Dynamic Login Node Resource Control and Monitoring with Arbiter 3

Kai Forrest*
kai.forrest@utah.edu
Center for High Performance Computing
University of Utah
Salt Lake City, Utah, USA

Jackson McKay*
jay.mckay@utah.edu
Center for High Performance Computing
University of Utah
Salt Lake City, Utah, USA

Paul Fischer
p.fischer@utah.edu
Center for High Performance Computing
University of Utah
Salt Lake City, Utah, USA

Abstract

Login nodes, also known as interactive nodes, are an essential component of high-performance computing (HPC) environments. Covering a range of different use cases, they provide users of a cluster with flexibility when developing and running programs. However, the resource management of these shared machines is an ongoing problem within the community, as each institution has different requirements of these nodes. In most cases, these nodes are shared simultaneously by multiple users, and providing users with freedom and fairness has proved to be a difficult task. To address these challenges, we present Arbiter 3, a software suite for setting and enforcing resource usage policies on Linux nodes via cgroups. In this paper we discuss the previous version of Arbiter, and the updates and improvements we have made in the new Arbiter 3.

Keywords

cgroups, systemd, observability, monitoring, OpenMetrics, Prometheus

ACM Reference Format:
Kai Forrest, Jackson McKay, and Paul Fischer. 2024. Dynamic Login Node Resource Control and Monitoring with Arbiter 3. In *Proceedings of Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

HPC clusters are composed of a variety of resource types, each of which must be managed for efficient use. Most of these resources are managed using well-established community solutions, such as workload schedulers for compute resources, and multi-tenant storage clusters. In some environments, an additional resource type is utilized: the shared interactive or "login" node. These nodes are used as network gateways to the HPC environment, and may also support interactive, non-batch workloads. Most institutions have policies that govern what is acceptable usage on these nodes, usually limiting users to running lightweight tasks such as submitting jobs, writing/editing programs, compiling code, and small test runs of a job. However, if a user chooses to ignore these policies and abuse the system resources, other users of the shared resources may be negatively impacted.

Arbiter was created to solve this problem by monitoring resource usage and automatically limiting users and processes that violate defined usage policies. To achieve this, Arbiter leverages the Linux cgroups that are automatically created by systemd. At a regular interval, Arbiter queries resource usage metrics (such as CPU and memory) from a central time-series database, evaluates whether any policies were violated, and remotely configures cgroup limits on each node to reflect those policies. By setting limits dynamically rather than statically (for example, 1 core per user), users can safely "burst" their usage for short periods, and most users will not notice the limits set on their usage. However, any sustained overuse will be promptly corrected by Arbiter.

Arbiter 2 [3] was the first implementation of this behavior, following a simple bash script that monitored behavior and sent emails to users. Since the release of v2, we have seen adoption at many institutions, and development has been maintained since. However, due to limitations with its initial design, we created a version with updated architecture and design to address the issues that v2 could not. We call this Arbiter 3.

2 Arbiter 2

Development on Arbiter 2 first began in 2017, and its initial release was in 2019. As the successor to the original Arbiter, it was written to provide the functionality to set resource limits as well as monitor their usage. Here we provide a brief overview of the design of v2, but the full details are available at https://github.com/chpc-uofu/arbiter2.

2.1 Design

Arbiter 2 was written as a monolithic Python application that ran locally on each node in a self-contained fashion. While this helped simplify the initial development process, this means that each individual instance of Arbiter running must perform the monitoring, evaluation, and resource limiting.

2.2 Linux cgroups

Internally, Arbiter 2 relies on the cgroups [2] that systemd [4] sets up for each user upon login. If the controllers are enabled, they will be created for each user at:

```
/sys/fs/cgroup/cpu/user.slice/user-$UID.slice
/sys/fs/cgroup/memory/user.slice/user-$UID.slice
```

*Both authors contributed equally to this research.

SC-W '24, November 17-22, 2024, Atlanta, GA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Memory usage is monitored with the `memory.usage_in_bytes` file of the memory controller, and CPU usage is monitored with the `cpu.usage_percpu` file of the CPU controller. Similarly, resource limits where set by modifying `cpu.cfs_quota_us` and `memory.limit_in_bytes` for CPU and memory respectively. Notably, these locations were hard-coded, and thus do not work with the unified hierarchy of cgroups v2.

2.3 Synchronization

After the initial release of v2, it was decided that some state needed to be shared between the various instances of arbiter. For example, if a user violates a policy on one interactive node in a cluster, they should be put in penalty on all interactive nodes in that cluster, so they could not avoid penalty by simply changing hosts. This was achieved by having each instance write their state to a MySQL database, which included users currently in penalty. Each instance would then read all of the statuses of each user in their cluster, and take the most recent state.

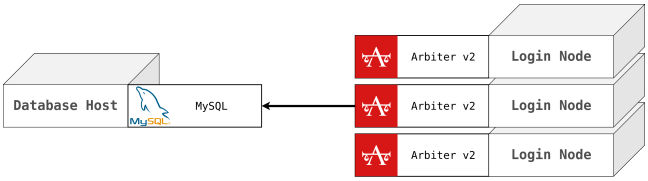

Figure 1: Arbiter 2 Architecture

3 Motivation

While Arbiter 2 provides a solution to the resource sharing problem, there are a few issues with it, of which we have been made aware of by the community and our own experience here at CHPC. The following are the primary issues we attempt to solve with Arbiter 3.

3.1 Performance

Because of the way Arbiter 2 is written, each individual instance must run the entire program loop. While this process is not too intensive, it does at times cause undue load on the node it is running on, sometimes resulting in a small but noticeable slow down for users. This can usually be attributed to the walking `/proc/pid/smaps` for per-process memory accounting, but some of it may be due to the inherent performance overhead of the Python interpreter.

3.2 Shared state

Functionality to sync the state between instances of Arbiter 2 was eventually added, for reasons mentioned previously. The original design did not take this functionality into account, and the implementation reflects this. Arbitrary changes to the code were added to make this behavior possible, making maintaining the project more difficult. When changes are made to the database schema, custom migrations have to be written to migrate existing data between the old and new tables.

In addition, the algorithm for deciding which state to use is very naive - each instance just takes the last "valid" state written, i.e. the most recent change for that synchronization group. This works well enough given the simplicity of the task, but is prone to issues if a machine goes down or loses connection.

3.3 Installation

As each instance of Arbiter 2 is largely self-contained, save for the optional synchronization feature, each requires a separate install of Python3 and all Arbiter dependencies. Managing these deployments can be time-consuming. In addition, the installation for Arbiter v2 itself has quite a few steps. Service files must be set up, permissions must be granted, and accounting must be enabled. Each of these steps is prone to mistake, meaning some installations must be debugged. Repeating this process across all hosts can end up being quite frustrating for administrators tasked with standing up Arbiter on their systems.

3.4 Supporting cgroups v2

With the kernel release of 4.5, released in 2014, a new version of cgroups was added. This version, v2, changed the hierarchy of the cgroup controllers. This new hierarchy (referred to as "unified") now has the controllers for memory and CPU for a given user slice at

```
/sys/fs/cgroup/user.slice/user-%UID.slice/memory
/sys/fs/cgroup/user.slice/user-%UID.slice/cpu
```

Because of the way Arbiter 2 is written, it does not support this new hierarchy. And while cgroups v2 has been in the kernel for some time, RHEL 9 now enables it by default. While a workaround does exists (modifying a kernel boot parameter) it is a short term solution to a problem that must be directly addressed.

3.5 Configuration

Version 2 offered configuration in the form of toml files. While these are easy to read and understand, they are again required for each individual instance of Arbiter. Most of the configuration may just be copied, but some lines specific to each host and must be modified individually.

3.6 Violation detection

Because Arbiter 2 only had access to instantaneous usage data, a *badness score* was used as a proxy for usage over time. Each user has an initial score of B, and the change in score for each resource r was calculated as follows:

$$\Delta B_r = \begin{cases} \eta \frac{u_r}{Q_r} & u_r \geq C_r \\ -\mu \left(1 - \frac{u_r}{Q_r}\right) & u_r < C_r \end{cases}$$

Where η and μ are the maximum and minimum change in badness score for a given unit of time, respectively. C_r is the usage threshold for a particular resource, and Q_r is the current quota for that resource. The total change in score ΔB is then the sum across all resources $R = \{r_0, r_1, ...r_n\}$, thus

$$\Delta B = \sum_{i=1}^{|R|} \Delta B_r$$

The violation function V is then

$$V = \begin{cases} 1 & \Delta B + B \geq 100 \\ 0 & \Delta B + B < 100 \end{cases}$$

This way of calculating violations is overly complicated, and is due to the limited data v2 has access to. Some of the configuration variables are confusing, such as η and μ, which must be derived from a desired time to penalty. Possibly the greatest issue with this method is communicating what just type of behavior will lead to a violation, this is not inherently obvious from the description of the function. Many sites using Arbiter v2 have had to write their own documentation.

4 Architecture

With Arbiter 3, we made some major architectural changes. We decided to split out the program into a few core components: the time-series database (TSDB), the Arbiter service, and the login Daemon (`cgroup-warden`).

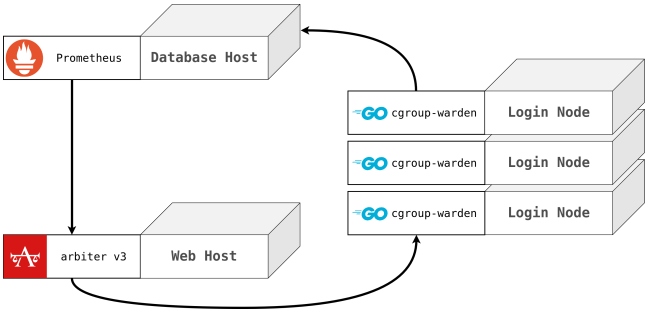

Figure 2: Arbiter 3 Architecture

5 Implementation

Development of version 3 started in 2022, after a discussion regarding cgroups v2 support. Each component was designed and developed with a different focus in mind, and the technologies used reflect that. Below we describe the core components.

5.1 Time-series database

The TSDB was included to provide v3 with access to historical data. This data is used to greatly simplify the calculation of violations, and provide a way to monitor user usage on interactive nodes in a historical fashion. The time series database scrapes the metric endpoints made available by each `cgroup-warden`, and then exposes an API endpoint that the Arbiter agent can interact with. In our set up, we used Prometheus to scrape the metrics and store the data, but any other compatible TSDB may be used.

5.2 Login Daemon

The cgroup-warden daemon is what provides Arbiter with information about process usage, and allows resource limits to be set. Each daemon runs on a login node, and expose endpoints for monitoring usage and enforcing limits. The daemon is written in Go for increased performance and ease of deployment. cgroup-warden provides two https endpoints, /metrics for the time-series data, and /control for the setting of resource limits.

5.2.1 Control. The control endpoint is where the Arbiter service interacts with the daemon. This endpoint accepts POST requests with JSON payloads, with the data describing a systemd property to modify with a value to modify for a particular user or cgroup. It then uses the go-systemd bindings to modify the property via sytemd. The arbiter service modifies CPUSecPerUSec and MemoryMax, but can modify any property. Additionally, because the daemon uses systemd bindings, it is cgroup version agnostic.

5.2.2 Metrics. The metrics endpoint is where the monitoring data is made available. This endpoint is functions similarly to the cgroup_-- exporter [1], which exports data about user slices. While this exporter cannot be used directly, inspiration was taken from it and applied here.

5.3 Arbiter core service

The core of Arbiter 3 is the Arbiter evaluation service. This program is what evaluates a user's usage according to its policies and sets the appropriate limits. It provides a web dashboard to monitor users and their violations, and to edit configurations.

5.3.1 Improvements to violation detection. When moving to a time-series database, it became clear that a much simpler form of the *badness score* in v2 could be replicated using existing aggregations in the time-series database queries, namely the average usage over time function. While these both effectively accomplish the same goal, it is worth noting here that there are slight changes in behaviors (e.g. no history reset on violation). The new method for calculating if a user is in violation is as follows:

$$U_{avg}(r) = \sum_{i=0}^{n} \frac{Usage(r, t_i)}{n}$$

$$V(r) = \begin{cases} 1 & U_{avg}(r) \geq \tau_r \\ 0 & U_{avg}(r) < \tau_r \end{cases}$$

Where $t_0, t_1, ...t_n$ are the timestamps at which metrics where scraped during the interval, n is the number of scrapes in the time interval, τ_r is the usage threshold for that resource, and $Usage$ is the user's usage for that resource at that time.

5.3.2 Web Interface. With Arbiter 3, one our goals was to simplify configuration and provide better interfaces for Arbiter admins. We selected the Python Django web framework to rapidly iterate and build the application. Arbiter leverages the built-in Django Admin interface for configuration on Arbiter's policies. In addition, we have made an admin dashboard to view recent violations in tabular and chart formats, view reports on recent usage with per process breakdown on resources consumed, and apply arbitrary limits/cgroup properties or "release" someone from penalty status.

5.3.3 Data Models. We define the Policies and Penalties of the Arbiter service as follows:

- A **Property** is a systemd property that can be set for a cgroup.
- A **Limit** is a value associated with one of those systemd Properties that will be set when a user goes in Penalty

- A **Penalty** is a set of limits (CPU and Memory) and duration that gets applied to a user when a Policy with this penalty gets violated
- A **Policy** is a (usage) rule that a user can violate, in the form of a threshold over time, which will cause a Violation to be made with the Policy's respective penalty
- A **Violation** is an instance of a violation of a specific policy for a user-host pairing, which has an associated Penalty and Policy
- A **Target** is a user-host pairing that Arbiter monitors usage and penalizes

Below is an entity-relationship diagram (ERD) describing Arbiter's data model:

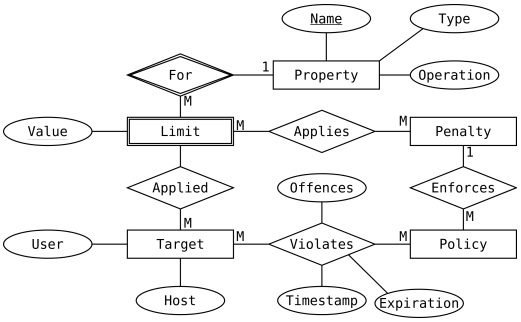

Figure 3: Arbiter 3 Data Models

5.3.4 Penalty Scaling. With this change in architecture for Arbiter 3, some of the features of Arbiter 2's policies were no longer realistic to implement on Arbiter 3's system. The most notable of these features was Arbiter 2's tiered penalty system where, if a user violated a policy right after exiting penalty status, their penalty would be "upgraded" to a harsher set of limits. With Arbiter 3, policies can potentially overlap for some nodes but not others, meaning tiered penalties in this system would add a needless complexity to evaluation and configuration. Instead, we opted to only have one penalty status associated with each policy and instead scale the **amount of time** the user was in penalty status relative to their number of recent violations. In addition, to replicate some behaviors of having a harsher penalty status, admins can instead make multiple policies with more demanding usage thresholds needed for violation.

6 In Practice

In action, v3 operates the same as v2 where a user gets limited uses more resources than allowed by the policy for long enough (memory or CPU-cores as shown in figure 4).

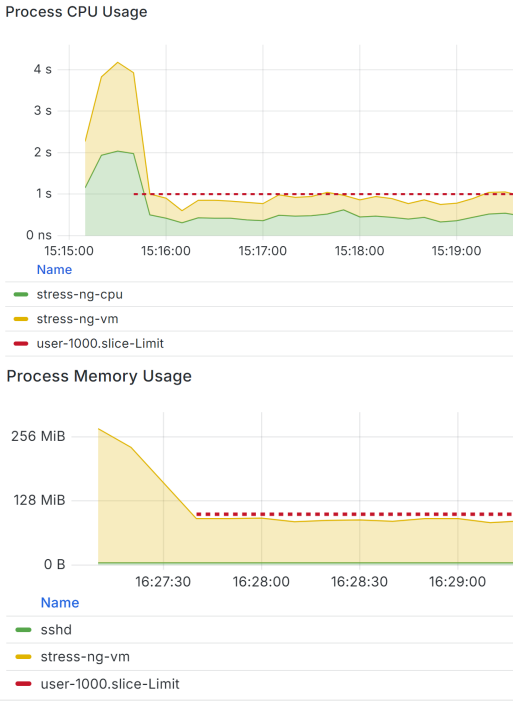

Figure 4: Arbiter 3 Usage Graphs

v3 also maintains the ability to send users emails detailing their usage and status upon a violation, only requiring a site-specific mapping function of usernames to emails.

7 Acknowledgments

We would like to thank the original authors of Arbiter 2, Dylan Gardner and Robben Migacz, for their work. We would also like to thank the entire staff at CHPC for their support.

References

[1] Trey Dockendorf. 2024. cgroup_exporter. https://github.com/treydock/cgroup_exporter.
[2] Linux Kernel Documentation. 2024. Control Group v2. https://docs.kernel.org/admin-guide/cgroup-v2.html. Accessed: 2024-08-08.
[3] Dylan Gardner, Robben Migacz, and Brian Haymore. 2019. Arbiter: Dynamically Limiting Resource Consumption on Login Nodes. In *Practice and Experience in Advanced Research Computing 2019: Rise of the Machines (Learning)* (Chicago, IL, USA) *(PEARC '19)*. Association for Computing Machinery, New York, NY, USA, Article 32, 7 pages. https://doi.org/10.1145/3332186.3333043
[4] Systmed Team. 2024. Systmed: A System for Modern Data Management. https://systemd.io/. Accessed: 2024-08-08.

A Source code and installation

The source code, installation guides, and contribution policies are available at https://github.com/chpc-uofu/arbiter and https://github.com/chpc-uofu/cgroup-warden.

Received 9 August 2024