ALFA/FairMQ

Shared Memory Transport: Status and Plans

... and other updates

Alexey Rybalchenko
(GSI Darmstadt, FairRoot group)

ALICE Offline Week
CERN, July 21, 2017

FairMQ

Core Concepts

Message passing framework with abstract transport interface (switch to future technologies possible).

General concepts:

- Hide all transport-specific details from the user.
- Clean, maintainable and extendable interface to different data transports (ZMQ, nanomsg, shared memory ...).
- Allow combinations of different transport in one device in a transparent way.
- Any device/channel can switch transport only via configuration, without modifying device/user code -> same API.

Ownership concepts:

- Message owns data.
- Sender device (user code) passes ownership of data to framework with send call.
- Framework transfers to next device, passes ownership to receiver.
- No sharing of ownership between different devices.

Shared Memory

Previous Presentation

Offline Week 2017-03-31 https://indico.cern.ch/event/624025/

Shared Memory

ongoing work

- ~~Improving shmem transport stability (in case of device crashes, restarts), watchdog via DDS task triggers.~~ **done!**
- ~~Extending the shmem transport configuration (segment size, …).~~ **done!**
- Ensuring smooth combination with future RDMA transport (allow RDMA to read/write directly from/to shared memory area). **ongoing**
- ~~Possibility to allocate a memory region, and transfer parts of it.~~ **done!**

Alexey Rybalchenko | ALICE Offline Week | 31.03.2017 6/11

Shared Memory

Region

To allow usage in more complex cases (e.g. detector readout), FairMQ allows to request memory regions from transport.

The user is responsible for management and ownership of data within this region.

Messages can the be created by taking a reference to the region and a pointer+size to the desired part of the region.

Shared Memory

Region example

Without region:

```
FairMQMessagePtr msg(NewMessageFor("channelA", 1000));

// access/fill the message via msg->GetData(), msg->GetSize()
// ...

// send the message out, giving ownership to framework/receiver
Send(msg, "channelA");

// the msg buffers are destroyed when no longer needed
// (in case of shared memory - when the last user is done with it)
```

With region:

```
FairMQRegionPtr region(NewRegionFor("channelA", 10000000));

// fill the region and prepare data in it for message(s)
// ...

// create and send one or more messages with the data from the region
FairMQMessagePtr msg(NewMessage(region, ptr, size));
Send(msg, "channelA");

// framework does not deallocate custom msg buffers within the region
// only the entire region (when the 'region' object is destroyed).
```

Detailed example https://github.com/rbx/FairRoot/tree/region/examples/advanced/Region

Shared Memory

cleanup via watchdog (1/2)

The shared memory watchdog tool is monitoring shared memory use and automatically cleans up shared memory in case of device crashes.

With default arguments the watchdog will run indefinitely with no output and clean up shared memory segment if it is open and no heartbeats from shared memory users arrive within a timeout period.

The watchdog cleans up the main shared memory segment, as well as all memory regions (described on the previous slides).

Shared Memory

cleanup via watchdog (2/2)

The watchdog can be further customized with following parameters:

`--segment-name <arg>` : customize the name of the shared memory segment (default is "fairmq_shmem_main").
`--cleanup` : start monitor, perform cleanup of the memory and quit.
`--self-destruct` : run until the memory segment is closed (either naturally via cleanup performed by devices or in case of a crash (no heartbeats within timeout)).
`--interactive` : run interactively, giving segment details and user input for various shmem operations.
`--timeout <arg>` : specify the timeout for the heartbeats from shmem transports in milliseconds (default 5000ms).

Without the `--self-destruct` option, the monitor will run continuously, monitoring (and cleaning up if needed) consecutive topologies.

Now the watchdog can be started either manually or as a DDS trigger (to be run in case of device crashes). Starting it automatically with each FairMQ shmem usage (but only once per node) is under development.

FairMQ

Device introspection helpers

- **Dump of all program options in machine readable format**

  `$ mydevice --print-options`

  puts out:

  `<option>:<value>:<type>:<description>`

  for all command line options (including custom ones).

- **Dump of registered device channels (physical channel names).**

  `$ mydevice --print-channels //` **prints channels that have been registered via:**

```
void MyDevice::RegisterChannelEndpoints(){
  RegisterChannelEndpoint("channelA", 1, 10000); // channel name, min/max sub-channels
  RegisterChannelEndpoint("channelB", 1, 1); // channel name, min/max sub-channels
}
```

  in this form:

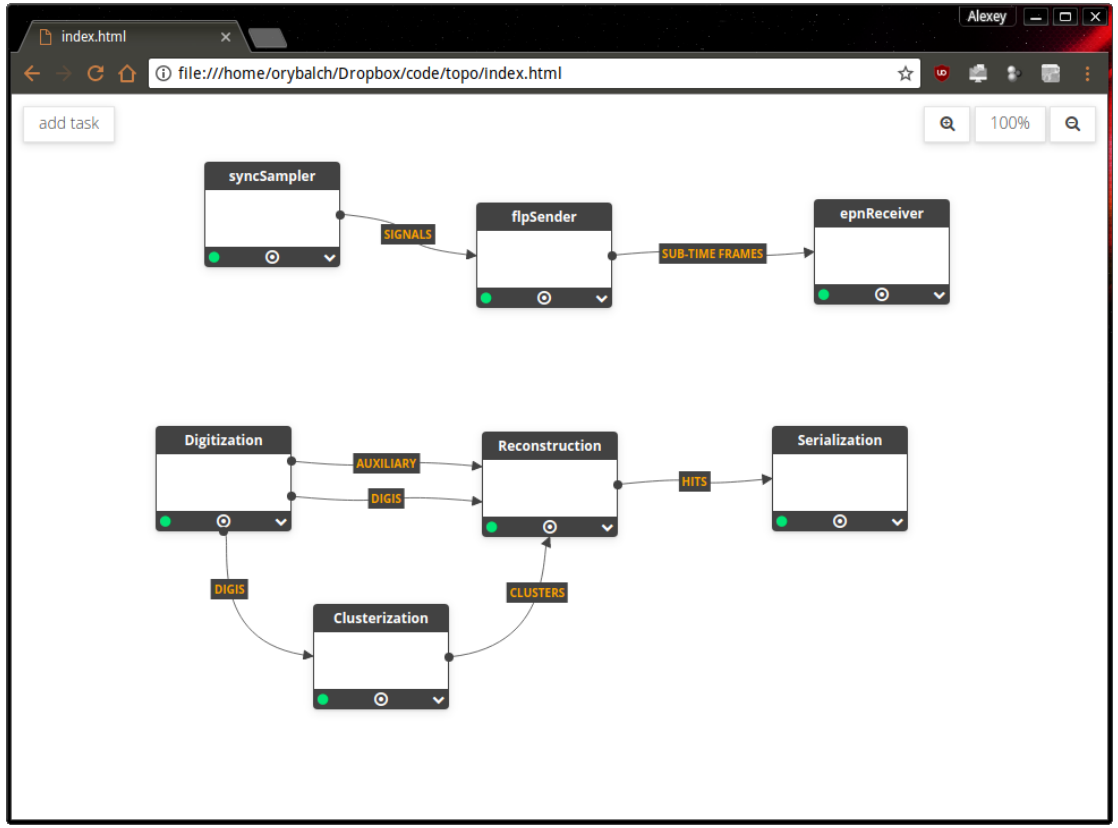
```
channelA:1:10000
channelB:1:1
```

In some cases final physical channels names can be purely dynamical (derived from configuration), therefore cannot be registered in advance in code.

FairMQ

Topology Viewer/Editor (**WORK IN PROGRESS, EARLY DESIGN PHASE**)

compute(data flow graph, resources list) → interactive topology → FairMQ config (DDS/JSON/scripts)



The first view allows to interactively create (or import) a data flow graph, representing processing tasks, and (optionally) their resource requirements.

Data flow graph can also be imported (e.g. result of `runDataProcessing` and WorkflowSpec from Data Processing Layer in O2 Framework?).

Types of resources: hostnames, special hardware (GPU), etc…

The final product of the tool is the configuration for the devices, either as JSON+start script, or as DDS topology.

The core functions (e.g. compute) should be available outside GUI for automation.

FairMQ

miscellaneous

- **Store/provide custom device version. E.g:**

```
MyDevice::MyDevice()
  : FairMQDevice({1.0.0})) { // ... }
```

Calling MyDevice executable with `--version` will print FairMQ version, custom device version (here 1.0.0) and used plugins versions.

- **Callbacks on state changes (mostly for use by plugins).**

More on plugins in next talk by Dennis Klein!

- **Documentation and examples cleanup.**

- **Many small bug fixes and improvements: config thread safety, exception handling, etc.**