FairMQ: Scope, messages & multipart API

Alexey Rybalchenko, GSI Darmstadt • 26.02.2016

Scope

Data Model
classes / structures

Digi, Hit, Time Frame, Heartbeat ...

Alice O^2

Data Protocol
message format, (de-)serialization

Raw binary, ROOT TMessage, FlatBuffers, Protobuf, MessagePack, boost ...

Data Transport
transfer, routing & ordering of binary blocks

ZeroMQ, nanomsg, shared memory ...

FairMQ

Messages API

current API

- Incoming messages are accessed with pointer+size.
- Messages are sent/received to/from user defined channels (ex.: “timeframes”, “heartbeats”, etc.)
- Message objects can be reused for next send after receiving (“rebuilt” with new buffer) - potentially more efficient use depending on the used transport.
- Outgoing messages are either pre-allocated and filled by user task/device, **or** taking over control of provided buffer+size (and deallocation via free() or a callback)).

used in: Tutorial3 (binary).
More efficient if a message has internal header or is allocated in shared memory!

used in:
flp2epn example, Tutorial3 (boost, flatbuffers, msgpack, protobuf, TMessage).

sender

```
std::unique_ptr<FairMQMessage> msg(fTransportFactory->CreateMessage(size));
// ...
// fill msg contents (msg->GetData(), msg->GetSize())
// ...
fChannels.at("digis").at(0).Send(msg);
```

allocation + deallocation
message buffer (**send**)

```
// ...
// build the buffer
// ...
std::unique_ptr<FairMQMessage> msg(fTransportFactory->CreateMessage(buf_ptr, buf_size));
fChannels.at("digis").at(0).Send(msg);
```

deallocation
message buffer (**send**)

sender / receiver

```
std::unique_ptr<FairMQMessage> msg(fTransportFactory->CreateMessage());
fChannels.at("digis").at(0).Receive(msg);
// ...
// use incoming msg (msg->GetData(), msg->GetSize())
// ...
msg->Rebuild(newSize);
// ...
// fill msg contents (msg->GetData(), msg->GetSize())
// ...
fChannels.at("hits").at(0).Send(msg);
```

allocation + deallocation
message buffer (**receive**)

allocation + deallocation
message buffer (**send**)

```
std::unique_ptr<FairMQMessage> msg(fTransportFactory->CreateMessage());
fChannels.at("digis").at(0).Receive(msg);
// ...
// use incoming msg (msg->GetData(), msg->GetSize())
// ...
// create outgoing buffer
// ...
msg->Rebuild(buf_ptr, buf_size);
fChannels.at("hits").at(0).Send(msg);
```

allocation + deallocation
message buffer (**receive**)

deallocation
message buffer (**send**)

Messages API

new API

- Incoming messages are accessed with pointer+size.
- Messages are sent/received to/from user defined channels (ex.: “timeframes”, “heartbeats”, etc.)
- Message objects can be reused for next send after receiving (“rebuilt” with new buffer) - potentially more efficient use depending on the used transport.
- Outgoing messages are either pre-allocated and filled by user task/device, **or** taking over control of provided buffer+size (and deallocation via free() or a callback)).

used in: Tutorial3 (binary).
More efficient if a message has internal header or is allocated in shared memory!

used in:
flp2epn example, Tutorial3 (boost, flatbuffers, msgpack, protobuf, TMessage).

sender

```
std::unique_ptr<FairMQMessage> msg(NewMessage(size));
// ...
// fill msg contents (msg->GetData(), msg->GetSize())
// ...
Send(msg, "digis"); // Send(msg, "digis", 0);
```

allocation + deallocation
message buffer (**send**)

```
// ...
// build the buffer
// ...
std::unique_ptr<FairMQMessage> msg(NewMessage(buf_ptr, buf_size)); // optional callback
Send(msg, "digis"); // Send(msg, "digis", 0);
```

deallocation
message buffer (**send**)

sender / receiver

```
std::unique_ptr<FairMQMessage> msg(NewMessage());
Receive(msg, "digis"); // Receive(msg, "digis", 0);
// ...
// use incoming msg (msg->GetData(), msg->GetSize())
// ...
msg->Rebuild(newSize);
// ...
// fill msg contents (msg->GetData(), msg->GetSize())
// ...
Send(msg, "hits"); // Send(msg, "hits", 0);
```

allocation + deallocation
message buffer (**receive**)

allocation + deallocation
message buffer (**send**)

```
std::unique_ptr<FairMQMessage> msg(NewMessage());
Receive(msg, "digis"); // Receive(msg, "digis", 0);
// ...
// use incoming msg (msg->GetData(), msg->GetSize())
// ...
// create outgoing buffer
// ...
msg->Rebuild(buf_ptr, buf_size); // optional callback
Send(msg, "hits"); // Send(msg, "hits", 0);
```

allocation + deallocation
message buffer (**receive**)

deallocation
message buffer (**send**)

Multipart API

- Single part: contiguous data chunk of any size (accessed via pointer+size).
- Receive() returns when **all** parts have arrived.
- New API: single Send/Receive call. Multipart messages are stored internally as a vector of unique pointers to messages (which are transferred transparently).

sender

current API

```
// build the header and body buffer
// ...
std::unique_ptr<FairMQMessage> headMsg(fTransportFactory->CreateMessage(head_buf_ptr, head_buf_size));
std::unique_ptr<FairMQMessage> bodyMsg(fTransportFactory->CreateMessage(body_buf_ptr, body_buf_size));

fChannels.at("digis").at(0).SendPart(headMsg);
fChannels.at("digis").at(0).Send(bodyMsg);
```

new API

```
// build the header and body buffer
// ...
FairMQParts parts;

parts.AddPart(NewMessage(head_buf_ptr, head_buf_size)); // or size only + fill the data
parts.AddPart(NewMessage(body_buf_ptr, body_buf_size)); // or size only + fill the data

Send(parts, "digis"); // Send(parts, "digis", 0);
```

sender / receiver

current API

```
std::unique_ptr<FairMQMessage> headMsg(fTransportFactory->CreateMessage());
std::unique_ptr<FairMQMessage> bodyMsg(fTransportFactory->CreateMessage());

fChannels.at("digis").at(0).Receive(headMsg);
fChannels.at("digis").at(0).Receive(bodyMsg);

// use incoming header and/or body
// ...
// create outgoing header and body buffer
// ...

headerMsg->Rebuild(head_buf_ptr, head_buf_size);
bodyMsg->Rebuild(body_buf_ptr, body_buf_size);

fChannels.at("hits").at(0).SendPart(headMsg);
fChannels.at("hits").at(0).Send(bodyMsg);
```

new API

```
FairMQParts parts;

Receive(parts, "digis"); // Receive(parts, "digis", 0);

// use incoming parts: parts.Size(), parts.At(i)->GetData(), parts.At(i)->GetSize()
// ...
// create outgoing header and body buffer
// ...

parts.At(0)->Rebuild(NewMessage(head_buf_ptr, head_buf_size)); // or size only + fill the data
parts.At(1)->Rebuild(NewMessage(body_buf_ptr, body_buf_size)); // or size only + fill the data

Send(parts, "hits"); // Send(parts, "hits", 0);
```