

ALFA/FairMQ

Status

Alexey Rybalchenko
(GSI Darmstadt, FairRoot group)

ALICE Offline Week
CERN, March 16, 2018

FairMQ

Recent developments

- **Completion callbacks for the UnmanagedRegion.**
- **Introduction of sessions.**
- **Polishing the shared memory monitor.**
- **Message creation for data of unknown size.**
- **Unification of FairRoot logging facilities.**

Shared Memory

UnmanagedRegion (**reminder**)

Goal: give user **full freedom with the memory layout** of his data (which can be required by certain hardware). But leave the allocation of the region to the transport, to ensure most efficient transfer.

In the contrary to normal messages, region messages are not cleaned up automatically, only the entire region is cleaned up once it is destructed. **Cleanup of the structures within region is left to the user.**

→ User can request memory region from a transport. She can then place her own data structures within the region and create messages out of subsets of the region.

Receiving devices see messages originating from a region just as a normal message - the region details are hidden behind the scenes. Only the region creator cares about the region management.

More info from Offline Week 2017.07.21 talk

<https://indico.cern.ch/event/649759/>

UnmanagedRegion

Completion callbacks

```
fRegion = FairMQUnmanagedRegionPtr(NewUnmanagedRegionFor(
    "chanA", // channel
    0,      // sub-channel
    100000000, // region size
    [](void* ptr, size_t size, void* hint) { // callback
        // cleanup/reuse part of the region
    }
));

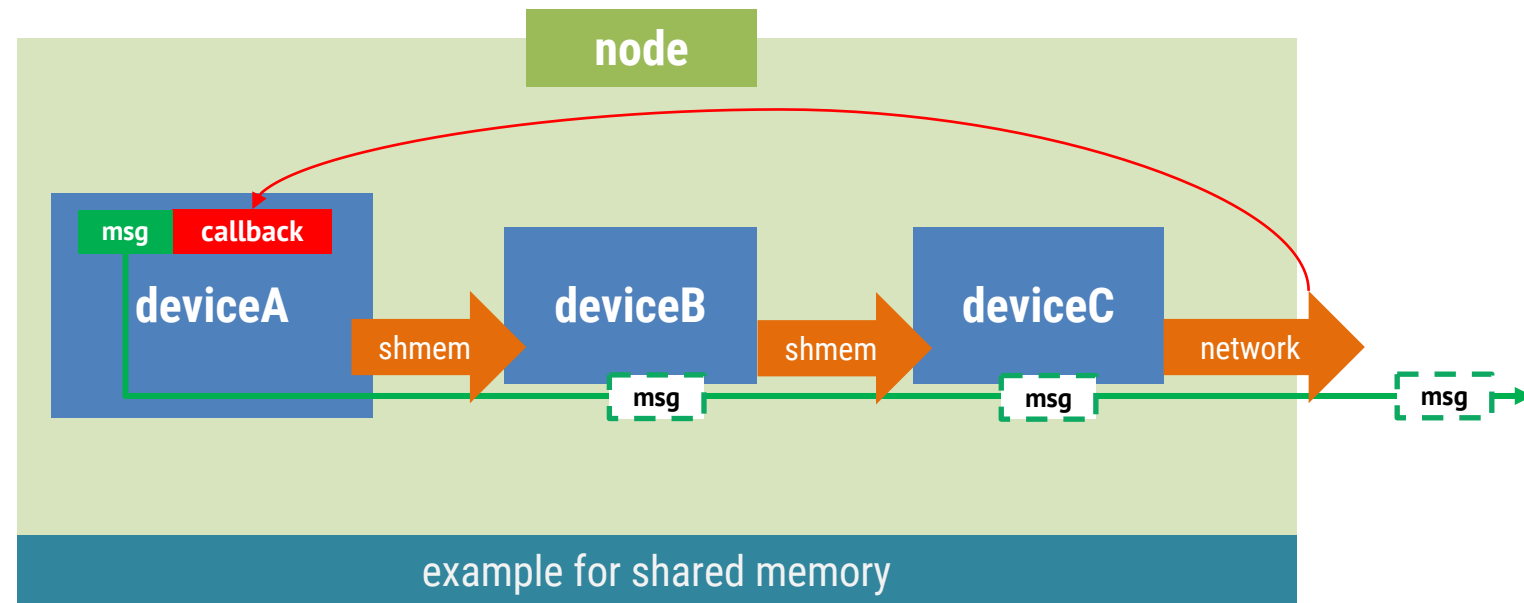
// ...

// create and send messages with the data from the region
FairMQMessagePtr msg(NewMessageFor(
    "chanA", // channel
    0,      // sub-channel
    fRegion, // msg from this region
    ptr,    // buffer ptr (within region)
    size,   // buffer size
    hint    // optional (given in the callback)
));

Send(msg, "chanA");
```

Region user should know when a buffer from a region is no longer needed by the transport (or, in case of shared memory, no longer needed by any of other devices).

For this we now provide a callback mechanism, that calls your handling code once transport is done with the message.



Request from WP6

Detailed example

<https://github.com/FairRootGroup/FairRoot/tree/dev/examples/advanced/Region>

Sessions

Running multiple independent topologies

It should be possible to run multiple independent FairMQ topologies on a same node without interference.

With the introduction of the shared memory transport come resources that are shared between devices. Such resources need to be kept local to a single topology, to avoid interference with other topologies of this or other users.

To enable this isolation, each topology must have a unique topology session id. It is provided to each device via `--session <id>` cmd argument. By default, the session id is simply „default“.

Names of the shared memory resources (and potentially other shared parameters in the future) derive from this session id.

Shared Memory Monitor

Tool for cleanup and monitoring

The shared memory monitor tool is now started with each shared memory transport in the cleanup mode (clean memory after use, if one or more devices crashed).

The monitor is automatically started as a daemon with shared memory transport (once per session id) and is shutdown automatically when corresponding shared memory is either properly closed or cleaned up.

The tool can still be started interactively to provide some monitoring information.

Message Creation

of unknown size

The primary way to create FairMQ messages is by providing desired size for the transport to allocate a buffer and then filling it with your data.

However, in some cases, the size of the data is not known in advance. Primary example for such a case is when compression is done on the data. Or as output of certain algorithms, e.g. TPC track finding and fitting.

To efficiently handle such cases (without additional copy) we now support creating a message by providing an upper bound size, filling it with data and subsequently telling us how much of the buffer was actually used. The transport will efficiently apply the used size to avoid further copy (ZeroMQ, shmemp). In case of shared memory, the unused space is immediately returned to the segment.

```
FairMQMessagePtr msg(NewMessageFor("chanA", 0, 1000));  
// ... fill the contents ...  
msg->SetUsedSize(500);  
Send(msg, "chanA");
```

Logging

Single library

Until recently, FairRoot project contained two separate loggers:

FairLogger: used throughout FairRoot in framework code, ROOT macros and all the derived experiments. Custom implementation.

FairMQLogger: trying to avoid expensive dependency on ROOT (that FairLogger has). Uses Boost.Log with some customizations.

Both loggers use macro-style interface `LOG(severity) << content;`.

Unfortunately, some of the logger details conflict with each other. Also: two implementations (bad for maintenance).

Therefore we wanted to have a single logging library and, if necessary, have old classes use it while providing backwards-compatibility or adding ROOT specific features on top.

These are the primary features that we need for the new logging library:

- Thread safety
- Good performance
- File and console output
- Allow to keep backwards-compatibility (at least for FairLogger, that is much more widely used at the moment).
- Support for custom severities (debug, info, etc.) and verbositys (high, low, etc.).

Logging

Boost.Log vs. custom implementation

Initial idea: use Boost.Log implementation from FairMQLogger, extend with missing APIs to cover everything FairLogger needs.

However, after switching everything to boost some performance values degraded:

Primarily coming from compilation overhead (ROOT macros), “hidden” log messages and timestamp calculations.

Furthermore, despite already considerable complexity of the Boost.Log library, several things still needed to be implemented on top:

- custom verbosity.
- custom timestamps (efficient ones).
- custom color formatting.
- FairLogger has a feature to call ROOT stack trace and exit(1) on fatal severity. With boost this would only be possible via a custom sinks, which would prevent us from using the features of sinks that boost does provide.

	old (two loggers)	new boost	new custom
compile time	7.1 min	9.2 min	6.9 min
test suite runtime	1.7 min	2.9 min	1.7 min
logger test runtime*	-	12 (-> 8 -> 1.6)** sec	0.1 sec
bin dir size	292.4 MB	293.8 MB	279.9 MB
lib dir size	146.3 MB	173.2 MB	132.9 MB

(*) – 32mil log calls in a “hidden” severity (e.g. debug during info setting).
(**) – applying some optimizations allowed to improve initial result.
first – replacing Boost.DateTime with custom std::chrono for timestamps.
second – adding a custom severity check before boost is even called.

→ **Final decision: use custom implementation → all requirements fulfilled, simple & fast**

Logging

main APIs

All log calls go through the provided **LOG(severity)** macro (thread-safe). Logging is done to cout, file output and/or custom sinks.

Severity is controlled via: `fair::Logger::SetConsoleSeverity("<severity level>");` // or `SetFileSeverity`

Severities: "nolog", "fatal", "error", "warn", "state", "info", "debug", "debug1", "debug2", "debug3", "debug4", "trace".

Verbosity is controlled via: `fair::Logger::SetVerbosity("<verbosity level>");`

Verbosities: "low", "medium", "high", "veryhigh".

Verbosities translate to:

low: [severity] message

medium: [HH:MM:SS][severity] message

high: [process name][HH:MM:SS:μS][severity] message

veryhigh: [process name][HH:MM:SS:μS][severity][file:line:function] message

For FairMQ devices, all the logging settings can be also changed via:

`--verbosity <level>`

`--severity <level>`

`--color <true/false>`

`--log-to-file <filename_prefix>` // this will turn off the console output

**<< FairLogger::endl;
no longer needed.**

All severity names are now lowercase. Uppercase is still available for backwards-compatibility.

Further docs

<https://github.com/FairRootGroup/FairRoot/blob/dev/fairmq/docs/Logging.md>

Logging

custom sinks

One of the requests for the logging library is to allow addition of custom sinks. This is supported in the following form:

```
Logger::AddCustomSink("MyCustomSink", "trace", [](const string& content, const LogMetaData& metadata) {
    cout << "content: " << content << endl;
    cout << "available metadata: " << endl;
    cout << "std::time_t timestamp: " << metadata.timestamp << endl;
    cout << "std::chrono::microseconds us: " << metadata.us.count() << endl;
    cout << "std::string process_name: " << metadata.process_name << endl;
    cout << "std::string file: " << metadata.file << endl;
    cout << "std::string line: " << metadata.line << endl;
    cout << "std::string func: " << metadata.func << endl;
    cout << "std::string severity_name: " << metadata.severity_name << endl;
    cout << "fair::Severity severity: " << static_cast<int>(metadata.severity) << endl;
});
```

If only output from custom sinks is desirable, console/file sinks must be deactivated by setting their severity to "nolog".

This feature also eliminates the need for any logging plugins – custom sinks provide sufficient flexibility to cover the needs.

Further docs <https://github.com/FairRootGroup/FairRoot/blob/dev/fairmq/docs/Logging.md>

Summary

Completion callbacks for the UnmanagedRegion.

Sessions for multiple independent topologies.

Full automation of the shared memory cleanup.

Message size flexibility via providing used size.

Cleanup the logging facilities.

Thank You!