Channel filtering in LArSoft

Gianluca Petrillo, Saba Sehrish, Erica Snider

University of Rochester/Fermilab

LArSoft Architecture Review Meeting, June 24th , 2015

Channel filtering

The channel filter provides information about goodness of each TPC readout channel.

The information hosted so far includes:

bad channel is dead of irremediably bad

noisy channel is noisy

non-physical channel has no actual data
(added by MicroBooNE to describe “wireless” channels)

Current implementation

Well...

```
class filter::ChannelFilter {
    public:

  enum ChannelStatus { GOOD        = 0,
                       NOISY       = 1,
                       DEAD        = 2,
                       NOTPHYSICAL = 3
                     };

  ChannelFilter();

  bool BadChannel(uint32_t channel);
  bool NoisyChannel(uint32_t channel);
  ChannelStatus GetChannelStatus(uint32_t channel) const;
  std::set<uint32_t> SetOfBadChannels()   const;
  std::set<uint32_t> SetOfNoisyChannels() const;
}; //class ChannelFilter
```

Listing 1: Current `ChannelFilter` class

The current implementation is a joke I will not detail here.
Just note the arguments of the constructor...

Current uses

Very simple to use: instantiate, then query.

```
filter::ChannelFilter chanFilt;

// ...

for(auto & itr : planeIDToHits){
  allhits.resize(itr.second.size());
  allhits.swap(itr.second);

  fDBScan.InitScan(allhits, chanFilt.SetOfBadChannels());

  // ...
}
```

Listing 2: Excerpts from `DBcluster` module

Currently used in:

calibration `recob::Wire` should not be created for bad channels

reconstruction algorithms for track-like clusters check if a gap was due to a bad channel (usually, in the wrong way)

event display

The issue

In short:

- experiment-dependent behaviour is hard-coded
- the channel maps are also hard-coded

(sorry about that)

Requirements of the new channel filtering:

- expose a single interface to the user code
- allow independent implementations by the experiments
- support as data sources: FHiCL configuration, text files, databases...
- as easy as the current one to use in the code

Proposed solution

LArSoft proposal:

A common service interface hiding experiment specific implementation of channel quality queries.

In particular, the database-based service model (used, for example, to retrieve pedestal information) seems suitable for our goals.
LArSoft would implement:

1. abstract service provider interface (framework-independent)
2. abstract `art` service interface
3. default implementation of both for FHiCL-driven data

Proposed service provider interface

The service provider might follow this interface:

```
class filter::ChannelQuality {
    public:
  using ChannelSet_t = std::set<raw::ChannelID_t>;

  virtual ~ChannelQuality() = 0;

  virtual bool isPresent(raw::ChannelID_t channel) const = 0;
  virtual bool isGood   (raw::ChannelID_t channel) const = 0;
  virtual bool isBad    (raw::ChannelID_t channel) const = 0;
  virtual bool isNoisy  (raw::ChannelID_t channel) const = 0;

  virtual ChannelSet_t GoodChannels() const = 0;
  virtual ChannelSet_t BadChannels() const = 0;
  virtual ChannelSet_t NoisyChannels() const = 0;

  virtual bool Update(lariov::IOVTimeStamp const& ts) = 0;

}; // class filter::ChannelQuality
```

Listing 3: A stub of `ChannelQuality` interface

The `art` service would just return the service provider.

Models

FHiCL file model (implemented in LArSoft)

- the service configuration contains all the channel information
- information is moved by the constructor into internal structures
- queries are replied with that local data
- the content is never updated

Database model (implemented by the experiments)

- the service configuration contains database connection directions
- the service provider deals with the specific database structure
- the service provider turns queries to the database;
  caching is an implementation detail
- the `art` service triggers content update on every new event

Additional features

Optional features that can be implemented on demand:

1. legacy `ChannelFilter` class reproducing the old behaviour (it will still require the new service to be configured)
2. iterators to channel IDs with specific quality (e.g. good)
3. iterators to channel IDs with custom quality
4. iterators to `raw::RawDigit` (as for channel IDs)
5. interface extension to get channel quality as map of bits
6. ...

Backup

Additional bit-based interface

```
class filter::ChannelQuality {
    public:
  // the stuff above, plus:

  constexpr size_t NBits = 32;
  using ChannelBits_t = std::bitset<NBits>;

  typedef enum {
    cqNonPhysical,                ///< no wire connected to the channel
    cqDead,                       ///< dead channel
    cqNoisy,                      ///< noisy channel

    cqCustomQualityStart = 16U ///< from this on: experiment-specific
  } ChannelQuality_t;

  virtual ChannelBits_t ChannelStatus
    (raw::ChannelID_t channel) const = 0;
  virtual bool isChannel
    (raw::ChannelID_t channel, ChannelBits_t mask) const;

}; // class filter::ChannelQuality
```

Listing 4: Additional (optional) interface for bit-based quality