MANTLE

Warning: Mantle is for research and development of metadata balancer algorithms, not for use on production CephFS clusters.

Multiple, active MDSs can migrate directories to balance metadata load. The policies for when, where, and how much to migrate are hard-coded into the metadata balancing module. Mantle is a programmable metadata balancer built into the MDS. The idea is to protect the mechanisms for balancing load (migration, replication, fragmentation) but stub out the balancing policies using Lua. Mantle is based on [1] but the current implementation does *NOT* have the following features from that paper:

1. Balancing API: in the paper, the user fills in when, where, how much, and load calculation policies; currently, Mantle only requires that Lua policies return a table of target loads (e.g., how much load to send to each MDS)
2. “How much” hook: in the paper, there was a hook that let the user control the fragment selector policy; currently, Mantle does not have this hook
3. Instantaneous CPU utilization as a metric

[1] Supercomputing ‘15 Paper: http://sc15.supercomputing.org/schedule/event_detail-evid=pap168.html

QUICKSTART WITH VSTART

Warning: Developing balancers with vstart is difficult because running all daemons and clients on one node can overload the system. Let it run for a while, even though you will likely see a bunch of lost heartbeat and laggy MDS warnings. Most of the time this guide will work but sometimes all MDSs lock up and you cannot actually see them spill. It is much better to run this on a cluster.

As a pre-requistie, we assume you have installed mdtest or pulled the Docker image. We use mdtest because we need to generate enough load to get over the MIN_OFFLOAD threshold that is arbitrarily set in the balancer. For example, this does not create enough metadata load:

```
while true; do
  touch "/cephfs/blah-`date`"
done
```

MANTLE WITH VSTART.SH

1. Start Ceph and tune the logging so we can see migrations happen:

```
cd build
../src/vstart.sh -n -l
for i in a b c; do
  bin/ceph --admin-daemon out/mds.$i.asok config set debug_ms 0
  bin/ceph --admin-daemon out/mds.$i.asok config set debug_mds 2
  bin/ceph --admin-daemon out/mds.$i.asok config set mds_beacon_grace 1500
done
```

2. Put the balancer into RADOS:

```
bin/rados put --pool=cephfs_metadata_a greedyspill.lua ../src/mds/balancers/greedyspill.lua
```

3. Activate Mantle:

```
bin/ceph fs set cephfs max_mds 5
bin/ceph fs set cephfs_a balancer greedyspill.lua
```

4. Mount CephFS in another window:

```
  bin/ceph-fuse /cephfs -o allow_other &
  tail -f out/mds.a.log


Note that if you look at the last MDS (which could be a, b, or c -- it's
random), you will see an an attempt to index a nil value. This is because the
last MDS tries to check the load of its neighbor, which does not exist.
```

5. Run a simple benchmark. In our case, we use the Docker mdtest image to create load:

```
for i in 0 1 2; do
  docker run -d \
    --name=client$i \
    -v /cephfs:/cephfs \
    michaelsevilla/mdtest \
    -F -C -n 100000 -d "/cephfs/client-test$i"
done
```

6. When you are done, you can kill all the clients with:

```
for i in 0 1 2 3; do docker rm -f client$i; done
```

OUTPUT

Looking at the log for the first MDS (could be a, b, or c), we see that everyone has no load:

```
2016-08-21 06:44:01.763930 7fd03aaf7700  0 lua.balancer MDS0: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:44:01.763966 7fd03aaf7700  0 lua.balancer MDS1: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:44:01.763982 7fd03aaf7700  0 lua.balancer MDS2: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:44:01.764010 7fd03aaf7700  2 lua.balancer when: not migrating! my_load=0.0 hisl
2016-08-21 06:44:01.764033 7fd03aaf7700  2 mds.0.bal  mantle decided that new targets={}
```

After the jobs starts, MDS0 gets about 1953 units of load. The greedy spill balancer dictates that half the load goes to your neighbor MDS, so we see that Mantle tries to send 1953 load units to MDS1.

```
2016-08-21 06:45:21.869994 7fd03aaf7700  0 lua.balancer MDS0: < auth.meta_load=5834.188908912
2016-08-21 06:45:21.870017 7fd03aaf7700  0 lua.balancer MDS1: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:45:21.870027 7fd03aaf7700  0 lua.balancer MDS2: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:45:21.870034 7fd03aaf7700  2 lua.balancer when: migrating! my_load=1953.3492228
2016-08-21 06:45:21.870050 7fd03aaf7700  2 mds.0.bal  mantle decided that new targets={0=0,1=
2016-08-21 06:45:21.870094 7fd03aaf7700  0 mds.0.bal    - exporting [0,0.52287 1.04574] 1030.
2016-08-21 06:45:21.870151 7fd03aaf7700  0 mds.0.migrator nicely exporting to mds.1 [dir 1000
```

Eventually load moves around:

```
2016-08-21 06:47:10.210253 7fd03aaf7700  0 lua.balancer MDS0: < auth.meta_load=415.7741430044
2016-08-21 06:47:10.210277 7fd03aaf7700  0 lua.balancer MDS1: < auth.meta_load=228.7202397769
2016-08-21 06:47:10.210290 7fd03aaf7700  0 lua.balancer MDS2: < auth.meta_load=0.0 all.meta_l
2016-08-21 06:47:10.210298 7fd03aaf7700  2 lua.balancer when: not migrating! my_load=415.7900
2016-08-21 06:47:10.210311 7fd03aaf7700  2 mds.0.bal  mantle decided that new targets={}
```

IMPLEMENTATION DETAILS

Most of the implementation is in MDBalancer. Metrics are passed to the balancer policies via the Lua stack and a list of loads is returned back to MDBalancer. It sits alongside the current balancer implementation and it's enabled with a Ceph CLI command ("ceph fs set cephfs balancer mybalancer.lua"). If the Lua policy fails (for whatever reason), we fall back to the original metadata load balancer. The balancer is stored in the RADOS metadata pool and a string in the MDSMap tells the MDSs which balancer to use.

EXPOSING METRICS TO LUA

Metrics are exposed directly to the Lua code as global variables instead of using a well-defined function signature. There is a global “mds” table, where each index is an MDS number (e.g., 0) and each value is a dictionary of metrics and values. The Lua code can grab metrics using something like this:

```
mds[0]["queue_len"]
```

This is in contrast to cls-lua in the OSDs, which has well-defined arguments (e.g., input/output bufferlists). Exposing the metrics directly makes it easier to add new metrics without having to change the API on the Lua side; we want the API to grow and shrink as we explore which metrics matter. The downside of this approach is that the person programming Lua balancer policies has to look at the Ceph source code to see which metrics are exposed. We figure that the Mantle developer will be in touch with MDS internals anyways.

The metrics exposed to the Lua policy are the same ones that are already stored in mds_load_t: auth.meta_load(), all.meta_load(), req_rate, queue_length, cpu_load_avg.

COMPILE/EXECUTE THE BALANCER

Here we use *lua_pcall* instead of *lua_call* because we want to handle errors in the MDBalancer. We do not want the error propagating up the call chain. The cls_lua class wants to handle the error itself because it must fail gracefully. For Mantle, we don’t care if a Lua error crashes our balancer – in that case, we will fall back to the original balancer.

The performance improvement of using *lua_call* over *lua_pcall* would not be leveraged here because the balancer is invoked every 10 seconds by default.

RETURNING POLICY DECISION TO C++

We force the Lua policy engine to return a table of values, corresponding to the amount of load to send to each MDS. These loads are inserted directly into the MDBalancer “my_targets” vector. We do not allow the MDS to return a table of MDSs and metrics because we want the decision to be completely made on the Lua side.

Iterating through tables returned by Lua is done through the stack. In Lua jargon: a dummy value is pushed onto the stack and the next iterator replaces the top of the stack with a (k, v) pair. After reading each value, pop that value but keep the key for the next call to *lua_next*.

READING FROM RADOS

All MDSs will read balancing code from RADOS when the balancer version changes in the MDS Map. The balancer pulls the Lua code from RADOS synchronously. We do this with a timeout: if the asynchronous read does not come back within half the balancing tick interval the operation is cancelled and a Connection Timeout error is returned. By default, the balancing tick interval is 10 seconds, so Mantle will use a 5 second second timeout. This design allows Mantle to immediately return an error if anything RADOS-related goes wrong.

We use this implementation because we do not want to do a blocking OSD read from inside the global MDS lock. Doing so would bring down the MDS cluster if any of the OSDs are not responsive – this is tested in the ceph-qa-suite by setting all OSDs to down/out and making sure the MDS cluster stays active.

One approach would be to asynchronously fire the read when handling the MDS Map and fill in the Lua code in the background. We cannot do this because the MDS does not support daemon-local fallbacks and the balancer assumes that all MDSs come to the same decision at the same time (e.g., importers, exporters, etc.).

DEBUGGING

Logging in a Lua policy will appear in the MDS log. The syntax is the same as the cls logging interface:

```
BAL_LOG(0, "this is a log message")
```

It is implemented by passing a function that wraps the *dout* logging framework (*dout_wrapper*) to Lua with the *lua_register()* primitive. The Lua code is actually calling the *dout* function in C++.

Warning and Info messages are centralized using the clog/Beacon. Successful messages are only sent on version changes by the first MDS to avoid spamming the *ceph -w* utility. These messages are used for the integration tests.

TESTING

Testing is done with the ceph-qa-suite (tasks.cephfs.test_mantle). We do not test invalid balancer logging and loading the actual Lua VM.