Artifacts

2024-05-29

What are “calibration” artifacts?

Originally there were just two (since Athena times):

- Field maps (B field of the MARCO solenoid, used in simulation)
- Material maps (material density in the inner tracker, used in reconstruction)

Additional uses for the same system:

- GDML files for the components of the barrel HCal (used in simulation)
- Calibration for ONNX ML example (used in reconstruction)
- PID LUT tables (used in reconstruction)
- Calibration for TMVA ML at low-Q2 tagger (in development for reconstruction)

This is quite a few, they now go to compact/calibrations.xml

How are they implemented?

FileLoader DD4hep plugin is instantiated either

- directly in <plugins> section

https://github.com/eic/epic/blob/main/compact/calibrations.xml

```xml
<plugin name="epic_FileLoader">
  <arg value="cache:$DETECTOR_PATH:/opt/detector"/>
  <arg value="file:calibrations/tmva/LowQ2_DNN_CPU.weights.xml"/>
  <arg value="url:https://raw.githubusercontent.com/eic/epic-data/e441e5d88d0a44a1e407d3083914ac1a7de26f41/tmva/LowQ2_DNN_CPU.weight
</plugin>
```

- or in the detector implementation (steered by custom XML)

https://github.com/eic/epic/blob/main/compact/fields/marco.xml

```xml
<field type="epic_FieldMapB" name="GlobalSolenoid" field_type="magnetic" coord_type="BrBz"
       field_map="fieldmaps/MARCO_v.6.4.1.1.3_1.7T_Magnetic_Field_Map_2022_11_14_rad_coords_cm_T.txt"
       url="https://github.com/eic/epic-data/raw/64b7ca6306b138b7f000e696c82bd8f72db1da56/MARCO_v.6.4.1.1
       cache="$DETECTOR_PATH:/opt/detector"
       scale="1.0">
```

https://github.com/eic/epic/blob/main/src/FieldMapB.cpp

```cpp
std::string field_map_file  = x_par.attr<std::string>(_Unicode(field_map));
std::string field_map_url   = x_par.attr<std::string>(_Unicode(url));
std::string field_map_cache = getAttrOrDefault<std::string>(x_par, _Unicode(cache), "");

EnsureFileFromURLExists(field_map_url, field_map_file, field_map_cache);
```

How does this work?

1. npsim or EICrecon loads geometry (e.g. epic_cratelake.xml)
2. For each defined epic FileLoader the instance file is found:
   a. In /opt/detector if you are using the container
   b. Downloaded from the URL
3. File is placed to **<prefix>/<hash>** where **prefix** is usually **“calibrations/”** and hash is calculated over the original URL
4. **<prefix>/<hash>** is symlinked to the final destination **<prefix>/<name>**
5. Files are loaded from the filesystem by the respective geometry/factory implementation(s)

What can be improved?

- The occasional bug

```
FileLoader     ERROR unable to link from calibrations/foobar to calibrations/de9b04819a3430af
FileLoader     ERROR check permissions and retry
```

- Shared state in the current working directory (potential for race conditions)
- EICrecon doesn't know if the file on disk was actually updated by the geometry, or a leftover (potential consistency issues)
- No way to query information about the calibrations (machinery that populates /opt/detectors is complicated)
- Using new EICrecon implies that one has to use new geometry (or factories that can't find their files would have to self-disable - e.g. missing PID LUTs disable reco particles)
- epic geometry is immutable (what is the way to deliver calibrations from running material map scans or ML training on CI?)
- Every artifact is processed on each startup even if unused (ugly)

What Are Artifacts?

Config-dependent objects or quantities needed during simulation or reconstruction

- Magnetic field (11 MB MARCO, 4 MB lumi)
- Acts material map (5 MB)
- Calorimeter sampling fractions (4 B)
- Trained machine learning models (2 MB low-Q2)
- PID lookup tables (45 MB)
- …

Typically a ‘calibration’ step (broadly construed) is necessary to determine the correct artifact to use for a specific ‘run’.

Artifact Distribution Through Geometry

Current approach: geometry distributes artifacts (historically motivated)

- Magnetic field: unique URL with e.g. commit hash in epic-data or wiki (don't): https://github.com/eic/epic-data/raw/64b7ca6306b138b7f000e696c82bd8f72db1da56/MARCO_v.6.4.1.1.3_1.7T_Magnetic_Field_Map_2022_11_14_rad_coords_cm_T.txt (URL hash: `65697e9cb0b6625b`, file: `fieldmaps/<hash>`)
- Caching by URL hash allows storing multiple versions with same filename
    - https://eicweb.phy.anl.gov/EIC/detectors/athena/uploads/6253bfcb92dd6bc236dd9a06084614ba/material-maps.cbor -> `calibrations/657d53d737220c61`
    - https://eicweb.phy.anl.gov/EIC/detectors/athena/uploads/77bd16bef5d811f31256279a3601c9d5/material-maps.cbor -> `??? (old)`
- Downloaded when used in geometry by DD4hep plugin (`FileLoader`)

Artifact Distribution Through Geometry: Drawbacks

- Not all artifacts are tightly coupled to geometry changes; e.g. epic/compact/calibrations.xml contains all PID LUT definitions, as well as some ONNX and TMVA files…
- ‘Calibrations’ artifacts often require geometry to determine. Workflow:
  - Geometry is tagged and released.
  - ‘Calibration’ is run to determine artifacts (cf. calibration turnaround time discussions).
  - Reconstruction proceeds with ‘calibration’ artifacts.

Conclusion: need an independent way to distribution calibration artifacts, which does not require editing the geometry but allows referencing it.

Storing URI references to permanent(!) storage instead of storing the large objects is workable solution. Allow filesystem (incl. cvmfs) cache for O(10k) jobs at once.

HSF Conditions DB White Paper

“A **global tag [GT]** is the top-level configuration of all conditions data. For a given **payload type** and a given **interval of validity [IOV]**, a global tag will resolve to one, and only one, conditions data payload. The **Global Tag** resolves to a particular payload type **Tag** via the **Global Tag Map** table. A payload type **Tag** consists of many non-overlapping intervals of validity or entries in the **IOV** table. Finally, each entry in the **IOV** table maps to a payload via its unique hash key in the **Payload** table.”

Separation of storing the object from storing reference to the object: e.g. can store entire database as sqlite and ship with jobs.

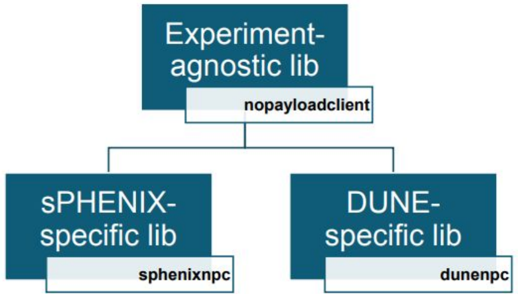
Ref: https://arxiv.org/pdf/1901.05429

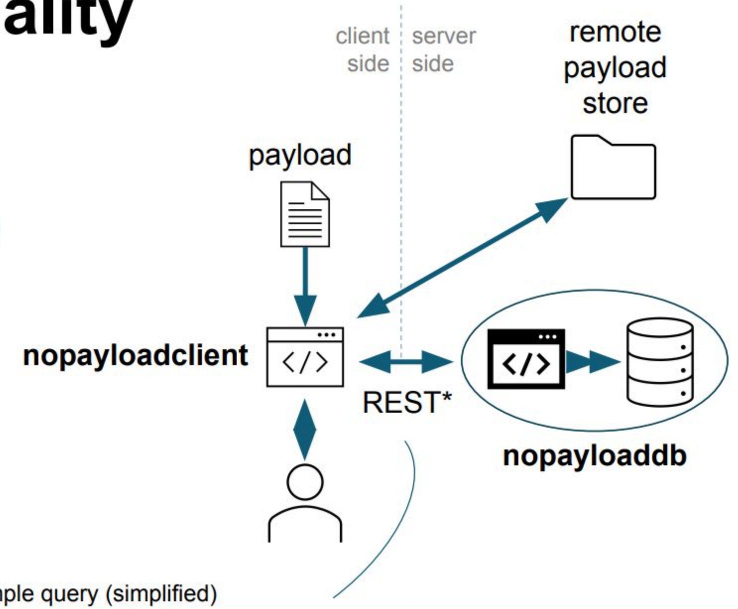
HSF Conditions DB Reference Implementation

Features & Functionality

nopayloadclient:

- Client-side stand-alone C++ tool
- Communicates with nopayloaddb (server)
- Local caching
- Handling of payloads





*Example query (simplified)

```
curl http://<host>/api/payloadiovs/?gtName=test_gt&iovNum=42
-> {type_1: url_1, type_2: url_2, …}
```

Ref: https://indico.cern.ch/event/1343984/timetable/?view=standard#48-towards-an-hsf-reference-im

HSF Conditions DB Reference Implementation

Spin up DB implementation:

```
git clone https://github.com/BNLNPPS/nopayloaddb
cd nopayloaddb
docker compose up -d
```

Test the DB implementation:

~~git clone https://github.com/BNLNPPS/nopayloadclient~~

```
eic-shell> cli_npc
(needs NOPAYLOADCLIENT_CONF)
cli_npc checkConnection
cli_npc createGlobalTag example_gt
```

HSF Conditions DB Reference Implementation

EICrecon #1428

```
#include <nopayloadclient/nopayloadclient.hpp>
Payload_service::Payload_service(JApplication *app)
: m_client("EICrecon") {
    m_application = app;
}
```

CI spins up nopayloaddb test instance for testing

Summary

We are alreading using URIs to locate artifacts.

We have a vision (HSF Conditions DB White Paper).

We have an implementation (HSF Conditions DB Reference Implementation).

We have a testing platform (EICrecon #1428).

We do still need permanent storage (Rucio?).