will reduce the efficiency regarding parallelizing around network I/O slowness.

   If you've already tuned this configuration variable, skip this step.
2. Increase the number of nodes in the cluster to better share the load.
3. Increase the node capacity so that it has more network out bandwidth.

Architecture

Purpose

Antenna handles incoming breakpad crash reports and saves them to AWS S3.

Requirements

Antenna is built with the following requirements:

1. **Return a crash id to the client quickly**

   Antenna should return a crash id and close the HTTP connection as quickly as possible. This means we need to save to AWS S3 as a separate step.
2. **Try hard not to drop crashes**

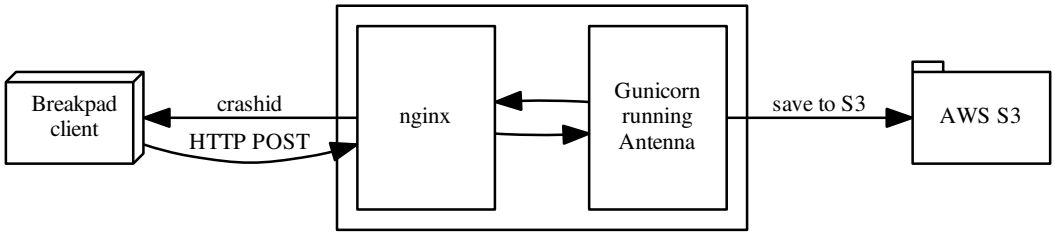
   Antenna tries hard not to drop crashes and lose data. It tries to get the crash to AWS S3 as quickly as possible so that it's sitting on as few crash reports as possible.
3. **Minimal dependencies**

   Every dependency we add is another software cycle we have to track causing us to have to update our code when they change.
4. **Make setting it up straight-forward**

   Antenna should be straight-forward to set up. Minimal configuration options. Solid documentation.
5. **Easy to test**

   Antenna should be built in such a way that it's easy to write tests for. Tests that are easy to read and easy to write are easy to verify and this will make it likely that the software is higher quality.

High-level architecture



We run multiple Antenna nodes behind an ELB.

Data flow

This is the rough data flow:

1. Breakpad client submits a crash report via HTTP POST with a multipart/form-data encoded payload.
2. Antenna's `BreakpadSubmitterResource` handles the HTTP POST request.

   If the payload is compressed, it uncompresses it.

   It extracts the payload converting it into a dict.

   It throttles the crash.

   It generates a crash id.

   It returns the crash id to the breakpad client.
3. The `BreakpadSubmitterResource` tosses the crash in the `crashmover_save_queue`. It tosses the crash in the `crashmover_save_queue`.
4. At this point, the HTTP conversation is done and the connection ends.
5. ... time passes depending on how many things are in the `crashmover_save_queue`.
6. A crashmover coroutine frees up, pulls the crash out of the `crashmover_save_queue`, and then tries to save it to whatever crashstorage class is set up. If it's *`S3CrashStorage`*, then it saves it to AWS S3.

   If the save is successful, then the coroutine moves on to the next crash in the queue.

   If the save is not successful, the coroutine puts the crash back in the queue and moves on with the next crash.

Diagnostics

Logs to stdout

Antenna logs its activity to stdout in mozlog format.

You can see crashes being accepted and saved:

```
{"Timestamp": 1493998643710555648, "Type": "antenna.breakpad_resource","Logger":
↪"antenna", "Hostname": "ebf44d051438", "EnvVersion": "2.0", "Severity": 6, "Pid":␣
↪15, "Fields": {"host_id": "ebf44d051438", "message": "8e01b4e0-f38f-4b16-bc5a-
↪043971170505: matched by is_firefox_desktop; returned DEFER"}}
{"Timestamp": 1493998645733482752, "Type": "antenna.breakpad_resource", "Logger":
↪"antenna", "Hostname": "ebf44d051438", "EnvVersion": "2.0", "Severity": 6, "Pid":␣
↪15, "Fields": {"host_id": "ebf44d051438", "message": "8e01b4e0-f38f-4b16-bc5a-
↪043971170505 saved"}}
```

You can see the heartbeat kicking off:

```
{"Timestamp": 1493998645532856576, "Type": "antenna.heartbeat", "Logger": "antenna",
↪"Hostname": "ebf44d051438", "EnvVersion": "2.0", "Severity": 7, "Pid": 15, "Fields
↪": {"host_id": "ebf44d051438", "message": "thump"}}
```

Statsd

Antenna sends data to statsd. Read the code for what's available where and what it means.

Here are some good ones:

- `breakpad_resource.incoming_crash`

  Counter. Denotes an incoming crash.

- `throttle.*`

  Counters. Throttle results. Possibilities: `accept`, `defer`, `reject`.

- `breakpad_resource.save_crash.count`

  Counter. Denotes a crash has been successfully saved.

- `breakpad_resource.save_queue_size`

  Gauge. Tells you how many things are sitting in the `crashmover_save_queue`.

  Note: If this number is > 0, it means that Antenna is having difficulties keeping up with incoming crashes.

- `breakpad_resource.on_post.time`

  Timing. This is the time it took to handle the HTTP POST request.

- `breakpad_resource.crash_save.time`

  Timing. This is the time it took to save the crash to S3.

- `breakpad_resource.crash_handling.time`

  Timing. This is the total time the crash was in Antenna-land from receiving the crash to saving it to S3.

Sentry

Antenna works with Sentry and will send unhandled startup errors and other unhandled errors to Sentry where you can more easily see what's going on. You can use the hosted Sentry or run your own Sentry instance–either will work fine.

AWS S3 file hierarchy

If you use the Amazon Web Services S3 crashstorage component, then crashes get saved in this hierarchy in the bucket:

- `/v2/raw_crash/<ENTROPY>/<DATE>/<CRASHID>`
- `/v1/dump_names/<CRASHID>`

And then one or more dumps in directories by dump name:

- `/v1/<DUMP_NAME>/<CRASHID>`

Note that `upload_file_minidump` gets converted to `dump`.

For example, a crash with id `00007bd0-2d1c-4865-af09-80bc00170413` and two dumps "upload_file_minidump" and "upload_file_minidump_flash1" gets these files saved:

```
v2/raw_crash/000/20170413/00007bd0-2d1c-4865-af09-80bc00170413

    Raw crash in serialized in JSON.

v1/dump_names/00007bd0-2d1c-4865-af09-80bc00170413

    Map of dump_name to file name serialized in JSON.

v1/dump/00007bd0-2d1c-4865-af09-80bc00170413

    upload_file_minidump dump.

v1/upload_file_minidump_flash1/00007bd0-2d1c-4865-af09-80bc00170413

    upload_file_minidump_flash1 dump.
```

All about breakpad

Links about breakpad

Breakpad project home page: https://chromium.googlesource.com/breakpad/breakpad

Firefox Breakpad page: https://wiki.mozilla.org/Breakpad

> Note: A lot of this is out of date.

Socorro docs: http://socorro.readthedocs.io/en/latest/

> Notes on testing collector and processor: http://socorro.readthedocs.io/en/latest/configuring-socorro.html#test-collection-and-processing

Where do reports come from?

From Ted:

> We use different code to submit crash reports on all 4 major platforms we ship Firefox on: Windows, OS X, Linux, Android, and we also have a separate path for submitting crash reports from within Firefox (for crashes in content processes, plugin processes, and used when you click an unsubmitted report in about:crashes).

For all the desktop platforms, the crashreporter client (the window that says "We're Sorry") is some C++ code that lives here: https://dxr.mozilla.org/mozilla-central/source/toolkit/crashreporter/client/

For Windows the submission code in the client is here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/client/crashreporter_win.cpp#391

which calls into Breakpad code here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/google-breakpad/src/common/windows/http_upload.cc#65

which uses WinINet APIs to do most of the hard work. If you look near the bottom of that function you can see that it does require a HTTP 200 response code for success, but it doesn't look like it cares about the response content-type.

For OS X the submission code is here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/client/crashreporter_osx.mm#555

It uses Cocoa APIs to do the real work. It also checks for HTTP status 200 for success.

For Linux the submission code is here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/client/crashreporter_gtk_common.cpp#190

which calls into Breakpad code here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/google-breakpad/src/common/linux/http_upload.cc#57

which calls into libcurl to do the work. It's a little hard for me to read, but it sets CURLOPT_FAILONERROR, which says it will only fail if the server returns a HTTP response code of 400 or higher, I believe.

For Android the submission code is here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/mobile/android/base/java/org/mozilla/gecko/CrashReporter.java#356

which uses Java APIs. The Android client *does* gzip-compress the request body, and it also looks like it checks for HTTP 200 (HttpURLConnection.HTTP_OK).

For the in-browser case, the submission code is here: https://dxr.mozilla.org/mozilla-central/rev/8d0aadfe7da782d415363880008b4ca027686137/toolkit/crashreporter/CrashSubmit.jsm#253

It uses XMLHttpRequest to submit, and it checks for HTTP status 200. I do note that it uses *responseText* on the XHR, so I'd have to read the XHR spec to see if that would break if the content-type of the response changed.

How do reports get to the collector?

Breakpad reports are submitted to a collector over HTTP.

Things to know about the HTTP POST request:

1. Incoming reports can be gzip compressed.

   This is particularly important for mobile.

2. The entire crash report and metadata is in the request body.

   Note that some of the information is duplicated in querystring variables to make logging and debugging easier.

3. HTTP POST request body is multi-part form data.

4. HTTP POST request body has previously had problems with null bytes and non-utf-8 characters. They've taken great pains to make sure it contains correct utf-8 characters.

   Still a good idea to do a pass on removing null bytes.

5. Content-length for HTTP POST request.

   TODO: Go through all the existing collector code to see if it *always* uses a Content-Length to determine the end of the data.

6. Crash reports can contain instructions on throttling.

   Crash report can contain:

   ```
   Throttleable=0
   ```

   If that's there and 0, then it should skip the throttler and be accepted, saved and processed.

   https://dxr.mozilla.org/mozilla-central/source/toolkit/crashreporter/CrashSubmit.jsm#282

7. Crash reports can contain a crash id.

   Crash report can contain:

   ```
   crash_id=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   ```

   We siphon crashes from our prod environment to our dev environment. We want these crash reports to end up with the same crash id. Thus it's possible for an incoming crash to have a crash id in the data.

   If it does have a crash id, we should use that.

Things to know about the HTTP POST response:

1. The HTTP POST response status code should be HTTP 200 if everything was fine.
2. Content-type for HTTP POST response can be anything, but `text/plain` is probably prudent.
3. HTTP POST response body should look like this:

   ```
   CrashID=bp-28a40956-d19e-48ff-a2ee-19a932160525
   ```

Testing breakpad crash reporting

When working on Antenna, it helps to be able to send real live crashes to your development instance. There are a few options:

1. Use Antenna's tools to send a fake crash:

   ```
   $ make shell
   app@c392a11dbfec:/app$ python -m testlib.mini_poster --url URL
   ```

2. Use an addon/webextension:

   - **Firefox < 57:** https://addons.mozilla.org/en-US/firefox/addon/crash-me-now-simple/
   - **Firefox >= 59 nightly/beta:** https://github.com/rhelmer/webext-experiment-crashme

3. Use Firefox and set the `MOZ_CRASHREPORTER_URL` environment variable:

   https://developer.mozilla.org/en-US/docs/Environment_variables_affecting_crash_reporting

   Then kill the Firefox process using the `kill` command.

(a) Run `ps -aef | grep firefox`. That will list all the Firefox processes.

Find the process id of the Firefox process you want to kill.

- main process looks something like `/usr/bin/firefox`
- content process looks something like `/usr/bin/firefox -contentproc -childID ...`

(b) The `kill` command lets you pass a signal to the process. By default, it passes `SIGTERM` which will kill the process in a way that doesn't launch the crash reporter.

You want to kill the process in a way that *does* launch the crash reporter. I've had success with `SIGABRT` and `SIGFPE`. For example:

- `kill -SIGABRT <PID>`
- `kill -SIGFPE <PID>`

What works for you will depend on the operating system and version of Firefox you're using.

Capturing an HTTP POST payload for a crash report

The HTTP POST payload for a crash report is sometimes handy to have. You can capture it this way:

1. Run `nc -l localhost 8000 > http_post.raw` in one terminal.
2. Run `MOZ_CRASHREPORTER_URL=http://localhost:8000/submit firefox` in a second terminal.
3. Kill a Firefox process using one of the methods in *Testing breakpad crash reporting*.
4. The Firefox process will crash and the crash report dialog will pop up. Make sure to submit the crash, then click on "Quit Firefox" button.

   That will send the crash to `nc` which will pipe it to the file.
5. Wait 30 seconds, then close the crash dialog window.

   You should have a raw HTTP POST in `http_post.raw`.

Project docs:

Contributing

Issues

Bugs and feature requests are tracked in Bugzilla.

Write up a new bug.

When writing up a bug, it helps to know the version of Antenna you're using.

Code conventions

All code files need to start with the MPLv2 header:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
```

PEP8 is nice. To lint your code, do:

```
$ make lint
```

If you hit issues, use `# noqa`.

Docker

Everything runs in a Docker container. Thus Antenna requires fewer things to get started and you're guaranteed to have the same setup as everyone else and it solves some other problems, too.

If you're not familiar with Docker and docker-compose, it's worth reading up on.

Documentation

Documentation for Antenna is build with Sphinx and is available on ReadTheDocs. API is automatically extracted from docstrings in the code.

To build the docs, run this:

```
$ make docs
```

Testing

To run the tests, run this:

```
$ make test
```

Tests go in `tests/`. Data required by tests goes in `tests/data/`.

If you need to run specific tests or pass in different arguments, you can run bash in the base container and then run `py.test` with whatever args you want. For example:

```
$ make shell
app@...$ py.test

<pytest output>

app@...$ py.test tests/unittest/test_crashstorage.py
```

We're using py.test for a test harness and test discovery.

Antenna Project specification: v1

Contents

Author Will Kahn-Greene

Last edited August 28th, 2017

History

https://github.com/mozilla-services/antenna/commits/master/docs/spec_v1.rst

Background

Socorro is the crash ingestion pipeline for Mozilla's products including Firefox, Firefox for Android, and others.

When Firefox crashes, the Breakpad crash reporter collects data, generates the crash payload, and sends it to Socorro via an HTTP POST.

The Socorro collector handles the incoming HTTP POST, extracts the crash from the HTTP request payload, saves all the parts to AWS S3, and then tells the processor to process the crash.

There are several problems with the current state of things:

1. Our current and future infrastructures don't work well with the "multiple separate components in the same repository" structure we currently have. When we do a deployment, we have to deploy and restart everything even if the component didn't change.
2. The different components have radically different uptime requirements.
3. The different components have radically different risk profiles and permissions requirements.

For these reasons, we want to extract the Socorro collector into a separate project in a separate repository with separate infrastructure.

Requirements for Antenna

Requirements for v1 of antenna:

1. Handle incoming HTTP POST requests on `/submit`
   - Handle gzip compressed HTTP POST request bodies.
   - Parse `multipart/form-data` into a raw crash.

   Antenna should parse HTTP payloads the same way that Socorro collector currently does.

   HTTP payloads are large:

   - average: 500kb
   - 95%: 1.5mb
   - max: 3mb
2. Throttle the crash
   - Examine the crash and apply throttling rules to it.
   - "Accepted" crashes are saved and processed.
   - "Deferred" crashes are saved, but not processed.
   - Rejected crashes get dropped.

   Throttling system and rules should match what Socorro collector currently does.

   Note: At some point, we could/should move this to the processor, but we can't easily do that without also changing the processor at the same time. To reduce the scope of this project, we're going to keep throttling in Antenna and then rewrite the processor and then maybe move throttling.

   ---
3. Generate a crash id and return it to the breakpad client.
   - Generate a crash id using the same scheme we're currently using.
   - Return the crash id to the client so that it can generate urls in about:crashes.
4. Add collector-generated bits to the crash report.
   - Add `uuid`, `dump_names`, `timestamp`, `submit_timestamp`, `legacy_processing` and `percentage` fields to raw crash.
5. Return crash id to client
   - This ends the HTTP session, so we want to get to this point as soon as possible.
6. Upload crash report files to S3
   - Use the same S3 "directory" scheme we're currently using.
   - Keep trying to save until all files are successfully saved.
   - Saving the raw crash file to S3 will trigger an AWS Lambda function to notify the processor of the crash to process.
7. Support Ops Dockerflow status endpoints
   - `/__version__`
   - `/__heartbeat__`
   - `/__lbheartbeat__`

8. Support Ops logging requirements
    - Use the new logging infrastructure.
9. Support Ops statsd for metrics
    - Use Datadog.

Crash reports and the current collector

Crash reports come in via `/submit` as an HTTP POST.

They have a `multipart/form-data` content-type.

The payload (HTTP POST request body) may or may not be compressed. If it's compressed, then we need to uncompress it.

The payload has a bunch of key/val pairs and also one or more binary parts.

Binary parts have filenames related to the dump files on the client's machine and `application/octet-stream` content-type.

The uuid and dump names are user-provided data and affect things like filenames and s3 pseudo-filenames. They should get sanitized.

Possible binary part names:

- `memory_report`
- `upload_file_minidump`
- `upload_file_minidump_browser`
- `upload_file_minidump_content`
- `upload_file_minidump_flash1`
- `upload_file_minidump_flash2`

Some of these come from `.dmp` files on the client computer.

Thus an HTTP POST something like this (long lines are wrapped for easier viewing):

```
Content-Type: multipart/form-data; boundary=------------------------c4ae5238
f12b6c82

--------------------------c4ae5238f12b6c82
Content-Disposition: form-data; name="Add-ons"

ubufox%40ubuntu.com:3.2,%7B972ce4c6-7e08-4474-a285-3208198ce6fd%7D:48.0,loop
%40mozilla.org:1.4.3,e10srollout%40mozilla.org:1.0,firefox%40getpocket.com:1
.0.4,langpack-en-GB%40firefox.mozilla.org:48.0,langpack-en-ZA%40firefox.mozi
lla.org:48.0
--------------------------c4ae5238f12b6c82
Content-Disposition: form-data; name="AddonsShouldHaveBlockedE10s"

1
--------------------------c4ae5238f12b6c82
Content-Disposition: form-data; name="BuildID"

20160728203720
--------------------------c4ae5238f12b6c82
Content-Disposition: form-data; name="upload_file_minidump"; filename="6da34
```

```
99e-f6ae-22d6-1e1fdac8-16464a16.dmp"
Content-Type: application/octet-stream

<BINARY CONTENT>
--------------------------c4ae5238f12b6c82--

etc.

--------------------------c4ae5238f12b6c82--
```

Which gets converted to a `raw_crash` like this:

```
{
    'dump_checksums': {
        'upload_file_minidump': 'e19d5cd5af0378da05f63f891c7467af'
    },
    'uuid': '00007bd0-2d1c-4865-af09-80bc02160513',
    'submitted_timestamp': '2016-05-13T00:00:00+00:00',
    'timestamp': 1315267200.0',
    'type_tag': 'bp',
    'Add-ons': '...',
    'AddonsShouldHaveBlockedE10s': '1',
    'BuildID': '20160728203720',
    ...
}
```

Which ends up in S3 like this:

```
v2/raw_crash/000/20160513/00007bd0-2d1c-4865-af09-80bc02160513

    Raw crash in serialized in JSON.

v1/dump_names/00007bd0-2d1c-4865-af09-80bc02160513

    Map of dump_name to file name serialized in JSON.

v1/dump/00007bd0-2d1c-4865-af09-80bc02160513

    Raw dump.
```

Crash ids

The Socorro collector generates crash ids that look like this:

```
de1bb258-cbbf-4589-a673-34f800160918
                              ^^^^^^^
                              ||____|
                              |  yymmdd
                              |
                              depth
```

The "depth" is used by `FSRadixTreeStorage` to figure out how many octet directories to use. That's the only place depth is used and Mozilla doesn't use `FSRadixTreeStorage` or any of its subclasses after the collector.

Antenna will (ab)use this character to encode the throttle result so that the lambda function listening to S3 save events knows which crashes to put in the processing queue just by looking at the crash id. Thus a crash id in Antenna looks like this:

```
de1bb258-cbbf-4589-a673-34f800160918
                             ^^^^^^^
                             ||____|
                             |  yymmdd
                             |
                             throttle result
```

where "throttle result" is either 0 for ACCEPT (save and process) or 1 for DEFER (save).

One side benefit of this is that we can list the contents of a directory in the bucket and know which crashes were slated for processing and which ones weren't by looking at the crash id.

Throttling

We were thinking of moving throttling to the processor, but in the interests of reducing the amount of work on other parts of Socorro that we'd have to land in lockstep with migrating to Antenna, we're going to keep the throttler in Antenna for now.

We should take the existing throttler code, clean it up and use that verbatim.

One thing we're going to change is that we're not going to specify throttling rules in configuration. Instead, we'll specify a Python dotted path to the variable holding the throttling rules which will be defined as Python code. That makes it wayyyyyy easier to write, review, verify correctness and maintain over time.

Logging

We'll use the new logging infrastructure. Antenna will use the Python logging system and log to stdout and that will get picked up by the node and sent to the logging infrastructure.

Metrics

Antenna will use the Datadog Python library to generate stats. These will be collected by the dd-agent on the node and sent to Datadog.

Test plan

flake8

Antenna will have a linter set up to lint the code base.

This will be run by developers and also run by CI for every pull request and merge to master.

This will help catch:

- silly mistakes, typos, and so on
- maintainability issues like code style, things to avoid in Python, and so on

`tests/unittest/`

Antenna will have a set of unit tests and integration tests in the repository alongside the code that will cover critical behavior for functions, methods, and classes in the application.

These will be written in pytest.

These will be run by developers and also run by CI for every pull request and merge to master.

This will help catch:

- bugs in the software
- regressions in behavior

`tests/systemtest/`

Antenna will have a system test that verifies node configuration and behavior.

This is critical because we don't want to put a dysfunctional or misconfigured node in service. If we did, that will cause us to lose crashes sent to that node because it may not be able to save them to S3.

Nothing is mocked in these tests–everything is live.

This can be run by the developer. This will be run on every node during a deployment before the node is put in service.

This will help catch:

- configuration issues in the server environment
- permission issues for saving data to to S3
- bugs in the software related to running in a server environment

loadtest

We want to run load tests on a single node as well as a scaling cluster of nodes to determine:

1. Is Antenna roughly comparable to the Socorro collector it is replacing in regards to resource usage under load?
2. How does a single node handle increasing load? At what point does the node fall down? What is the performance behavior for a node under load in regards to CPU, memory usage, disk usage, network up/down, and throughput.
3. How does a cluster of nodes handle increasing load? Does the system spin up new nodes effectively? Do the conditions for scaling up and down work well for the specific context of the Antenna application?
4. How does Antenna handle representative load? How about 3x load? How about 10x load?
5. How does Antenna handle load over a period of time?

This then informs us whether we need to make changes and what kind of changes we should make.

We'll do two rounds of load testing. The first round is a "lite" round just to get us rough answers for basic performance questions.

https://github.com/willkg/antenna-loadtests/tree/antenna-loadtest-lite

Second round will be run multiple times and will be more comprehensive.

https://github.com/mozilla-services/antenna-loadtests

We'll use this load test system going forward whenever we make substantial changes that might impact performance.

Research and decisions

nginx like telemetry edge vs. python architecture thoughts

The current collector has a web process that:

1. handles incoming HTTP requests
2. converts the multipart/form-data HTTP payload into two JSON documents (`raw_crash` and `dump_names`) and one binary file for each dump
3. throttles the crash based on configured rules
4. generates a crash id and returns it to the breakpad client
5. saves the crash report data files to local disk

Then there's a crashmover process that runs as a service on the same node and:

1. uploads crash report data files to S3
2. adds a message to RabbitMQ with the crashid telling the processor to process that crash
3. sends some data to statsd

My first collector rewrite (June 2016-ish) folded the web and crashmover processes into a single process using asyncio and an eventloop so that we could return the crash id to the client as quickly as possible, but continue to do the additional work of uploading to S3 and notifying RabbitMQ. This also has the nicety that we don't have to use the disk to queue crash reports up and theoretically we could run this on Heroku[1].

My second collector (August 2016-ish) rewrite merely extracted the collector bits from the existing Socorro code base. I did this attempt figuring it was the fastest way to extract the collector. However, it left us with two processes. I abandoned this one, too.

In August 2016, I traded emails with Mark Reid regarding the Telemetry edge which serves roughly the same purpose as the Socorro collector. At the time, they had a heka-based edge but were moving to an nginx-based one called nginx_moz_ingest. The edge sends incoming payloads directly to Kafka.

The edge looked interesting, but there are a few things that Socorro needs currently that the edge doesn't do:

1. Socorro needs to generate and return a CrashID
2. Socorro needs to convert the multipart/form-data payload into two JSON documents (`raw_crash` and `dump_names`) and one binary file for each dump
3. Socorro has large crash reports and needs to save to S3
4. Socorro currently throttles crashes in the collector
5. Socorro currently uses RabbitMQ to queue crashes up for processing

In September 2016 at the work week, I talked with Rob Helmer about this and he suggested we build it all in nginx using modules similar to what Telemetry did. He has a basic collector that generates a uuid and saves the crash report to disk[2]. We could use a uuid module and then tweak the outcome of that with the date.

We could move the throttling to the processor. This is tricky because it means we're making changes to multiple components at the same time which greatly increases the scope of the project.

At the work week, we decided we can't just send crash payloads to Kafka because we get too many of them and they're too large.

We could use an nginx S3 upload module to upload it to S3. We had some concerns about the various S3 failure scenarios and how to deal with those and how doing everything as an nginx module makes that more tricky. We could instead have nginx save it to disk and have a service using inotify notice it on disk and then upload it to S3.

We could push converting the payload from multipart/form-data to a series of separate files to the processor, but that heavily affects the processor, the webapp, and possibly a bunch of other tools.

We could write a lua module for converting in nginx, but that's more work to do.

[1] Heroku can run docker containers now, so it's probably the case we don't have to worry about the "only one process!" thing anymore.
[2] Rob's gist: https://gist.github.com/rhelmer/00dd0f9e4076260078367f763bc9aaf3

Given all that, my current thinking is that we've got the following rough options:

1. This is a doable project using nginx, c, lua, and such and follow what Telemetry did with the edge, but there are a lot of differences.

   Doing that will likely give us a collector that's closer to the Telemetry collector which is nice.

   There are a decent number of things we'd have to figure out how to do in a way that mirrors the current collector or this project becomes a lot bigger since it'd also involve making changes to the processor, webapp, and any thing that uses the raw crash data.

   The current Socorro team has zero experience building nginx modules or using lua. It'd take time to level up on these things. Will's done some similar-ish things and we could use what Rob and Telemetry have built. Still, we have no existing skills here and I suggest this makes it more likely for it to take "a long time" to design, implement, review, test, and get to prod.

2. This is a doable project using Python. Doing that will likely give us a collector that has a lifetime of like 2 years, thus it's a stopgap between now and whatever the future holds.

   We could use Python 2 which expires in a couple of years.

   We could use Python 3 which reduces the compelling need to rewrite it in Python 3 later.

   We can't use Python 3's asyncio because the things we need like boto don't support it, yet.

   We could use gevent which lets us do asynchronous I/O and has an event loop.

   This is just like one of the earlier collector rewrites I was working on (Antenna). The current Socorro team has experience in this field. Further, we've reduced the requirements from the original collector, it'd probably take "a short time" to design, implement, review, test and push to prod.

   After rewriting the collector, we plan to extract/rewrite other parts of Socorro. After that work is done, it should be a lot easier to make chances to components and change how data flows through the system and what shape it's in.

   After that, we would be in a much better place to switch to something like the Telemetry edge.

Given that, I'm inclined to go the Python route. At some point it may prove to be an unenthusing decision, but I don't think the risks are high enough that it'll ever be a **wrong** decision.

WSGI framework thoughts

We wanted to use a framework with the following properties:

1. good usage, well maintained, good docs
2. minimal magic
3. minimal dependencies
4. no db
5. easy to write tests against
6. works well with gunicorn and gevent

I spent a few days looking at CherryPy, Flask, Bottle and Falcon. I wrote prototypes in all of them that used gunicorn and gevent.

Here's my unscientific hand-wavey summaries:

- CherryPy

  We were using it already, so I figured it was worth looking at. It's nice, but there's a lot of it and I decided I liked Falcon better.

- Flask

  It's well used, I'm familiar with it, we use it in other places at Mozilla. But it includes Jinja2 and a ton of other dependencies and there's some magic (thread-local vars, etc).

- Bottle

  I didn't like Bottle at all. It's in one massive file and just didn't appeal to me at all.

- Falcon

  Falcon had all the properties I was looking for. It's nice and was easy to implement the things I wanted to in the prototype.

I decided to go with Falcon.

We should write the code in such a way that if we decide to switch to something else, it's not a complete rewrite.

gevent thoughts

Falcon lists "works great with async libraries like gevent" as a feature, so it should be fine.

- http://falcon.readthedocs.io/en/stable/index.html?highlight=gevent#features

While looking into whether boto supported Python 3's asyncio, I read several comments in their issue tracker from people who use boto with gevent without problems. Interestingly, the boto2 issue tracker has some open issues around gevent, but the boto3 issue tracker has none. From that anecdata, I think we're probably fine with boto.

- https://github.com/gevent/gevent/issues/535#issuecomment-162565389
- https://github.com/boto/boto/issues?utf8=%E2%9C%93&q=is%3Aissue%20is%3Aopen%20gevent
- https://github.com/boto/boto3/issues?utf8=%E2%9C%93&q=is%3Aissue%20is%3Aopen%20gevent

I've heard reports that there are problems with New Relic and gevent, but nothing recent enough to discount the "it's probably fixed by now" possibilities. Combing their forums suggests some people have problems, but each one seems to be fixed or alleviated.

- https://discuss.newrelic.com/search?q=gevent

I feel pretty confident that we'll be fine using gevent. A system test and a load test might tell us more.

Lonnen brought up this article from the Netflix blog where they had problems switching to async i/o with Zuul 2 which is Java-based:

http://techblog.netflix.com/2016/09/zuul-2-netflix-journey-to-asynchronous.html

There's a lot of big differences between their project and ours. Still, we should give some thought to alleviating the complexities of debugging event-driven code and making sure all the libs we use are gevent-friendly.

boto2 vs. boto3

According to the boto documentation, boto3 is stable and recommended for daily use.

- boto2: http://boto.cloudhackers.com/en/latest/
- boto3: https://github.com/boto/boto3

Socorro uses boto2. I think we'll go with boto3 because it's the future.

S3 and bucket names

AWS Rules for bucket names:

http://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html

Note that they do suggest using periods in bucket names in the rules.

S3 REST requests:

http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAPI.html

Note, they talk about two styles:

- "virtual hosted-style request" which is like `http://examplebucket.s3-us-west-2.amazonaws.com/puppy.jpg`
- "path-style request" which is like `http://s3-us-west-2.amazonaws.com/examplebucket/puppy.jpg`

Path-style requires that you use the region-specific endpoint. You'll get an HTTP 307 if you try to access a bucket that's not in US east if you use endpoints `http://s3.amazonaws.com` or an endpoint for a different region than where the bucket resides.

In the page on virtual hosted-style requests:

http://docs.aws.amazon.com/AmazonS3/latest/dev/VirtualHosting.html

they say:

> When using virtual hosted–style buckets with SSL, the SSL wild card certificate only matches buckets that do not contain periods. To work around this, use HTTP or write your own certificate verification logic.

Socorro currently uses `boto.s3.connect_to_region` and `boto.s3.connection.OrdinaryCallingFormat`. Buckets are located in us-west-2.

Boto3 changes the API around. Instead of calling it "calling_format", they call it "addressing_style".

From that I conclude the following:

1. In order to support the s3 buckets we currently have and use SSL, we need to continue using path-style requests and specify the region.
2. With boto3, this means specifying the `region_name` when creating the session client. I'll have to figure out what the default for `addressing_style` is and if it's not what we want, how to change it.
3. In the future, we shouldn't use dotted names–it doesn't seem like a big deal, but it'll probably make things easier.

I think that covers the open questions we had for the s3 crash store in Antenna.

CHAPTER 2

Indices and tables

- genindex
- search

Index