ROOT TTree objects passing between Narval Actors

Gennaro Tortone
INFN Napoli

Introduction

Narval framework is a powerful tool to design a Data Acquisition Software as a composition of modules (actors) where the transport layer of data is achieved using common network protocols or other techniques (e.g. shared memory, custom bus, …). Data format is a key feature in this scenario and each actor must be able to manage event structures in proper way in order to carry out processing or storage.

This tutorial will show how to transmit and receive ROOT TTree data objects between Narval C++ actors. The Narval topology of the example is a simple Producer → Filter → Consumer pattern that is very common in DAQ design. GRU will be also used in Filter actor to visually show transmitted data.

Producer code

The "core" of each Narval C++ actor is the *process_block* method that has the following signature:

```
void pbuffer::process_block (void *output_buffer,
                             unsigned int size_of_output_buffer,
                             unsigned int *used_size_of_output_buffer,
                             unsigned int *error_code)
```

output_buffer is a pointer to pre-allocated memory area - size definition of *output_buffer* area is contained in topology definition file and available in C++ code throug *size_of_output_buffer* parameter.

The Producer code fills a TTree branch with random values then serialize the tree for network transmission.

```
   {
 1   TTree *t1 = new TTree();
 2   TBufferFile *tbuf = new TBufferFile(TBuffer::kWrite,32000);
 3   Double_t random;
     char *source;


     cout << "PROD::process_block : libproduce.cc - start\n";

 4   t1->Branch("random",&random,"random/D");
 5   for (Int_t i=0;i<100;i++) {
 6      random = gRandom->Rndm();
 7      t1->Fill();
     }

 8   t1->Streamer(*tbuf);

 9   source = tbuf->Buffer();
10   *used_size_of_output_buffer = tbuf->BufferSize();

11   memcpy((char *) output_buffer, source, tbuf->BufferSize());

     cout << "PROD::process_block : libproduce.cc - done\n";

     some_algo_dependent_data++;
12   *error_code = 0;

13   tbuf->Delete();
14   t1->Delete();
   }
```

At lines 1 and 2 a new TTree and TBufferFile are initialized. *tbuf* object is needed for data serialization and has a size of 32000 bytes.

Starting from line 4 the TTree object is populated with a branch called *random* and with 100 random values.

Line 4 invokes *Streamer* method of TTree object and, as result, it is serialized in *tbuf*.

At the end (lines from 9 to 11) *output_buffer* is updated with serialized data (array of *char)* with *memcpy* function and lines 13 - 14 provide deallocation of temporary objects.

Filter code

The *process_block* of Narval C++ Filter has the following signature:

```
void fbuffer::process_block (void *input_buffer,
                             unsigned int size_of_input_buffer,
                             void *output_buffer,
                             unsigned int size_of_output_buffer,
                             unsigned int *used_size_of_output_buffer,
                             unsigned int *error_code)
```

input_buffer and *output_buffer* are pointers to pre-allocated memory area - size definition of *input_buffer* and *output_buffer* area are contained in topology definition file and available in C++ code throug *size_of_input_buffer* and *size_of_output_buffer* parameters.

The Filter receives TTree serialized objects in its *input_buffer* parameter. As next step object deserialization is performed and all TTree random values are used to update a TH1I ROOT histogram available on network socket thanks to GRU.

GRU socket histogram is initialized in main object constructor:

```
fbuffer::fbuffer ()
{
  Int_t port = 9090;

  MyDB = new GSpectra();
  serv = new GNetServerRoot(MyDB);

  serv->SetPort(port);
  serv->StartGNetServer();

  spe = new TH1I ("MySpectra","MySpectra",1000,0,1000);
  MyDB->AddSpectrum(spe,"SpectraFamily");
}
```

The filtering routine is the *process_block* method:

```
   {
 1   TTree *t2 = new TTree();
 2   TBranch *bpoint;
 3   TBufferFile *tbuf2 = new TBufferFile(TBuffer::kWrite, 32000);
     Double_t value;

     cout << "FILT::process_block : libfilter.cc - start\n";

 4   tbuf2->Reset();
 5   tbuf2->SetWriteMode();
 6      tbuf2->SetBuffer((char *) input_buffer, size_of_input_buffer);

 7   tbuf2->SetReadMode();
 8   t2->Streamer(*tbuf2);

 9   bpoint = t2->GetBranch("random");
10   bpoint->SetAddress(&value);

11   Int_t nentries = (Int_t) t2->GetEntries();
12   for (Int_t i=0;i<nentries;i++) {

13      t2->GetEntry(i);
14      bpoint->GetEntry(i);
15      spe->Fill(value*1000);
     }

16   memcpy((char *) output_buffer, source, tbuf->BufferSize());
17   *used_size_of_output_buffer = size_of_input_buffer;
18   *error_code = 0;

19   tbuf2->Delete();
20   t2->Delete();

     cout << "FILT::process_block : libfilter.cc - done\n";
   }
```

At lines 1 – 3 a new TTree, TBufferFile and TBranch are initialized. *tbuf* object is needed for data serialization and has a size of 32000 bytes, *bpoint* is an object needed to point to TTree branch data.

On lines 4 – 6 the *tbuf* object is initialized with *input_buffer* (serialized data from Producer) and on line 8 the local TTree object is initialized with serialized data.

Lines from 9 to 15 browse TTree object and fill ROOT TH1I *spe* histogram.

At the end (lines from 16 to 18) *output_buffer* is updated with original serialized data (array of *char)* with *memcpy* function and lines 19 - 20 provide deallocation of temporary objects.

Consumer actor is not shown but it is possible to transfer TTree object following the above method.

With VIGRU it is possible to visualize in real-time histogram content and apply fitting functions.

Complete source code for Producer, Filter and Consumer of this tutorial is available on Narval wiki. Topology file and start/stop scripts are also included.

References

Narval homepage
http://narval.in2p3.fr

Narval wiki
https://forge.in2p3.fr/projects/narval/wiki

GRU
http://wiki.ganil.fr/gap/wiki/Documentation/Gru/Gru

VIGRU
http://wiki.ganil.fr/gap/wiki/Documentation/Gru/Vigru

ROOT
http://root.cern.ch