Google Summer of Code 2017

PROPOSAL

ORGANISATION : CEPH

PROJECT :

“CEPH-MGR : SMARTER REWEIGHT BY UTILISATION”

MENTOR : KEFU CHAI

About Me

Basic Information

Name	Spandan Kumar Sahu
University	IIT Kharagpur
Major	Computer Science and Engineering
Email	spandankumarsahu@gmail.com, spandankumarsahu@iitkgp.ac.in
Timezone	IST (UTC +5:30)
Contact No	+91 7477858944
Github Profile	https://github.com/SpandanKumarSahu

Background

I am a second year (fourth semester) undergraduate student of Computer Science and Engineering department in Indian Institute of Technology, Kharagpur. I am familiar with languages like C, C++, Python and JAVA, as a part of several course works and projects. Additionally, as required by the project, I have undertaken Probability and Statistics course this semester (scheduled to be complete before 23rd April, 2017). I am equally versed with using git.

I am a software team member of Swarm Robotics, IIT Kharagpur, a robotics research group that focuses to create distributed systems for robots. My first stint with Ceph was when I was exploring distributed storage systems for the Swarm Robotics, in order to avoid data loss in certain cases. Google Summer of Code provides me with a unique and exciting opportunity to understand the working of Ceph in and out by contributing to it.

CEPH Architecture

Ceph works on CRUSH algorithm which is a variant of consistent hashing. The CRUSH algorithm distributes data objects among storage devices according to a per-device weight value, approximating a uniform probability distribution. The distribution is controlled by a hierarchical cluster map representing the available storage resources and composed of the logical elements from which it is built[1]. The OSDs (Object Storage Devices) in Ceph are initially weighted in proportion to their hardware capabilities, like total storage capability, write/access times etc. This is how Ceph is able to maintain a statistically balanced utilization of storage and bandwidth resources.

A complete Ceph implementation is likely to consist of many Ceph storage clusters. Ceph Storage Clusters consist of two types of daemons: a *Ceph OSD Daemon* (OSD) stores data as objects on a storage node; and a *Ceph Monitor* (MON) maintains a master copy of the cluster map.
A Ceph storage cluster consists of multiple pools. Each pool can have many (usually a power of 2) number of Placement Groups (PGs) that are served by a given set of OSDs. A pool has a resilience (number of OSDs are allowed to fail without loss of data) and CRUSH rules, that enables CRUSH to identify a rule for the placement of objects and its replicas. We can also set ownership levels for Pools.
A bucket is a collection of devices or other buckets, allowing them to form interior nodes in a storage hierarchy in which devices are always at the leaves. The cluster map is composed of devices and buckets, both of which have numerical identifiers and weight values associated with them.

Analysing the Problem

CRUSH maps contain a list of OSDs, a list of 'buckets' aggregating the devices into physical locations, and a list of rules that tell CRUSH how it should replicate data in a Ceph Cluster's pools. At the very beginning, the administrator is very likely specify a weight relative to the total capacity of the device. But over the time, the utilization of storage devices could become unbalanced with respect to the initial weights assigned. The discussions over the bug#15653[2] provides a good example how the imbalance occurs.

Existing/Attempted Solutions

A very naive way would be to rebalance, every time the distribution becomes imbalance beyond a limit. However, this would incur substantial performance degradation.

A better way would be a re-weight algorithm, which is based on the amount of utilisation of the Placement Groups or the OSDs. In this case, at each draw, the average utilisation is calculated, and each device is reweighted with a product of normalised average utilisation and the device's assigned weight. The reweight by utilisation of PGs, also has a similar algorithm. The problem with this, is that it is still not "accurate" enough, especially when there is more number of draws.[6]

Another work around, is using multivariate distributions to reweight the buckets, which has an added advantage of being further simplified for large sets, using Central Limit Theorem.[3] This was implemented, though not merged in the PR #10218 [4]. The reasons as mentioned in the comments by Sage Weil and Adam C. Emerson, was that the distributions did not converge well, especially if the objects had more than two weights. This is because,

"If you have two weights one could increase the higher weight a whole lot on the second and subsequent draws to try and keep closer to the desired distribution, but it would effectively be not drawing low-weight objects in the second draw. Low-weight objects would have a chance of being the first draw, their chance of appearing at all in the second draw would go down radically as we approached the desired distribution."

Proposed Solution

Overall Layout

In the current reweight-by-utilisation, when the OSDs are reweighted, there is actual transfer of data in between the disks. [15]. This is okay, if the number of writes is low, or there hasn't been any change in the CRUSH map, that we need to rebalance everything. I propose that, the we shall use the weights of the OSDs, as a guidance and we shall actually do `'ceph osd crush reweight {osd_name} {reweight_value}'` less frequently, and not at every write sequence, so as to decrease the overall network usage. This is because, in the current reweight_by_utilisation, due to frequent reweights, data moves in and out of OSDs, quite

frequently. Therefore, I propose to, decouple the weight of the OSDs in the CRUSH map and the weights that I shall be using in my proposed reweight algorithm. I would only do 'ceph osd crush reweight' after a significant number of write sequences. But, this might not produce desirable distribution for small number of write sequence, so I will allow for the reweight algorithm to take '--small' as a flag, in which case, it would actually do 'ceph osd crush reweight'.

Part 1(a): Evaluation metrics

Now, for the first part, as to creating a evaluation tool, most of the data shall be accessed from 'crushtool' and some data shall be accessed from other crush usage data like 'ceph osd stat' , 'iostat' for input/output related statistics.[15] The evaluation tool, would provide for two kind of analysis, one cluster analysis and the other OSDs level analysis. The various performance metrics, like usage data, network bandwidth usage, latency etc, would be evaluated against the benchmark statistics, and assigned a score on the basis of their deviation from the set benchmark. For this, I shall assume that the result of the analysis on the algorithm, produces a Gaussian distribution of the metrics, with mean and variance equal to that of the analysis for that particular metric. The signed deviation, from the benchmark value to the mean value, shall determine the score of the algorithm in that metric.

As an example, consider, the mean of the metric analysis for 'network bandwidth usage' was 0.5 and variance was 0.2. And let the benchmark for the same was 0.4. Now, I would consider a Gaussian (Normal) distribution ~ N(0.5, 0.2). Since, the metric is a negative metric, that is higher the value of network usage, the worse it is, there for the signed deviation will be (-1) * ($\phi(0.5) - \phi(0.4)$), where ϕ = cumulative distribution function (cdf) of the Normal distribution.

Part 1(b) : Assigning score

The total score of the algorithm shall be the sum of the scores of the algorithm across all the metrics, where the scores of the metrics shall be weighted. We can not assign equal weights to all the metrics, because some metrics will be more important for some cases than other metrics. So I propose for implementing only two case scenarios, one is efficient network usage mode, and the other being efficient data distribution mode. More case scenarios can be provided by assigning different weights to the metrics or by letting the user choose on which metrics to choose from and assign their weights. However, for the time being I shall implement only the two of it, with a hope, that if I shall be able to complete things on time, I will take this as

an additional goal. It is quite obvious that in the first case, higher weights will be given to network usage than others and in the second case, higher weights will be for data distribution on the disk.

Part 2(a) : The Proposed Algorithm

For the second part, the method I propose to solve this problem is to use an implementation of a feedback mechanism like PID. Here is a broad overview of how to achieve it:

- ★ The initial weight distribution would be set as “distribution to be achieved” and stored permanently in a file. The “distribution to be achieved” would be a map, which represents the weights of the devices in their hierarchical order (if any). (See Tree Buckets[1]).
- ★ The current weight distribution, would be representative of the actual weight distribution on the devices, again in their hierarchical order.
- ★ At each step, we shall calculate the “current difference” between the “distribution to be achieved” and the actual distribution and also maintain a “sum” of these differences. Each of the differences are calculated as weighted sum of signed individual differences in the expected and actual distributions. This way we can prioritize imbalances in the OSDs/buckets with greater weight, because such differences are more prominent and inflict greater performance degradation in comparison to those with imbalances in those with lower weights.
- ★ Let, the current difference be e_p, and e_{diff} equals $e_p(d)-e_p(d-1)$, where $e_p(d)$ is the “current error” after ‘d’ draws. Similarly, we maintain e_i, which is the summation over all e_p. We also maintain, constants K_p , K_i and K_d, which show relative weightage of e_p, e_i and e_{diff}.
- ★ We then calculate a total_error which equals $(K_p * e_p)+(K_i * e_i) + (K_d * e_{diff})$.
  We then reweight the “current weight distribution” by adding signed total_error scaled in proportion to the device’s initial weight, in the weights of individual devices and hence update the “current differences” accordingly. (for tree buckets, we simply propagate the changes up the hierarchy tree).
- ★ The toughest of the task is to find appropriate initial values for K_p , K_i and K_d, which can be handled using readily available Machine Learning algorithms, like gradient descent, though we need to have enough data to train the model, in beforehand.[5] The value of K_p, K_i and K_d should also be updated at each iteration.

This system is dynamic, because it not only takes into effect any imbalance that has been accumulated in the past, but also the size of the current draw along with the inevitable current deviation from the balanced state.

I will demonstrate the idea with an example.

1. Consider we've 5 OSDs with weights, having weights [10,10,10,10,1].
   PS : We keep the assigned weights as it is, and make a copy of it, which will be the actual weights we will be working with, and we use the normalised version of it. So, we maintain target_weight_distribution as [(10/41),(10/41),(10/41),(10/41),(1/41)], until the weights are changed.
2. At each 'step' or 'draw' we maintain:
   We need to keep the assigned weight so as to know what distribution we need to have, at all times.
   - current_load_distribution, which is [0,0,0,0,0] initially.
   - current_error in load distribution, which is [(10/41),(10/41),(10/41),(10/41),(1/41)] initially
   - derivative_error in load distribution, which is [0,0,0,0,0] initially.
   - integral_error in load distribution, which is [0,0,0,0,0] initially.
   - current_weight_distribution, which is [(10/41),(10/41),(10/41),(10/41),(1/41)] initially
3. We have certain constants (which I shall explain later, how to determine), named Kp, Ki and Kd which will be the coefficients of current_error, integral_error and derivative_error, in the expression:
   total_error=(Kp*current_error)+(Ki*integral_error)+(Kd*derivative_error)

4. At each 'step' or 'draw':
   current_error = current_load_distribution - target_weight_distribution
   derivative_error = current_error - previous_error
   integral_error = integral_error + current_error
   total_error = (Kp*current_error)+(Ki*integral_error)+(Kd*derivative_error) previous_error = current_error

   Now, since, imbalance for an OSD with greater weight is more significant than the one with lower weight, we need to scale them to their target_weight_distribution.

So, total_error = (total_error).(target_weight_distribution)
// This is simply multiplying the transpose of the vector target_weight_distribution with total_error.
Now, we need to re-assign the weights as :
current_weight_distribution = current_weight_distribution - total_error

Then we normalise current_weight distribution. And reweight with this.

As an example:

Let us assume, that the PID_Tuner function (explained below), returned
Kp=0.08, Ki=0.01, and Kd=0.01

I will be henceforth, using normalised weights.

Initially, target_weight_distribution : [0.2439, 0.2439, 0.2439, 0.2439, 0.0244]

At the first draw :
Probability of choosing OSD 'a' = 0.2439
After choosing 'a' :
current_load_distribuion : [1,0,0,0,0]
current_error : [0.7561, -0.2439, -0.2439, -0.2439, -0.0244]
integral_error : [0.7561, -0.2439, -0.2439, -0.2439, -0.0244]
derivative_error: [0.7561, -0.2439, -0.2439, -0.2439, -0.0244]
total_error=[0.0756, -0.0244, -0.0244, -0.0244, -0.0024]

total_error= total_error.(target_weight_distribution)
= [0.0184,-0.0059,-0.0059,-0.0059,-0.0006]
current_weight_distribution = [0.2439, 0.2439, 0.2439, 0.2439, 0.0244] - [0.0184,-0.0059,-0.0059,-0.0059,-0.0006]
= [0.2255, 0.2498, 0.2498, 0.2498, 0.0250]
= [0.2255, 0.2498, 0.2498, 0.2498, 0.0250] // After normalisation

Probability of choosing OSD 'e' = 0.0244
After choosing 'e' :

current_load_distribuion : [0,0,0,0,1]
current_error : [-0.2439, -0.2439, -0.2439, -0.2439, 0.9756]
integral_error : [-0.2439, -0.2439, -0.2439, -0.2439, 0.9756]
derivative_error: [-0.2439, -0.2439, -0.2439, -0.2439, 0.9756]
total_error = [-0.0244, -0.0244, -0.0244, -0.0244, 0.0975]

total_error= total_error.(target_weight_distribution)
= [-0.0059,-0.0059,-0.0059,-0.0059,0.0023]

current_weight_distribution = [0.2439, 0.2439, 0.2439, 0.2439, 0.0244] -
[-0.0059,-0.0059,-0.0059,-0.0059,0.0023]
= [0.2498, 0.2498, 0.2498, 0.2498, 0.0221]
= [0.2498, 0.2498, 0.2498, 0.2498, 0.0221] // After normalisation

Now, let us calculate the probability of
getting OSD 'e' = (0.0244) + (0.0236) = 0.0480
getting OSD 'a' = (0.2439) + (0.2360) + 0.0006 = 0.4805
We see that the expected ratio is maintained. Similarly, we can also show for higher degrees of num_rep, but the values of Kp, Ki and Kd must be appropriate.

Significance of each term :

1. current_error :
   This is the prime reason, we need to change the weight distribution, i.e. with the current_weight_distribution, we can't achieve the target_load_distribution.
2. integral_error :
   This is to ensure, if a device hasn't been chosen for a long time, this factor will increase the probability that it should have higher chances of getting selected in the subsequent turns. It is accounting for the differences in load distributions in the past.
3. derivative_error :
   This reduces the probability of the same device getting selected in successive fashion, more frequently than it should be.It is accounting for the possible differences in load distributions in future.

Part 2(b) : Reasons for the Algorithm

This algorithm rests on the principle that the values of Kp, Ki and Kd are dependent only the CRUSH algorithm, and hence are system independent.

This is explained by

" Given a single integer input value x, CRUSH will output an ordered list R of n distinct storage targets. CRUSH utilizes a strong multi-input integer hash function whose inputs include x, making the mapping completely deterministic and independently calculable using only the cluster map, placement rules, and x. " [2]

-- CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data by
Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Carlos Maltzahn (See Reference)

The value of Kp, Ki and Kd in a PID depends on the system or in better words, depends on "who takes the actions". Since it is the CRUSH algorithm who is responsible for the selection of devices and CRUSH is same for all the systems, therefore it becomes system independent.

Part 2(c) : Why I vouch for this algorithm?

1. This is dynamic. Dynamic in the sense that, it takes into account, the decisions the CRUSH has made in the past and prevent certain selections in the future.

   So, if the lower weighted device hasn't been selected for a long time, the integral_error for that device will keep on increasing, and it will increase the chances of it getting selected in the subsequent trials.

   Similarly, if the lower weighted device is once selected, it's weight is significantly reduced, till sufficient turns.
   This ensures that highly improbable situations where distribution is skewed to lower weighted devices.
2. We can convert the normalised distribution, back to integer weights (with some negligible error) and vice-versa.
3. All calculations are simple. Though, the distribution of objects in OSDs can be viewed as a Gaussian distribution, with mean proportional to the weight of the device, and standard

deviation proportional to the error in distribution that can be allowed for, but the calculations in that case would be very heavy and time consuming.

4. Though it has been explained for OSDs in buckets, this can be similarly and recursively applied to tree buckets. I am not sure, as how to apply for Straw buckets though.
5. It can account for any sudden change in the distribution.
   For example, if a device is down, the change in the object distribution can occur without any rebalance algorithm.
   Though, to attain the desired distribution, we might have to wait some turns. This might not be such a great feature of this.

Part 2(d) : Motivation behind this algorithm

This PID-based feedback algorithm, has been used in several areas, especially, in cases of linear systems. This is a linear system, because the the current_load_distribution is theoretically a linear function of the target_weight_distribution

My personal stint with such algorithms has been in multiple cases, where I had to 'hold on' to a particular value, despite errors and noises.That is the tasks demanded for the system to be balanced, quickly, if disturbed.

PID Tuning : How to find the value of Kp, Ki and Kd

Though we can manually find the values of these, but for a large system, it is very tiresome.

1. There are existing PID_tuners (can be write one from the scratch too, I have included this in the proposal too), that need us to define a cost function, that we need to minimise. In this case, it would be current_error,i.e. error in load distribution. They use PID tuning algorithms, based on Machine Learning concepts (more specifically, gradient descent). We need to provide data (controls: current_weight_distribution, iteration_number output: current_load_distribution) and the model would train upon it.

   A simple PID_Tuner was developed by my friend, for the Swarm Project, an autonomous, distributed robotics project. Though he has used Matlab for the same, converting to C++ shall not be much pain. Here[5] is a link for it.
2. There are also many PID auto tuning algorithms, like Ziegler-Nichols, which can auto-tune the values of Kp, Ki and Kd, as the system works.
   There is however a much simpler implementation. [14]

Tuning control algorithms is, however, tricky in that there is no one single best solution. You can tune a controller for faster rise time, or less overshoot or various other objectives. What we could do initialise the values from the output of a trained/semi-trained system, and use the auto-tune at each step, for best results. In either case, we would be needing data, and the more the data, the better. For this, I propose to write a test cases generator, that simply iterates and writes over the data onto a file, using the test cluster. If data is already available and accessible, it would be more than helpful.

Proposed Goals

1. Understand the CRUSH algorithm in depth and learn to use the tools to test the CEPH algorithm.
2. Identify the various processes that run when a reweight algorithm is called. Identify the independent metrics that can justify the performance differences between different reweight algorithms. Some of the metrics could be, load distribution between various OSDs with respect to the assigned weights, overall movement of data to and from various devices, total latency, CPU cycles, network bandwidth, IOPS, number of 'slow requests' etc.
3. Build a tool, to compute an overall score to different algorithms, by taking into weighted consideration of the various performance metrics as described above. Also, assign individual scores to the algorithm under each performance metric. Different performance metrics need to considered for Cluster level analysis and node level analysis. Generate a performance report using the same.
4. Discuss improvements and finer details in the proposed reweight algorithm and implement the same.
5. Test the algorithm against the evaluation tool and account for anomalies.

This is a challenging and interesting project on its own. There are certain additional goals (see Additional Goals) and more would be discovered during the course of the project, which I would be happy to take up, if time permits and mentors give a go ahead. But the task on hand shall be the top most priority, unless the additional goal has a direct implication on the project. I can also take up those additional goals post the completion of the Google Summer of Code, 2017.

How will I go about with this project?

Understand related areas

a) Establish a cluster for testing.
b) Understand the CRUSH algorithm, from src/crush.
c) Understand the various benchmarks in src/tools/bench and src/tools/fio like fio benchmark, network benchmark and rbd benchmark and the 'crushtool' to get an idea which metrics classify as factors of performance.[8] [9]
d) List the metrics that could be used to evaluate an algorithm. Amongst them, identify the "independent" metrics.

Develop evaluation tool

a) Assign suitable weights to the various independent performance metrics and assign a score under each performance header, and an overall score based on the weighted importance of the independent factors.
b) Create a module for ceph manager daemon in Python that extracts the "independent" metrics, and calculates the scores under various performance headers and generates a report while running the specified reweight_by_utilisation algorithm.
c) Design easily comparable dummy reweight algorithms and extensively test the evaluation tool. Explain the variation in the performance based on the data of the chosen metrics. Hints can be taken from an existing test file for reweight_by_utilisation.[7]

Implementing the proposed reweight algorithm

a) Describe the algorithm in more mathematical terms, and explain the reasons as to why the algorithm should generate the accurate results. Discuss with the community through mailing list, for possible improvements, if any.
b) Implement the algorithm in Python and design appropriate test cases. Initially, let the values of the parameters be set to some suitable (not yet initialised) values.

c) Run the test cases over multiple times and record all the data at each iteration (i.e. for different value of d).
d) Define the cost function and the independent variables for the model described in Proposed Solutions. Write a Python script for a PID tuning, based on gradient descent algorithm. Repeat this procedure, till the error levels at each iteration step is within acceptable limits.
e) Plugin the values of K_p , K_i and K_d thus obtained in the reweight_by_utilisation algorithm and rerun the test cases. Investigate for any fallacies and/or explain anomalies.

Bugs, Review and Documentation

a) Before attempting the task, I would notify my mentor about the task in details, and ask for suggestions/corrections, so that I avoid repetition of work and keep the project on schedule.
b) I would send a PR, at the completion of a complete build and correct output of the test cases. Each PR shall also include the appropriate additions in the documentation and test cases. There would be however a final PR of only changes in documentation, at the completion of each group of tasks, as classified above.
c) I would document the source code and the tests, at completion of each task mentioned above. And every Sunday, I would devote a few additional hours just to maintain the readability and documentation of the tasks I do, alongwith other files that affect this project. This is to avoid difficulties for others who would wish to contribute to this project in future.

Timeline (Tentative)

I understand that the project is not a trivial one. Hence I will be utilising every time that I can spare, in prior, to keep the project under schedule. I would like to utilise a part of the Community bonding period to write some of the codes, though most part of the community bonding would be used to interact with the community, explaining my work in depth and asking for suggestions.

1. 4th April - 15th April
    - Spend more time going through the codebase and get familiar with tools I don't know much about.
    - Setup the test cluster and get familiar with tools to test CRUSH algorithms.
    - Get a heads up, with the recent developments at thread on the ceph/devel mailing list, titled "crush multipick anomaly".[10]
    - Complete the unfinished bug fixes and send the PRs accordingly.
2. 22nd April - 29th April (Tentative, yet to be declared)
    - My spring semester exams.
3. 5th May - 20th May (1st and 2nd week)
    - I have gone through the Ceph documentation and architecture, and I am familiar with the workflow. So, during this period, I would discuss, my project with my mentor, and the community in general, with sufficiently details, take inputs and suggestions and make necessary changes to the project schedule, if necessary.
    - Evaluate the candidate metrics for determining the efficiency of reweight algorithms. Tabulate the data and send a comprehensive summary of the results and conclusions to the community, and my mentor in particular.
4. 21st May - 28th May
    - Design and code the tool to evaluate the performance of reweight process/algorithm.
    - Design suitable test cases.
    - Bug fixes.
    - Make iterative improvements suggested by the community and mentors.
5. 29th May - 30th May
    - Design a detailed, formal, mathematical formulation of the feedback-based algorithm presented, incorporating the changes suggested by the community and mentors.
    - Verify the algorithm's convergence, and provide a proof sketch about it's validity and usefulness.
    - Explore other approaches to this problem, if any and add to the list of additional goals.
6. 31st May - 15th June
    - Convert the algorithm mentioned above into Python scripts utilising the data available from the evaluation tools developed in the first phase.
    - Bug fixes

7. 16th June - 3rd July
    - Design test cases and evaluate the performance of the algorithm. Record all the results of the test cases. Iterate over different values of Kp, Ki and Kd, and record the performance of this algorithm under different metrics. This will be the data set for the next phase of work.
    - Explain anomalies and check for efficiency of the code.
    - Account for drop in performance, under any performance header.
    - Phase 1 evaluations from 26th June to 2nd July.
8. 4th July - 10th July
    - Buffer week.
    - Integration testing.
    - Bug fixes
9. 11th July - 22nd July
    - Set up a machine learning system, based on gradient descent, to provide the initial values of K_p , K_i and K_d.
    - Bug fixes.
10. 23rd July - 25th July
    - Plug-in the values of K_p , K_i and K_d, and re-run the test cases. Observe and infer the results.
    - Notify the community and the mentors about the results.
11. 25th July - 29th July
    - Phase 2 Evaluations. Buffer time.
12. 30th July - 25th August
    - Buffer time. Can be used to implement the other additional goal.
    - If the project is behind scheduled, due to any reason, then this will serve as a backup.
13. 25th August - 28th August
    - Finishing off any remaining work.
    - Final touches to the project. Getting everything reviewed by the mentor.
    - Implement any last minute suggestions.
    - Final testing and final PR.

Additional Goals

- Allow the user to choose different weights for different metrics, thus allowing them to create a customised testing scenario, for the evaluation of reweight algorithms.
- Even after plugging in the values of K_p , K_i and K_d , they should be liable to change over the iterations that occur at each time during actual deployment.

- Hence, at each step the performance should be logged and the additional data generated should be analysed to introduce changes in the values of the K_p , K_i and K_d.
- Write a bash script to automate this process at every ‘draw’.

Plans for Summer

Currently, I don’t have any plans for summer. I will start my work early, knowing well, that it is not a very trivial project. I can devote about 50-52 effective working hours during my summer holidays and I will scale this up, if required, with my best attempts to finish the project prior to 3rd August.

How can mentors keep a track over my work?

The work I will be doing will be pushed to my fork of ceph/ceph repository.[13]. I will be sending PR as I complete the logically significant tasks, and I would tag my mentors over the PRs. I generally reply to emails within 8 hours, unless any unavoidable situation happens. I would try to answer queries of other ceph users, as I gain more knowledge about the working of ceph. This way, I would get along with the community better.

Things to remember

1. The code shall be compatible with Python 2 and Python 3.
2. All features to have separate tests.
3. Run ‘make check’ tests on local machine.
4. Using Travis for continuous integration.

Contributions to Ceph till now

1. [Merged] bug #17195:
   Stop indefinite thread waiting in krbd.cc[11]
2. [Active]
   cleanup#19284: script to find broken URL [12]
3. Currently working on :
   bug #17453 : ceph-mgr doesn't forget about MDS daemons that have gone away
4. Bug suggested by mentor (shifted to another bug) :
   bug #15653 : crush: low weight devices get too many objects for num_rep > 1

References

[1] : http://www.ssrc.ucsc.edu/papers/weil-sc06.pdf

[2] : http://tracker.ceph.com/issues/15653

[3] : https://github.com/ceph/ceph/pull/10218/files

[4] : https://github.com/ceph/ceph/pull/10218

[5] : https://github.com/ManashRaja/learnPID

[6] : https://github.com/ceph/ceph/pull/7890/files

[7] : https://github.com/ceph/ceph/pull/8027/files

[8] : http://crush.readthedocs.io/en/latest/

[9] : http://libcrush.org/dachary/libcrush/blob/wip-sheepdog/compare.c

[10] : https://github.com/plafl/notebooks/blob/master/converted/replication.pdf

[11] : https://github.com/ceph/ceph/pull/14051

[12] : https://github.com/ceph/ceph/pull/14052

[13] : https://github.com/SpandanKumarSahu/ceph

[14] : https://github.com/br3ttb/Arduino-PID-AutoTune-Library/tree/master/PID_AutoTune_v0

[15] : http://docs.ceph.com/docs/master/rados/operations/monitoring-osd-pg/