EPJ Web of Conferences **245**, 04043 (2020) https://doi.org/10.1051/epjconf/202024504043
CHEP 2019

ServiceX

A Distributed, Caching, Columnar Data Delivery Service

B. Galewsky[1,*], *R.* Gardner[2,**], *L.* Gray[5,***], *M.* Neubauer[1,****], *J.* Pivarski[3,†], *M.* Proffitt[4,‡], *I.* Vukotic[2,§], *G.* Watts[4,¶], and *M.* Weinberg[2,‖]

[1]University of Illinois at Urbana-Champaign
[2]The University of Chicago
[3]Princeton University
[4]University of Washington
[5]Fermilab

Abstract. We will describe a component of the Intelligent Data Delivery Service being developed in collaboration with IRIS-HEP and the LHC experiments. ServiceX is an experiment-agnostic service to enable on-demand data delivery specifically tailored for nearly-interactive vectorized analysis. This work is motivated by the data engineering challenges posed by HL-LHC data volumes and the increasing popularity of python and Spark-based analysis workflows.

ServiceX gives analyzers the ability to query events by dataset metadata. It uses containerized transformations to extract just the data required for the analysis. This operation is colocated with the data to avoid transferring unnecessary branches over the WAN. Simple filtering operations are supported to further reduce the amount of data transferred.

Transformed events are cached in a columnar datastore to accelerate delivery of subsequent similar requests. ServiceX will learn commonly related columns and automatically include them in the transformation to increase the potential for cache hits by other users.

Selected events are streamed to the analysis system using an efficient wire protocol that can be readily consumed by a variety of computational frameworks. This reduces time-to-insight for physics analysis by delegating to ServiceX the complexity of event selection, slimming, reformatting, and streaming.

[*]e-mail: bengal1@illinois.edu
[**]e-mail: rwg@uchicago.edu
[***]e-mail: Lindsey.Gray@cern.ch
[****]e-mail: msn@illinois.edu
[†]e-mail: pivarski@princeton.edu
[‡]e-mail: masonlp@uw.edu
[§]e-mail: ivukotic@uchicago.edu
[¶]e-mail: gwatts@uw.edu
[‖]e-mail: mweinberg@uchicago.edu

1 Introduction

Data organization in HEP experiments often follows a standard workflow, starting from raw detector data or simulated data, which is processed by reconstruction algorithms into higher-level physics objects. This reconstructed data is stored in ROOT files in a nested branch structure and used as input to analysis software to produce physics insights. While the reconstruction step is typically performed centrally and a standardized output is made available to the entire collaboration, the analysis software is often written by small groups or individual analysts, with idiosyncratic selection and processing steps.

This approach carries with it a number of issues. Because transmission of reconstructed ROOT files via wide area network is expensive, the analysis step must often be performed via jobs running on the grid, which involve substantial overhead and require care and attention from the user. Further, the output of these jobs is immutable, so changes to the analysis structure often require rerunning the analysis code from scratch. Lastly, the format itself can impose barriers to insight: too much craft is required to extract features (e.g. for ML applications) and the tools required to interact with ROOT files are not generally transferable to other formats.

2 ServiceX overview

ServiceX, a part of the IRIS-HEP Data Organization, Management, and Access (DOMA) effort, is an experiment-agnostic service that seeks to address these problems by enabling on-demand data delivery, specifically tailored for nearly-interactive, high performance array-based and Pythonic analyses. It provides uniform backend interfaces to data storage services and frontend (client-facing) service endpoints for multiple different data formats and organizational structures.

It is capable of retrieving and delivering data from data lakes, using Rucio to find and access the data. The service is capable of on-the-fly data transformations to enable data delivery in a variety of different columnar formats, including small flat ROOT files and streaming Apache Arrow buffers. In addition, ServiceX includes pre-processing functionality for event data and preparation for clustering frameworks (e.g. Apache Spark). Eventually, it will be able to automatically unpack compressed formats, potentially using hardware accelerated techniques, and will prefilter events so that only useful data is transmitted to the user.

2.1 ServiceX features

ServiceX orchestrates containerized experiment-approved frameworks in order to produce data in columnar formats. It includes a REST server interface that can be colocated with the data and can easily be deployed to a local Tier 3 cluster. The service supports simple event selection to enable users to extract only required subsets of data, and streams the output to analysis code in the form of uproot-based awkward arrays [3].

This output is managed by the Kafka [2] message broker, illustrated in figure 1. The message broker caches results in order to make them available for instant replay so output can be quickly analyzed and reanalyzed multiple times.

The service is also capable of using Object Store to manage the output as a series flat ntuples stored inside small ROOT files or HDF5 files. This effectively allows ServiceX to provide an "ntuples-as-a-service" functionality.

Finally, ServiceX is designed to be completely transactional, allowing data in the request to be processed in parallel while ensuring all data is processed exactly once. This means there is no danger of missed or double-counted events, even in the event of failure of one of the transformation threads.

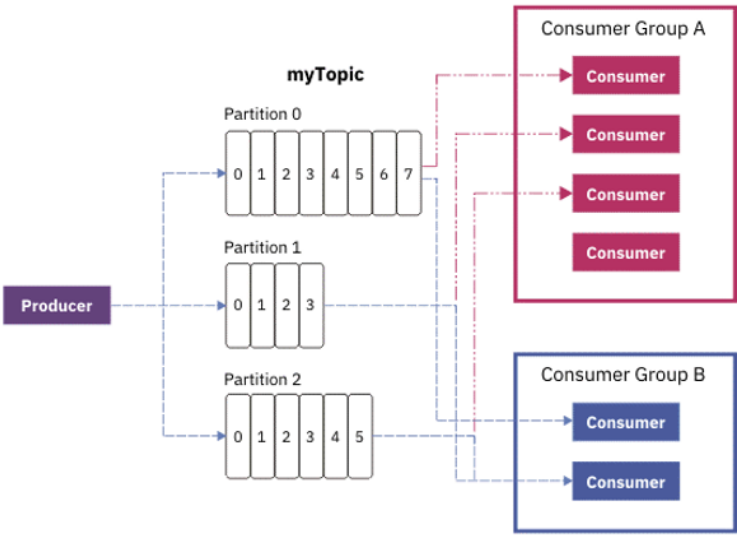

Figure 1. Kafka uses a fault-tolerant, low-latency publish-subscribe messaging system.

2.2 Simple transform requests

An example transform request is presented below for illustration:

```
{
  "did":  "mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge",
  "columns":  "Electrons.pt(), Electrons.eta(), Muons.eta(), Muons.phi(), Muons.e()",
  "image":  "sslhep/servicex-transformer:v0.1",
  "result-destination":  "kafka",
  "kafka":{
    "broker":  "servicex-kafka-1.slateci.net:19092"
  },
  "chunk-size":  9000,
  "workers":  17
}
```

Here the `did` field gives the ID of the dataset in Rucio, while `columns` gives the list of columns requested by the user. The `image` field has the versioned image to be used for the transformation. This piece can be easily swapped out for different transformers in order to transform input files with different formats. Next the request specifies the Kafka broker to which the output will be assigned. The last two fields indicate that each message in the Kafka topic will contain 9000 events, and a total of 17 transformers will be spun up to process the dataset.

The output of this transformation request arrives as an Arrow table comprising 147,000 rows (one for each event in the dataset) and 8 columns (corresponding to the attributes specified in the request). Each entry contains an array whose length is the number of particles (i.e. electrons or muons) in the event. The output is intended to work seamlessly with the uproot framework.

3 ServiceX implementation

3.1 ServiceX architecture

The current architecture employed in ServiceX is illustrated in figure 2. The service relies on a number of interconnected microservices run in Kubernetes [4] which communicate via the REST server. The DID finder is responsible for querying Rucio for the location and metadata of the files in the user request. The preflight check then takes this information and performs some preliminary transformations to verify the request is properly formed (e.g. refers to an existing dataset and requests columns that can be found in the files). The transform manager is responsible for spinning up the appropriate number of transformers to transform the dataset, and ensures that each file is claimed by exactly one transformer. Finally, the transformer itself is responsible for converting the input format into the requested output.

The input files to be transformed correspond to messages in the RabbitMQ [5] broker, which is responsible for ensuring that the service is fully transactional. Destination endpoints are provided for both MinIO [6] and Kafka [7]; in the former case output can be retrieved as files from the object store, and in the later case the output can be streamed directly to a user.

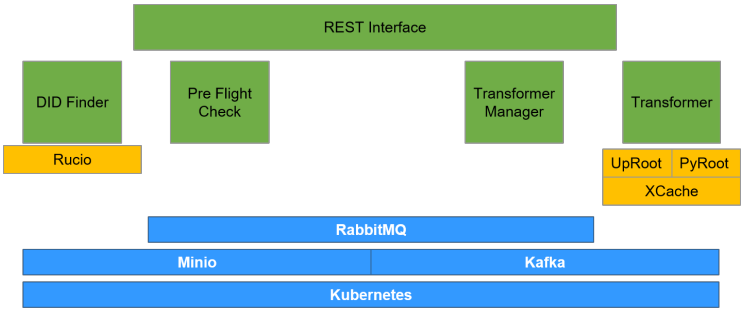

Figure 2. Architecture for ServiceX version 1.

3.2 Python transformers

The transformers are the engine of the service, responsible for changing the format of the input files into an analysis-friendly columnar output and sending this output to the message broker.

There are currently images available for transformers compatible with two different input formats. First there is a transformer for the ATLAS xAOD/DAOD reconstruction format, based on the ATLAS Analysis Base software. It employs pyROOT to access individual branches, and requires the use of `for` loops due to the embedded data structures. The transformer has some limitations: the need to loop over events in pyROOT makes the transformation quite slow, and the format contains no information about the calibrated physics objects. Further, filtering is not yet implemented in this transformer.

The second available transformer can handle ROOT files containing flat trees (i.e. trees whose branches are all simple arrays). This makes them appropriate both for typical end-stage analysis ntuples and for the CMS nanoAOD format, which features this flat tree structure. Here the transformation can be performed directly with uproot tools, so the process is much faster. This transformer also does not currently have in-place filtering implemented.

3.3 Scalability testing

The performance of ServiceX also needs to be measured in terms of its ability to scale up or down in response to multiple input parameters. It is necessary for the service to be able to handle multiple concurrent users requesting many distinct columns of data from different large input datasets simultaneously. In order to ensure this functionality the service has several requirements: to make the system robust against individual pod failures, the transform manager must have transactional integrity, guaranteeing that each file in the request is transformed exactly once. Further, the system must be capable of horizontally scaling the number of transformation pods automatically in response to both the request size and the system load. Moreover, the system needs a scheduling mechanism to handle simultaneous requests and facilitate the efficient sharing of resources among different users of the service.

Smooth and robust operation of the service at scale depends on system performance under several metrics, including CPU usage, network usage, and memory consumed by running transformers. A key piece of the development effort of ServiceX centers on accurately evaluating and optimizing these metrics under different types of input loads.

4 What's next?

ServiceX is being developed as an open source project intended to meet a broad range of analysis needs within HEP. Consequently we welcome involvement from the HEP community. Version 1 is available today and can be found at [1]. It builds on an extensive ecosystem of open-source technologies, shown in figure 3. We invite users to try out the ServiceX demo and to reach out to us about providing data for your analysis.

Figure 3. Multiple open-source technologies were used in the development of ServiceX.

In addition, the project features many exciting new developments, including the use of very fast C++ transformers and the implementation of a selection language to enable in-place filtering. Those who are interested are encouraged to join the discussion on new features, and to help build ServiceX!

4.1 Version 2–with C++ transformers

ServiceX version 2 features transformers based on C++ code, and became available in January 2020. This version accepts SQL-like selection statements specifying requested output columns as well as filtering requirements, and also enables the implementation of experiment-specific calibrations on the raw physics objects. The architecture for this version of ServiceX

differs slightly from the current architecture in the addition of the code generator service, which is responsible for converting these statements into compiled C++ code and passing this to the transformation manager (see figure 4).

In the case of the ATLAS xAOD format, this enables the transformers to use ATLAS-specific code running in the EventLoop framework. They perform the transformations much faster than the previous implementation, and have access to the calibration information.

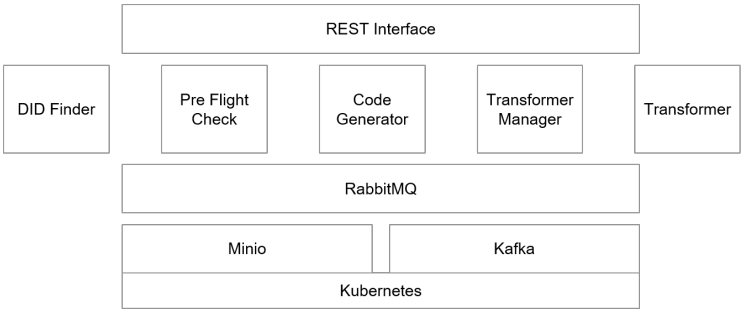

Figure 4. Architecture for ServiceX version 2.

4.2 Selection code

An example of a typical selection statement is shown below.

```
xAOD.Where(
  lambda e:
    e.jet_pT.Where(lambda pT:
    pT > 1000).Count() > 0)
  .Select('lambda e:
    [e.eventNumber, e.CalibJet_pT]'
)
```

This statement will extract the event number and calibrated transverse momenta of all jets in the event, provided there is at least one jet in the event with an uncalibrated transverse momentum greater than 1 GeV.

References

[1] *https://github.com/ssl-hep/ServiceX*
[2] *https://ibm.github.io/event-streams/about/key-concepts/*
[3] *https://github.com/scikit-hep/uproot*
[4] *https://kubernetes.io/*
[5] *https://www.rabbitmq.com/*
[6] *https://min.io/*
[7] *https://kafka.apache.org/*