Yoda: a MPI-based Event Service implementation for HPCs

(Draft)

1. Introduction

This document describes high level design of Yoda: a MPI-based lightweight Event Service for running on the compute nodes of HPC machines. A schematic view of such Event Service implementation is shown on Figure 1.

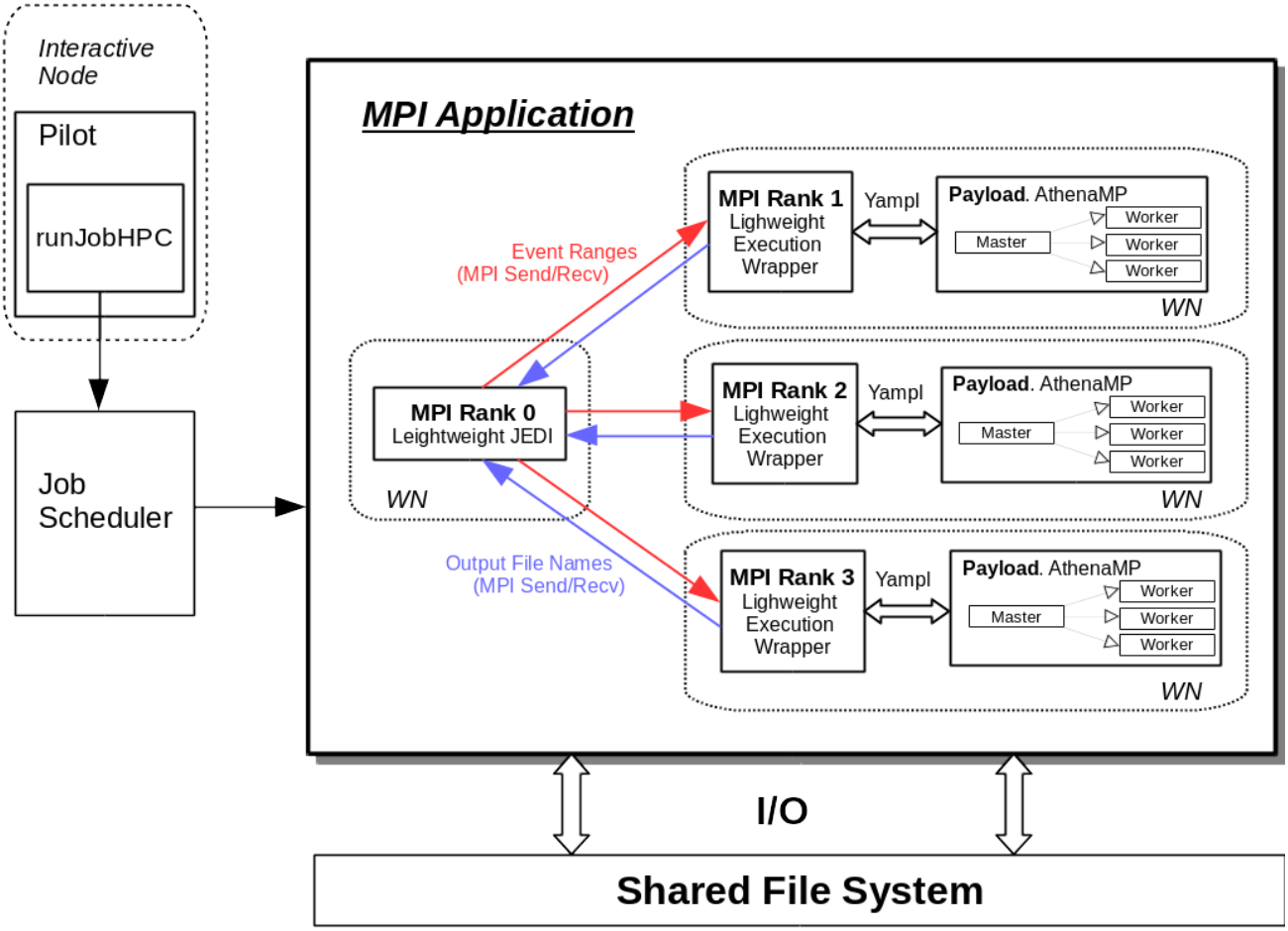

Figure 1. Schematic view of Yoda (MPI-based lightweight Event Service)

From the job submission perspective, Yoda application looks like a single MPI job, which can be submitted for execution as follows:

```
aprun -n X -N 1 -d <number of cores per node> yoda.py [input parameters]
```

Here, X>1 is the number of MPI Ranks for the given job, and each compute node assigned to the job runs only one Rank. We distinguish two types of MPI Ranks within Yoda application:

1. **Rank 0**. Lightweight JEDI
2. **Rank N, N=1..X-1**. Lightweight Execution Wrapper (Pilot)

In the subsequent section we will go over various components of Yoda in more details.

2. Yoda Components

2.1. JEDI (Rank 0)

Rank 0 within the Yoda application is a lightweight implementation of JEDI, which has mostly the same responsibilities as the PanDA/JEDI in the conventional Event Service. They include: generation of event ranges and their delivery to Pilot/Execution Wrapper applications on the compute nodes (Rank 1 … N-1), bookkeeping and monitoring for the generated event ranges, initiation of retries (if needed) and possibly also starting the merging jobs on the same compute node (to be determined).

The important differences from the conventional PanDA/JEDI include:

1. The communication between Rank 0 and the lightweight execution wrappers (Rank 1 … N-1) will be implemented using MPI point-to-point communication mechanisms (as opposite to the communication over HTTP protocol in the conventional Event Service). On the other hand, the message formats and the message contents for both implementations of the Event Service – conventional and Yoda – are expected to be identical.
2. The Event Table and the Job Table for the Yoda job will be stored inside SQLite database files residing on the shared file system (as opposite to the Oracle database used by the conventional Event Service). The information from these SQLite files can be passed outside HPC system to the central PanDA services for external monitoring of the running Yoda job.

2.2. Execution Wrapper

The Execution Wrapper (Rank 1 … N-1 within Yoda application) is expected to be a lightweight script, which will perform basically the same functions as the Pilot on the worker nodes inside conventional Event Service. Upon starting on the compute node, the execution wrapper launches two processes: AthenaMP payload job and the Token Extractor. Once the payload has gone through the initialization, the execution wrapper starts retrieving event ranges from Rank 0 using MPI point-to-point communication mechanisms. The retrieved event ranges are then passed over to the payload job using Yampl channel, exactly the same way it is implemented for the conventional Event Service. The payload reports the completion of event range processing back to the execution wrapper using the same Yampl channel. Finally, the range completion report – including the location of corresponding output file – is delivered to the Rank 0 using MPI communication mechanisms.

The important differences from the conventional Pilot/Job Launcher include:

1. Communication with the Rank 0 (PanDA/JEDI) are done using MPI mechanisms, although the message format and message contents are expected to be the same as in the conventional Event Service.
2. Rank 1 … N-1 does not attempt to store the output files produced by AthenaMP workers, as they are already available on the shared file system, it only reports their locations back to Rank 0.

Given the modular structure of the Pilot, it is expected that we can reuse already existing code components for implementing Rank 1 … N-1 for Yoda. The important change will be to substitute HTTP-based communication with MPI-based one.

2.3. AthenaMP payload

No changes are expected for the AthenaMP payload. The payloads running within the conventional Event Service and within Yoda will be absolutely identical.

3. Running

3.1. Before submitting the job

Before submitting the job all necessary input files need to be copied over to the shared file system. The Token Extractors read event tokens from TAG files, thus such TAG files also need to be created in advance and placed to the shared file system.

3.2. Submitting the job

The whole application will be submitted to the batch system as a single MPI-job:

```
aprun -n X -N 1 -d <number of cores per node> yoda.py [input parameters]
```

Here the yoda.py leverages Python interface to MPI (for example: `mpi4py`) in order to find out its MPI rank. Depending on the retrieved value, the script starts either the lightweight JEDI (rank==0) or the execution wrapper (rank!=0). A skeleton python code for such wrapper script is shown below:

```
from mpi4py import MPI
mpirank = MPI.COMM_WORLD.Get_rank()
if mpirank==0:
     # Run Lightweight JEDI
else
     # Run Execution Wrapper
```

3.3 Handling of output files

Each worker process of the AthenaMP payload job produces a number of output files, one output for each event range. These files are written out to the shared file system and their location is first reported by the AthenaMP to the Execution Wrapper, and then the latter one passes this information over to the lightweight JEDI.

After that it is up to JEDI/Rank0 to decide how to proceed with merging. One option would be to initiate a merger job(s) during the execution of Yoda. If this option is found to be not optimal (running I/O intensive task on a diskless compute node on HPC), then the Rank0 can simply collect the information about all output files and pass it over to the job submitter application (PilotHPC ?), which will then proceed with merging after the Yoda job has finished.

Finally, another possibility for handling output files is to follow the approach of the conventional Event Service and upload them to an external aggregation point, like an Object Store, where they will be merged into larger output files later on.