Visualizing Workflows with the Dragon Telemetry Service

Indira Pimpalkhare
Maria Kalantzi
Colin Wahl
indira.pimpalkhare@hpe.com
maria.kalantzi@hpe.com
colin.wahl@hpe.com
Hewlett Packard Enterprise
Seattle, Washington, USA

Abstract

The Dragon Telemetry Service is an easy-to-use, scalable means for users to visualize both hardware and custom metrics for complex workflows implemented in Dragon. We discuss in-depth the Dragon runtime, the architecture and capabilities of the telemetry service, and how the telemetry service compares to existing tools. Use of the telemetry service is demonstrated for a multi-language *AI-in-the-loop* workflow where both built-in hardware metrics and custom user metrics are visualized in a Grafana dashboard.

1 Overview

1.1 Problem Statement

There is broad interest in enabling complex workflows like those presented in [3] to run on high performance computing (HPC) resources. Many of these example workflows use machine learning tools, often written in Python, alongside scientific applications - which in contrast to the machine learning tools are often written in C/C++ or Fortran. This distinguishes them from traditional HPC applications that are typically written in a single language. Although C/C++ implementations of machine learning tools are available, they are limited in capability and make it challenging for scientists to rapidly test different approaches. The scales and complex stacks of emerging workflows make it difficult to profile them using traditional HPC or cloud-based telemetry tools. Dragon [5] enables the creation of complex HPC+AI workflows and provides a built-in telemetry feature. Dragon is a high-performance distributed runtime for managing processes and data at-scale. It utilizes high-performance communication objects to enable efficient and transparent management of memory and movement of data - both on- and off-node.

1.2 Dragon Telemetry Service

The Dragon Telemetry Service is integrated into the Dragon runtime. It is designed to be scalable and enable users to visualize hardware metrics as well as custom data streams. Dragon's High Speed Transport Agent (HSTA) allows the telemetry service to take advantage of remote direct memory access (RDMA) primitives when available - making it significantly more scalable than cloud-native telemetry tools on HPC systems. To enable visualization, the Dragon Telemetry Service provides HTTP endpoints that allow Grafana's [7] OpenTSDB data stream to query both hardware metrics and custom metrics.

The telemetry service is composed of the following four components: an `Aggregator`, `Collectors`, `Dragon Servers`, and `Metric Servers`. The `Aggregator` is responsible for handling requests from Grafana. It parses those requests and using `queues` sends requests to the `Dragon Servers` on nodes that data needs to be retrieved from. The `Dragon Server` gathers the required data from the local database and then returns these results to the `Aggregator` via another `queue`. The `collectors` are responsible for collecting the hardware metrics and the `Metric Server` adds data from both the `Collector` and user application to the local database. Figure 1 shows how these components are distributed in a multi-node allocation and how the components interact with Grafana, the user application, and each other.

1.3 Existing Solutions

Tool	Pros	Cons
Work Load Manager (WLM) [8]	no installation	single metric at a time, no visualization, difficult to gather user-defined metrics
LDMS [1]	RDMA-enabled, run-time visualization, can generate custom metrics for Kokkos applications	no Python support, no support for general custom application metrics
Prometheus [9]	run-time visualization	limited scalability due to single-node database and TCP communication
Weights & Biases [4]	integrates with common ML frameworks, end-to-end tracking of ML pipelines, version control for models	does not generalize to non-ML workflows, cannot interact with data directly from workflow
HPCtoolkit [10], CrayPat [6]	easy to generate detailed traces	limited Python support, sampling and tracing data only available for post-processing

Table 1: Comparison of commonly used telemetry tools

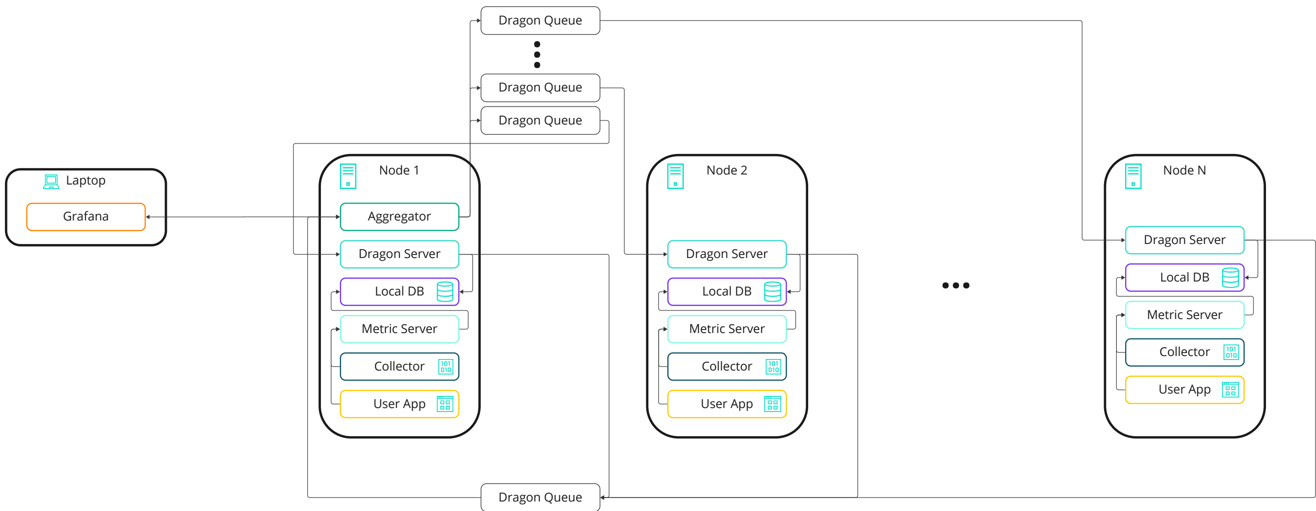

Figure 1: Dragon telemetry service architecture

Any discussion of a new telemetry solution is incomplete without discussing existing tools. Table 1 provides a list of common tools and some of the pros and cons of each. Many of the most popular tools were designed for the cloud with system administrators rather than application developers and scientists in mind. This can make them difficult for users to get access to or install given permission differences. These tools typically do not utilize RDMA-enabled networks and struggle to scale on leadership class HPC systems. An exception to this is the Lightweight Distributed Metric Service (LDMS) [1], which does utilize RDMA primitives and has been run at impressive scales.

Another category of tools, that are built with scientists and HPC users in mind, are statistical profiling tools. Profiling tools provide timing and statistics for different function calls. This information is only available after the application has completed running. These tools serve an orthogonal purpose to the Dragon Telemetry Service. The traces that statistical profiling tools provide are great for optimizing monolithic applications but these profilers have limited support for multi-language workflows. Statistical performance analysis tools do not allow for real-time visualization of a workflow or provide a way for the workflow to react to the performance data.

2 Proof of Solution

Being a part of the Dragon infrastructure, telemetry can be run with minimal changes to existing code. We present the following demo to show what users can achieve with the Dragon Telemetry Service. We run an *AI-in-the-loop* workflow which uses a small model implemented using PyTorch [2] to compute an approximation of $\sin(x)$. It includes an MPI job that computes the Taylor approximation and an MPI job that generates a new training data set when needed. The architecture of the workflow is shown in Figure 2.

To initialize the Dragon Telemetry Service, users simply need to add the argument `telemetry-level=[level]` to the usual Dragon startup command. This launches an additional telemetry head process that runs alongside the workflow. The telemetry `Aggregator` exposes a HTTP API that allows Grafana to request, retrieve, and visualize metrics that have been collected by telemetry `Collectors`. The Dragon Telemetry Service also provides a simple interface to add custom metrics so users can add metrics that are relevant to their workflows. A sample code snippet is shown in Figure 4.

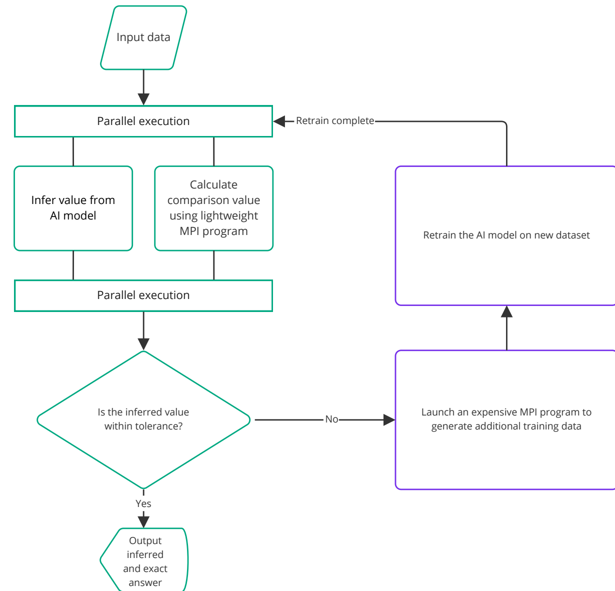

Figure 2: Workflow diagram for *AI-in-the-loop* example. This workflow has multiple MPI programs and manages Python processes while these programs are executing.

Figure 3: Grafana dashboard with Dragon Telemetry Service metrics

```
# import telemetry for custom metrics
from dragon.telemetry.telemetry
    import Telemetry

# initialize telemetry
dt = Telemetry()

# inside target process...
dt.add_data("predicted", float(model_val))
dt.add_data("actual", float(math.sin(x)))

# Close telemetry processes
dt.finalize()
```

Figure 4: Code sample for adding custom metrics to telemetry.

Custom metrics can be assigned a telemetry level. This level is hierarchical, and setting the `telemetry-level=[level]` at launch to a certain value ensures that only those metrics that are less than or equal to that level are collected.

To simplify the entire process, we provide an easily imported Grafana dashboard template. This dashboard can be further customized according to the context of the application. As seen in Figure 3, we have added custom metrics such as the value of $\sin(x)$, the value predicted by the model, and the difference between these two values during every iteration of training. The first panel compares the actual and predicted values. The second panel shows the changes in the difference metric during each iteration. In the last panel we can see that the CPU usage increases significantly while the computationally expensive training and inference jobs are being executed.

3 Future Work

Next steps for the Dragon Telemetry Service include: support for GPU hardware metrics, an API for users to query and analyze metrics from within the workflow, ability to save databases and visualize data offline, and multi-language custom metric support. Investigation of other telemetry tools will continue and any opportunities to integrate with existing tools to improve scalability or ease-of-use will be taken.

4 Acknowledgments

We would like to acknowledge the rest of the Dragon team: Michael Burke, Yian Chen, Eric Cozzi, Zach Crisler, Julius Donnert, Sanian Gaffar, Veena Ghorakavi, Faisal Hadi, Nick Hill, Kent Lee, Peter Mendygral, Davin Potts, Nick Radcliffe, and Ashish Vinodkumar.

References

[1] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. 2014. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 154–165.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan,

Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

[3] Deborah Bard, Taylor Groves, Brandon Cook, Laurie Stephey, Wahid Bhimji, Steve Farrell, Brian Austin, Kevin Gott, Shane Canon, Kristy Kallback-Rose, Jay Srinivasan, Hai Ah Nam, and Nicholas J. Wright. 2023. *Workflow Archetypes White Paper.* Technical Report. National Energy Research Scientific Computing Center, Lawrence Berkeley National Lab, Berkeley, CA 94720.

[4] Weights & Biases. 2024. https://wandb.ai/site.

[5] Michael Burke, Eric Cozzi, Zach Crisler, Julius Donnert, Veena Ghorakavi, Faisal Hadi, Nick Hill, Maria Kalantzi, Kent Lee, Pete Mendygral, Indira Pimpalkhare, Davin Potts, Nick Radcliffe, and Colin Wahl. 2024. *DragonHPC.* https://www.dragonhpc.org/

[6] Steve Kaufmann and Bill Homer. 2003. Craypat-cray x1 performance analysis tool. *Cray User Group (May 2003)* (2003).

[7] Grafana Labs. 2024. https://grafana.com/.

[8] National Energy Research Scientific Computing. 2017-2024. https://docs.nersc.gov/jobs/monitoring/.

[9] Prometheus. 2014-2024. https://prometheus.io/.

[10] Nathan Tallent, John Mellor-Crummey, Laksono Adhianto, Michael Fagan, and Mark Krentel. 2008. HPCToolkit: performance tools for scientific computing. In *Journal of Physics: Conference Series*, Vol. 125. IOP Publishing, 012088.