Transient condition storage

Hadrien Grasland

LAL – Orsay

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

Introduction

- Review of the ATLAS ICondSvc proposal revealed a number of fundamental shortcomings:
  - Design is complex and largely undocumented
  - Leaks many implementation details
  - Strongly tied to ATLAS infrastructure
- Benedikt and I hence decided to propose a new interface:
  - Take into account ATLAS' use cases and requirements
  - Design interface so that it can wrap ATLAS infrastructure
  - But account better for the wider Gaudi ecosystem

Interface design process

- High-level overview given at LHCb computing workshop[1]
- To produce concrete interface proposal and prove feasibility, started writing a prototype implementation
- Prototype uses C++/Boost, written outside of Gaudi
  - Enables faster progress towards AIDA-2020 milestone
  - More interesting for the linear collider community
- Focus so far: transient condition storage

[1] https://indico.cern.ch/event/464394/contributions/2344285/attachments/1370811/2078980/Conditions_20161114.pdf

Problem statement

- Some entities produce and consume condition data
  - IO services, algorithms...
- Other entities hold condition data in RAM
  - DetectorStore, DD4Hep, ATLAS ConditionStore...
- We shouldn't need to know or care who does what
  - Provide standard interface to condition storage
  - Use handles for data access & dependency tracking

TransientConditionStorageSvc

- Standardized interface to in-RAM condition storage
- Sets storage bounds (N sets of conditions/unbounded)
- Tracks condition producers & consumers
  - Can provide this information to the Gaudi scheduler
- Manages condition storage for events
- Abstracts storage using handles and slots
  - Handle = “Mean to read or write a condition”
  - Slot = “One version of the condition data”

Bounded condition storage

- Query implementation capabilities

```
// This method indicates the maximum storage capacity supported by the active implementation.
// If no conceptual limit exists, UNBOUNDED_STORAGE will be returned.
static size_t max_capacity();
```

- Setup condition storage accordingly

```
TransientConditionStorageSvc(       ConditionSvc & conditionService,
                              const size_t         capacity );
class UnsupportedStorageCapacity : public ConditionPrototypeException { };
```

- Monitor storage usage

```
// This method indicates how many condition storage slots are currently available.
// If no conceptual limit exists, UNBOUNDED_STORAGE will be returned.
size_t availableStorage();
```

Condition dataflow tracking

- Register condition readers and writers...

```
// By requesting a read handle, a Gaudi component notifies the condition management infrastructure that
// its execution should be scheduled after the corresponding condition is produced.
template< typename T >
ConditionReadHandle<T> getReadAccess( const detail::ConditionUserID & client,
                                      const detail::ConditionID     & targetID,
                                      const ConditionKind             targetKind );

// By requesting a write handle, a Gaudi component identifies itself as the producer of a condition.
template< typename T >
ConditionWriteHandle<T> getWriteAccess( const detail::ConditionUserID & client,
                                        const detail::ConditionID     & targetID,
                                        const ConditionKind             targetKind );
```

- ...with configuration error checking!

```
// Condition type checking errors will be reported as follows
class InconsistentConditionType : public ConditionPrototypeException { };

// Attempts to register two writers for a single condition will be reported as follows
class WriterAlreadyRegistered : public ConditionPrototypeException { };
```

Condition data access

- Conditions are accessed via thread- and type-safe handles

```
template< typename T >
class ConditionReadHandle
{
public:

   // Condition handles are movable, but not copyable
   ConditionReadHandle( const ConditionReadHandle & ) = delete;
   ConditionReadHandle( ConditionReadHandle && other ) = default;
   ConditionReadHandle & operator=( const ConditionReadHandle & ) = delete;
   ConditionReadHandle & operator=( ConditionReadHandle && other ) = default;
```

- WriteHandles can put condition data into storage

```
   // Tell whether the condition is set or needs to be (re)generated
   bool exists( const ConditionSlotID slot ) const;

   // Set the value of the condition
   void put( const ConditionSlotID     slot,
             ConditionData<T> && value ) const;
   class ConditionAlreadySet : public ConditionPrototypeException { };
```

- ReadHandles access this data by const reference

```
   // Provide read-only access to the condition data
   const ConditionData<T> & get( const ConditionSlotID slot ) const;
```

A storage allocation caveat

- Imagine that a new event comes, and we cannot give it a condition storage slot yet
  - Classic scenario: all condition slots are used up
  - Other scenarios exist if condition slot initialization is asynchronous (e.g. done by CondAlgs)
- What should we do?
  - Return an error code? (Forces client to poll, less efficient)
  - Block the event scheduling thread? (May deadlock)
  - Best strategy depends on scheduler implementation!

Taking a third option

- The client knows best what to do, so let it choose
- C++ Concurrency TS futures give it many possibilities
  - Poll slot allocation status: is_ready()
  - Wait for slot allocation to complete: get()
  - Execute code once the slot is ready: then()
  - No extra threads needed, mutexes are optional
- Sadly not a priority of the C++ standard committee...
  - ...but an implementation is available in Boost::Thread

Storage management interface

- Condition storage is requested asynchronously...

```
cpp_next::future<ConditionSlotID> allocateSlot( const detail::TimePoint & eventTimestamp );
```

- ...and disposed of after use[2]

```
void liberateSlot( const ConditionSlotID slot );
```

- Behind the scenes, clever machinery can be used to avoid storage and effort duplication:
  - Events reuse entire condition slots when they fit
  - New slots reuse matching data from other slots
  - Condition data is accessed by shared_ptr

[2] This step could be automated by making ConditionSlotID an RAII wrapper

Transient storage status

- Transient storage prototype is now implemented
- Proves (efficient) feasibility of abstract condition storage
- More work will obviously be needed for Gaudi integration
  - Interface consistency with event data storage
  - Format of the dataflow report to Gaudi scheduler
  - Mechanism to signal cached/missing conditions
  - Component/condition identifiers, time representation...
- All the data is there, it just needs to be formatted right

Outlook

- So far, shown that we can store conditions in RAM and tell Gaudi how to schedule producers and consumers
- Also need example of producer/consumer components
  - Show how interface makes it easy to manipulate conditions
  - If we stumble upon anything complicated, simplify it
- Provide a basic outline of Gaudi scheduler integration
  - Helps define our requirements for said integration
- Prototype should be complete in time for mid-January
- Gaudi & experiment integration to come after that

Thanks for your attention

Prototype code @ https://gitlab.cern.ch/hgraslan/conditions-prototype