Introduction to JUNO Offline Software Framework

Xingtao Huang

Shandong University
Sep. 19, 2016

On behalf of JUNO Offline Framework Group
Outline

- Offline Software Environment
- Overview of Offline Software System
- Main Components of Framework
  - Algorithm
  - Service
  - Task
  - Incident
  - Event Data Model
  - Event Buffer
  - Configuration
  - Logging

- Summary
Offline Software Environments

- **Programming language:** hybrid programming of C++ and Python
  - C++: main part implementation
  - Python: job configuration interface

- **Packages management tool:** CMT(Configuration Management Tool)
  - Help developers to compile packages easily
  - Help users to setup the environment for running the application easily

- **Operation System:** Linux
  - Official support: Scientific Linux (Now SL 6+ gcc4.4)
  - Some testing on other OS (Ubuntu, Debian …)

- **Codes Management:** SVN
  - Keep the history of code evolution
  - Synchronization and sharing between developers
  - Tag and release
Consists of 3 Parts (Projects):

- **SNiPER**: the framework
- **Offline**: include event model, geometry, generators, simulation, reconstruction, etc.
- **External Libraries (EI)**: frequently used third-party software and tools
Three Layers’ Infrastructure

User’s Application Layer

Do not care where the data comes from

Users’ algorithm:
1. get data from event buffer
2. execute calculation
3. put results back to buffer

Do not care where the data will go

Core Software Layer

① Framework architecture management
- Interfaces, Services, Input/Output, UI, Logging, etc.

② Event buffer management
- Manage event data
- Send data to algorithms
- Get results from algorithms

③ Application management
- Load and plugin algorithms
- Manage and execute algorithms

Python UI Layer

run a batch job or interactively debug a module
Main components of SNiPER

From Users’ point of view:

- Algorithm
- Service
- Task
- Incident
- Event Data Model
- Event Buffer
- ROOT Input/Output

\[\text{Algorithm, Service and Task follow modular design}\]

- Dynamically Loadable Element (DLElement)
- Low couplings between each other
- Support parallel development of applications
Algorithm

- The smallest unit of users’ codes:
  ➞ perform event calculation, i.e.
    - Vertex reconstruction
    - Correlation analysis
Algorithm

- The smallest unit of users’ codes:
  - perform event calculation, i.e.
    - Vertex reconstruction
    - Correlation analysis

- SNiPER provides user interface (AlgBase)
  - User’s algorithm must inherit from AlgBase
    - Its constructor takes one std::string parameter.
    - Must implement 3 member functions
      - bool initialize() : called once per Task (at the beginning of task)
      - bool execute() : called once per Event
      - bool finalize() : called once per Task (at the end of task)
Algorithm

- The smallest unit of users’ codes:
  - perform event calculation, i.e.
    - Vertex reconstruction
    - Correlation analysis

- SNiPER provides user interface (AlgBase)
  - User’s algorithm must inherit from AlgBase
    - Its constructor takes one std::string parameter.
    - Must implement 3 member functions
      - bool initialize() : called once per Task (at the beginning of task)
      - bool execute() : called once per Event
      - bool finalize() : called once per Task (at the end of task)

- One data processing (or Task) consists of one or more algorithms
Service

- Usually a piece of codes for common use:
  - Histogram Service
  - Random Service
  - Geometry Service
  - Root Input/Output services, etc.

- User interface, SvcBase, is provided by SNiPER
  - New services must inherit from it
Service

- Usually a piece of codes for common use:
  - Histogram Service
  - Random Service
  - Geometry Service
  - Root Input/Output services, etc.

- User interface, SvcBase, is provided by SNiPER
  - New services must inherit from it

- One data processing (or Task) usually use one or more services
  - For example, CalibTask uses I/O Svc/ DataBase Svc/ Geometry Svc etc.
Service

- Usually a piece of codes for common use:
  - Histogram Service
  - Random Service
  - Geometry Service
  - Root Input/Output services, etc.

- User interface, SvcBase, is provided by SNiPER
  - New services must inherit from it

- One data processing (or Task) usually use one or more services
  - For example, CalibTask uses I/OSvc/ DataBase Svc/ Geometry Svc etc.

- User interface, SniperPtr, is used to retrieve a service with its name

```cpp
// Example
SniperPtr<RootWriter> m_rw("RootWriter"); // using name of Service
...... // checking service’s validation
m_rw->attach("FILE1", m_tree1); // use the service
```
A lightweight application manager:

- Manage its algorithms/services
- Control algorithm execution
- Configure input/output Systems
- Manage Event Buffer etc.
**Task**

- **A lightweight application manager:**
  - Manage its algorithms/services
  - Control algorithm execution
  - Configure input/output Systems
  - Manage Event Buffer etc.

- **A job may have more than one Tasks**
  - TopTask and SubTask
**Task**

- **A lightweight application manager:**
  - Manage its algorithms/services
  - Control algorithm execution
  - Configure input/output Systems
  - Manage Event Buffer etc.

- **A job may have more than one Tasks**
  - TopTask and SubTask

- **Each task can be configured individually**
  - Set Time window of its buffer
  - Add new algorithms/services
Sequential Execution and Jump Execution

- **Task controls event processing procedure (Event Loop)**
  - Performs sequential execution of its algorithms
  - Execution order is defined when adding algorithms into the Task

- **SubTask can be executed by firing Incidents**

- **Both sequential and jump executions are implemented**
Multi-Task Use Cases

M.C. Data Production:

- **Event Mixing** (IBD with N types of backgrounds)
  - TopTask + N SubTasks
  - Each SubTask reads event data from one background sample
  - TopTask mixes all events together according to their rates
Multi-Task Use Cases

M.C. Data Production:

- **Event Mixing** (IBD with N types of backgrounds)
  - TopTask + N SubTasks
  - Each SubTask reads event data from one background sample
  - TopTask mixes all events together according to their rates

- **Events Splitting** (IBD)
  - 1 event entry in Gen. and Sim.
  - 2 event entries from Trigger
  - RecTask loops twice for one SimTask
Data Processing and Event Data Model

- Defines the data unit to be processed in offline data processing
- In form of C++ classes
- Finally they are converted into persistent type and saved into disk
JUNO EDM is developed based on ROOT

- ROOT is powerful and popular in HEP experiments
- From data management point of view, ROOT provides:
  - Objects I/O (Streamers)
  - Run Time Type Identification (RTTI)
  - Inspection
  - Drawing etc.

All event objects are derived from ROOT TObject

Many benefits:
- ROOT I/O streamers simplify framework I/O system
- Same definition for transient event data and persistent event data
- ROOT Class schema evolution makes multiple versions compatible
Design of Event Data Model

- Two-layer design to reduce the I/O burden
  - Header Object
  - Event Object
- The small and characteristic data are defined in the Header Object
  - Run/Event id, Energy, Momentum etc.
  - Serve as ‘tag’ data for selection
- The large and detailed data are defined in Event Object
- The goal is to load as little as possible data and speed up event selection
- SmartRef is a new type of Smart Reference Pointer
Design of Event Data Model

- EvtNavigator is developed as the event catalog
  - Serve as index of all event objects
  - Provide convenient navigations between the event data at different data processing stages
  - User can get event objects of one event at different processing stages via data paths
Implementation of Event Data Model

IEvtObject

TObject

int_t_uniqueID

EventObject

int mRefCount

EvtNavigator

vector<SmartRef*> m_header

TTimStamp* m_timestamp

HeaderObject

int m_eventID

int m_runID

GenHeader

SimHeader

CalibHeader

RecHeader

PhyHeader

GenEvent

SimEvent

CalibEvent

RecEvent

PhyEvent
Event Buffer (I)

It’s a memory place allocated dynamically to hold multiple events within certain Time Window.

- Each Task has its own Event Buffer
- Event Buffer is identified with the name of Task
- User’s Algorithms get event data from Event Buffer
- User’s Algorithms put/update event data into Event Buffer
Event Buffer (II)

- Size of Event Buffer changes with Time Window
- Time window is configured in the Job Script
- Events within the buffer are deleted or added with the current event
- Easy to perform *IBD correlation analysis* in future
Event Layout in Buffer and Root File

- **In Event Buffer**
  - Directory Structure
  - Unique path

- **In Root File**
  - Tree Structure
  - Tree/branch name
    - same with buffer path

```
/Event
/Event/Gen
......
/Event/Sim
/Event/Sim/SimHeader
/Event/Sim/SimCDEvent
/Event/Sim/SimTTEvent
......
```

```
Event
  Gen
  Sim
    SimHeader
    SimCDEvent
    SimTTEvent

Meta
  EvtNavigator
  FioMetaData
  JobConfig

JunoGeom
```
User interface, SniperDataPtr, is provided to retrieve the Event Buffer and Get Current Event with the path

```cpp
SniperDataPtr<JM::NavBuffer> navBuf(getScope(), "/Event");
......
m_buf = navBuf.data();
JM::EvtNavigator* nav=m_buf->curEvt();
```
User Interface to Access Buffer/Event

- **User interface, SniperDataPtr, is provided to retrieve the Event Buffer and Get Current Event with the path**

  ```
  SniperDataPtr< JM::NavBuffer > navBuf( getScope(), "/Event" );
  ...
  m_buf = navBuf.data();
  JM::EvtNavigator* nav = m_buf->curEvt();
  ```

- **The Service, BufferMemMgr, is used to put/adopt event with a unique path**

  ```
  SniperPtr<IDataMemMgr> mMgr( getScope(), "BufferMemMgr" );
  JM::EvtNavigator* nav = new JM::EvtNavigator();
  mMgr->adopt( nav, "/Event" );
  ```
Property

- Configurable variable at run time

- Declare a property in DLElement(Alg,Svc, Task)

```c++
//suppose m_str is a string data member
declProp("MyString", m_str);
```

- Configure a property in Python script

```python
alg.property("MyString").set("string value")
```

- Types can be declared as properties:
  - scalar: C++ build in types and std::string
  - std::vector with scalar element type
  - std::map with scalar key type and scalar value type
_logging

- **SniperLog**: a simple log mechanism supports different output levels
  - 0: LogTest
  - 2: LogDebug
  - 3: LogInfo
  - 4: LogWarn
  - 5: LogError
  - 6: LogFatal

- Each DLElement(Alg,Svc, Task) has its own LogLevel and can be set at run time

- The output message includes more information, such as
  - where it happens
  - message level
  - message contents

- very helpful for debugging
Summary and Plan

- SNiPER has been designed and developed from scratch
- 9 official major versions has been released
- Applications have been developed based on SNiPER

Future Plan

- More services and tools to be added (End of 2016)
  - Database Service, Validation Toolkits …
- Parallel computing to be implemented
  - Multi-threads with Intel TBB is being under development (2017)
  - MPI is a long-term planning (2019)
- Set up the whole data processing chain
  - Perform commissioning
  - Optimize some functionalities
- Investigate upgrades of some tools and technologies
  - CMake
  - C++ 11
  - ……
Thanks a lot!