Dynamic Source Selection Strategies for CERN's Rucio

David Cheng
CPSC 534
david.cheng@yale.edu

Jeffrey Chen
CPSC 534
jeffrey.chen@yale.edu

Christian Martinez
CPSC 434
chris.martinez@yale.edu

1 Introduction

Rucio [1] is a scientific data management tool used in conjunction with other software at CERN to ensure that the organization's storage centers around the world store the right data in the right amount. One key workload of Rucio is copying files from one Rucio Storage Element (RSE) to another. If we want one RSE to have a copy of a file, there may be multiple possible RSEs with the specified file that can serve as the source of the file copy. Thus, a source selection policy is required to determine the "best" RSE to copy from.

The current source selection policy in Rucio is static—source RSEs are first filtered by boolean operators checking certain characteristics of the RSE (e.g. whether the RSE is restricted or blocklisted, tape or disk type), and then the RSEs are ranked based on innate properties. Currently, the source nodes are ranked based on their path distance from the destination node. The existing path distance calculation uses Dijkstra's Algorithm to calculate the distance between each source node with the destination node, and the node with the shortest distance is chosen.

Ticket #5012 aims to make the source selection policy dynamic. This includes accounting for existing loads and fluctuating properties of each source RSE. In this report, we discuss our implementation of two dynamic source selection strategies: `FailureRate` and `TransferWaitTime`. Both strategies take into account real-time properties of the source nodes: the failure rate and wait time (in the source node's current task queue) respectively. Additionally, we implement metrics for wait time and transfer time to analyze the performance of our new source selection strategies versus existing ones. Using the new metrics, we run simulations to experimentally validate the performance boost of our `TranferWaitTime` strategy. Additionally, the `FailureRate` strategy has been successfully reviewed by the Rucio team and merged into the Rucio main branch.

2 Source Selection Infrastructure

When a request is received, the `Preparer` daemon must build the transfer path needed to satisfy the request. If we want to have a file replicated on a destination RSE, there may be multiple source RSEs that can send copies to the destination. The possible sources and associated transfer paths are ranked and returned by `build_transfer_paths()` in `transfer.py`.

In order to conduct this ranking, a cost vector is computed for each possible source. This cost vector is n-dimensional where the operator specifies n `SourceRankingStrategy`. Each `SourceRankingStrategy` assigns an integer cost to each source based on different criteria where smaller costs are preferred. Costs in the front of the vector are considered first and later costs are only considered in the case of ties.

The benefit of this infrastructure is that it supports many different source ranking strategies that can be easily interchanged and interlaced. The ***control plane*** of this infrastructure is that the operator is able to change the configuration file by listing the source ranking strategies they would like and in what order. This can be done as the Rucio system is running since every time the transfer path rankings are built, the configuration file is re-read. If there are no strategies specified by the operator, there is a default set of source strategies that is used. We can think of this infrastructure as similar to a *pipeline* structure (which is used in nginx and Kubernetes) in which we can think of source ranking strategies as modules we can easily plug into the source selection process.

3 Our Contributions

Since we would like to make source selection dynamic, this means we would like to process data about currently queued and recently executed transactions and use that in our decision-making when selecting sources to initiate future transfers. To do this, we implemented two dynamic source ranking strategies: `TransferWaitTime` (PR #6415) and `FailureRate` (PR #6403). Additionally, we developed performance metrics that measure the effectiveness of a given source selection strategy

by keeping track of wait time and transfer times for requests (PR #6420).

3.1 Strategies: Design and Theoretical Analysis

The `TransferWaitTime` strategy takes into account the existing load of the source nodes. Each source has a queue of transfer requests it needs to process. Each transfer request specifies a file with a certain size in bytes that will be copied onto a destination node. When choosing a source node for a new request, we approximate the time it will take for the source nodes to finish processing their queues before processing the new request. Let $N(s)$ be the number of files in a source node's queue. Then, we can estimate the time it takes to process the queue of source s as

$$T(s) = N(s)T_f + \sum_{i=1}^{N(s)} \texttt{bytes}(\text{file i}) * T_b \quad (1)$$

where $N(s)$ is the number of files in the queue of source s, T_f is the overhead cost for transferring even a very small file ($\approx$ 10 s as was provided by discussions with the Rucio members), and T_b is the time that it takes for one byte to be sent assuming a constant throughput of 10 Mbps.

We prefer that the source nodes with shorter queue wait times will be selected but to ensure that the best source nodes according to `TransferWaitTime` are not oversaturated, we also introduce exploration in the form of randomization. First, we compute the average transfer wait time among all possible source nodes. Next, we ensure that source nodes with transfer wait times below the average (i.e. better than average source nodes) will be selected 90% of the time, while source nodes with transfer wait times above the average will be selected 10% of the time. Within the 90% and 10%, we distribute the probability that a source node is picked proportionally by wait time. Note that to compute weights, we take the reciprocal of wait times and normalize because lower transfer wait times are better, so they should have a higher weight and consequently higher probability of being chosen.

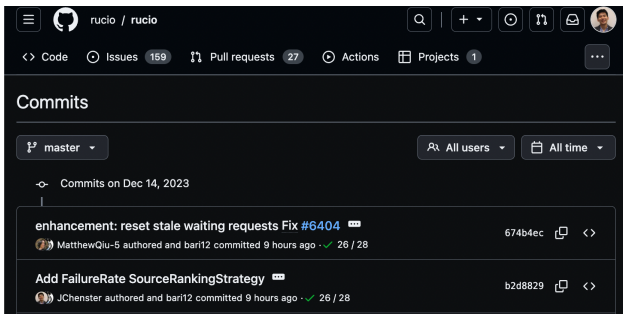


Figure 1: Screenshot of commit for `FailureRateStrategy` into the Rucio main branch

The `FailureRate` strategy was thoroughly tested, code reviewed, approved, and **merged into the main production branch** on December 14 (commit link)! It is designed to be used in conjunction with other source selection strategies. The use case is that given a set of some sources that we consider equally good based on static criteria (such as path distance, number of hops, disk vs tape), we would like to differentiate between them based on their ability to handle their current load.

Based on an analysis of a representative JSON log from the Rucio team of past requests and their failures, we noticed that failures occurred a non-trivial amount for each source node in a past hour time window. This demonstrates that `FailureRate` is a useful strategy because the failure rate indicates how well a source node is handling its current load. If it is overloaded, then it will fail more often. Additionally, using failure rate is more useful than simply looking at the load of an RSE since RSEs in Rucio are heterogeneous and have different storage and throughput capabilities. As a result, different RSEs may handle the same load differently e.g. an RSE may have a low load, but we may still not want to choose it because it may have a high failure rate.

3.2 Performance Metric

Our final contribution is our design of a performance metric that measures the effectiveness of a strategy. The metric should answer the question: is one strategy *better* than another strategy? To answer this question, we store two measurements for any given request: the wait time and the transfer time. The wait time is the time a request spends between its submitted stage and processing stage. Essentially, it measures how long a request spent waiting in a source node's task queue. An efficient source selection algorithm will balance the load effectively between source nodes such that the average wait time of the topology decreases. The transfer time of a request includes the wait time and the actual time it took to copy the file from the source node to the destination node. Like the wait time, the overall transfer time of the topology will decrease with a better source selection algorithm.

3.3 Implementation

The `TransferWaitTimeStrategy` is implemented as a `SourceRankingStrategy` class. We first calculate the estimated waiting time of each source node's queue using Equation 1. Let S be the set of source nodes we are choosing between with cardinality n. We then calculate the average wait time as $T_{\text{avg}} = \frac{\sum_{s \in S} T(s)}{n}$. Next, we want to assign weights that are proportional to wait time among below-average and above-average nodes. In order to guarantee proportionality and that lower wait times have higher weights, we use the reciprocal of wait time. We normalize this and then multiply by the exploration factor α to obtain weights for all our sources

that forms a probability density function that sums up to 1. We use $\alpha = 0.1$ so that we explore above-average wait nodes 10% of the time and below-average wait nodes 90% of the time.

Mathematically, we have:

$$w(s) = \begin{cases} \alpha \cdot \frac{\frac{1}{T(s)}}{\sum_{u \in S} \frac{1}{T(u)} 1_{T(u) \geq T_{\text{avg}}}} & T(s) \geq T_{\text{avg}} \\ (1-\alpha) \cdot \frac{\frac{1}{T(s)}}{\sum_{u \in S} \frac{1}{T(u)} 1_{T(u) < T_{\text{avg}}}} & T(s) < T_{\text{avg}} \end{cases} \quad (2)$$

A naive implementation of sorted weighted random sampling (where we pick one element at random, pop it from a list of contenders, and repeat) is $O(n^2)$. However, because the `SourceRankingStrategy` class only requires that we return costs for each source, we can speed up the runtime to $O(n)$ using an efficient solution to weighted random sampling [2]. The idea is simple. We transform weights into keys using the following formula:

$$k(s) = u(s)^{1/w(s)} \quad (3)$$

where $u(s)$ is a uniform randomly generated float from $(0,1)$. It then follows that $k(s) \in (0,1)$ with high floating point precision so we multiply by $-2^{30} \approx -10^9$ to obtain a granular enough integer cost as desired (the negative sign is so that higher weight nodes have smaller cost). We robustly account for division by 0 errors by ensuring that $T(s)$ is non-zero by adding a small epsilon when it is 0.

The `FailureRate` strategy is also implemented as a `SourceRankingStrategy` class. To calculate the failure rate for a particular node, we fetch the node's `files_done` and `files_failed` from the `TransferStatsManager` in the past hour. The cost of each source node is therefore the rate of failed file transfers in the past hour times 10,000 (to allow 4 decimals of precision in the failure rate in integer representation). The node with the lowest failure rate will be chosen by this strategy, and lower-ranked nodes may be chosen as backups.

To implement our new metrics, in `request.py`, whenever the `TransferStatsManager` observes a change in a transfer's state to `DONE`, we record the wait time and transfer time of the transfer into the `MetricManager`. The `MetricManager` stores statistics that will be pulled by the Prometheus client. The Prometheus client will display a histogram of the wait time and transfer time of all requests in time buckets. With these raw values stored, we can utilize Prometheus's query engine to write queries such as "How many requests ended in the last 24 hours took more than an hour to complete?" or "What is the duration of transfers if we ignore the highest 5 percentile?" These queries will allow us to see the general trend of the wait time and transfer time as source selection strategies are swapped.

3.4 Unit Tests

We implemented unit testing for both of our source selection strategies. For each of our new strategies, we mocked the configuration file to include and exclude our new strategy to check that the effect of having or not having our strategy was as intended.

For `TransferWaitTime`, we add mock queued requests to the database such that our 4 sources have estimated queue wait times of 20, 40, 60, and 120 seconds. Using Equation 2, we should then obtain that $w(1) = \frac{3}{10}, w(2) = \frac{3}{5}, w(3) = \frac{1}{15}, w(4) = \frac{1}{30}$. We simulate 100 requests and assert that the number of times a source is picked is reasonable with respect to these weights. We had to be crafty about the assertions since it is a randomized algorithm. If we consider the probability of selecting a source as a success in a binomial trial, we can construct a 99.9936% confidence interval for how many times a source is picked using 4 standard deviations: $[100w(s) - 4\sigma(s), 100 + 4\sigma(s)]$ where we calculate the standard deviation as $\sigma(s) = \sqrt{100(w(s)(1-w(s))} \leq 2\sqrt{6}$ [3].

For `FailureRate`, we mocked data about past transfers in the `TransferStats` database table with different failure rates for different sources. We then ensured that all other strategies being equal, the sources with lower failures were preferred.

4 Evaluation

To evaluate the performance of our `TransferWaitTime` strategy, we developed an environment that simulates the life cycle of transfers. Requests are generated and submitted for the actual transfer of files. We include our source code for the simulation in the `wait_sim` branch of our forked Rucio repository.

4.1 Simulation Environment

Within our environment, we first construct a topology with 7 RSEs. Each RSE is assigned a random weight that corresponds to its likelihood of being chosen as a destination source node that requires a file. Each RSE has a 75% chance of being linked to each other RSE, with random distances in the range $[1, 100]$.

With the topology created, we proceed to generate requests. A constant number of requests are generated, and in each request generation a RSE is probabilistically chosen as a destination node based on its weight. A rule is instantiated, and files are added to all other nodes, signifying that those nodes are potential source nodes for this particular request.

After the generation of requests, the simulation begins. We start a submitter daemon that is analogous to the submitter in Rucio. The submitter fetches any requests that are in a queued state and runs the source selection strategy for those requests. After selecting a source node, the submitter will update the

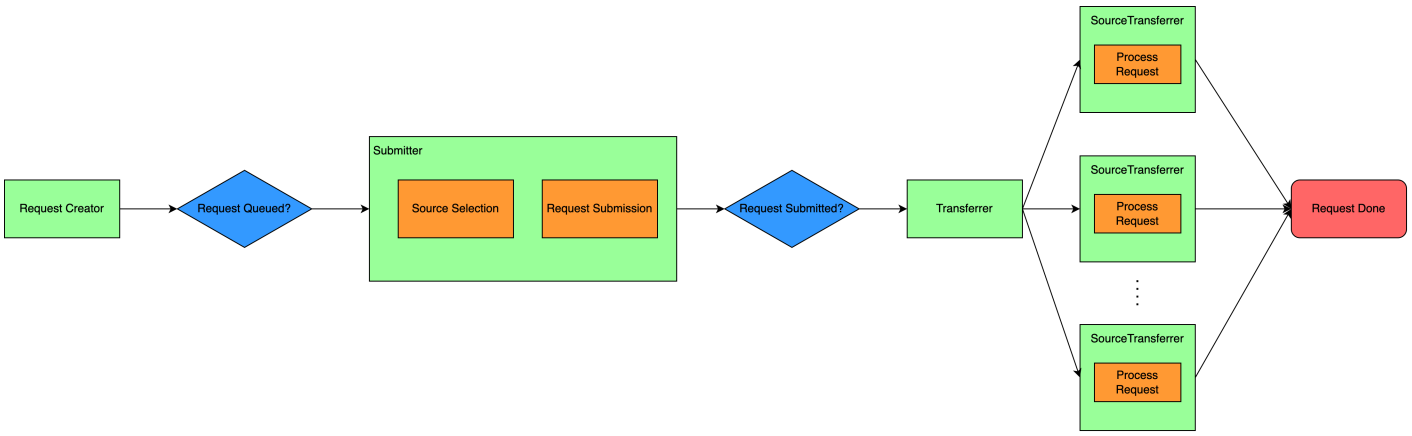


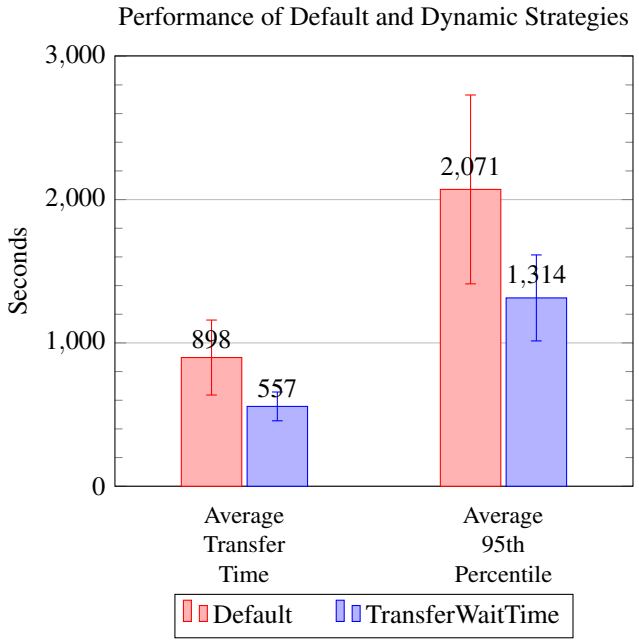
Figure 2: Life cycle of a request in the simulation environment

request's state to a submitted state and the request's source RSE to the selected source node.

Simultaneously, we initialize a transferrer daemon that emulates the transfers of files between nodes. The transferrer will fetch any submitted requests in the request table, and send these requests to the corresponding queues of `SourceTransferrer` objects, which are mocks of an actual RSE. Each node will have an instantiated `SourceTransferrer` object and a queue of requests to process. A `SourceTransferrer` will pop requests from its queue and simulate work by sleeping for a set amount of time depending on a file overhead and the number of bytes the file contains. Once the file has been transferred, the `SourceTransferrer` will mark the request as done in the Request table. The submitter and transferrer daemons will continually submit and process requests until an end time is reached.

At the end of the simulation, we measure two metrics that are performance indicators for source selection: wait time and transfer time. We go through the Request table for all completed requests and calculate the average wait time and average transfer time of the simulation. The same simulation is run with different source selection strategies: one with the default (static) strategies Rucio currently uses, and another run with our `TransferWaitTime` strategy.

4.2 Simulation Results



Above we see the results of the simulations we performed. The graphed values represent the averages across ten simulations. In red we show the average values for the simulations using default (static) strategies, and in blue we show the average values for the simulations using our dynamic `TransferWaitTime` strategy. The left group represents the average of the average transfer times for our ten simulations, and the group on the right represents the average of the 95th percentile transfer times of our ten simulations. The error bars represent the standard deviation of these values for the ten simulations.

It is evident that our dynamic `TransferWaitTime` strategy performed significantly better than the default strategies that Rucio currently uses. On average, transfers for a file took

about 40% less time to complete when using our dynamic strategy for a 1.6x speedup in request processing.

The extremes for request processing times were also significantly improved. Using the default source selection strategies, the tail end of transfers was composed of requests that could take upwards of 3400 seconds to be fulfilled, with the average 95th percentile across our ten simulations being 2071 seconds. Using our dynamic strategy, however, all requests were completed within 1700 seconds, with an average 95th percentile across our simulations of 1314 seconds.

This is where a dynamic strategy like `TransferWaitTime` is able to shine. It is able to take into account the state of the entire system and make an informed selection. On the other hand, the current static strategies are unaware of how the system is changing and can easily overwhelm RSEs with requests. This causes requests to sit in queues with long times and for the end user to have to wait twice as long. We see this with the standard deviation of 95th percentiles. Using the `TransferWaitTime` strategy, we have a much tighter bound on how long a request may take to be fulfilled, so a user submitting a request has a much better understanding of when they may expect to see their request completed.

5 Conclusion

We implemented two new end-to-end strategies to dynamically select source RSEs for file transfers in Rucio. `TransferWaitTime` aims to minimize the amount of time transfers wait in queue by estimating the amount of time it takes to process the current queue of contender source nodes and using these estimates to choose source nodes with weighted randomness. We derive a mathematically sound and efficient algorithm to do this with a small amount of exploration so that even the least desirable RSEs are chosen sometimes in order to ensure proper load balancing. We then experimentally verify our theoretical hypothesis using a comprehensive simulation environment which reveals that using this dynamic strategy cuts request processing times by 40% on average and reduces the worst-case request processing times to nearly half as well. In addition, we implement a `FailureRate` strategy which is aimed at addressing the relatively frequent failed file transfers that are observed each hour. The `FailureRate` strategy supplements existing source strategies such that all else being equal, we select source nodes with a lower failure rate in the past hour. The Rucio team agreed with its practical and non-invasive use case and the change was successfully merged into the production branch. Newly incorporated metrics for topology-wide queue wait time and transfer time rounds out our contributions to source selection, enabling more in-depth monitoring and analysis of different source selection strategies. This allows us to analyze the effectiveness of not only our new strategies but also potential future ones as well.

References

[1] BARISITS, M., BEERMANN, T., BERGHAUS, F., BOCKELMAN, B., BOGADO, J., CAMERON, D., CHRISTIDIS, D., CIANGOTTINI, D., DIMITROV, G., ELSING, M., GARONNE, V., DI GIROLAMO, A., GOOSSENS, L., GUAN, W., GUENTHER, J., JAVUREK, T., KUHN, D., LASSNIG, M., LOPEZ, F., MAGINI, N., MOLFETAS, A., NAIRZ, A., OULD-SAADA, F., PRENNER, S., SERFON, C., STEWART, G., VAANDERING, E., VASILEVA, P., VIGNE, R., AND WEGNER, T. Rucio: Scientific data management. *Computing and Software for Big Science 3*, 1 (Aug. 2019).

[2] EFRAIMIDIS, P., AND SPIRAKIS, P. *Weighted Random Sampling*. Springer US, Boston, MA, 2008, pp. 1024–1027.

[3] MAYFIELD, P. Understanding binomial confidence intervals, 2018.