

multipart messages explained

M.Krzewicki, FIAS

dramatis personæ

- *Frame*: data with its meta-data (header+payload).
- *Message*: a map of frames (e.g. a timeframe)
- *Transport layer*:
 - everything below user code (device):
manages memory, buffers, queues and the actual transport, exposes Send() and Recv() etc.
- *Protocols + data model*:
 - boundary vague, BUT in our case (maybe):
 - *data model*:
 - binary representation of meta data (“headers”).
 - binary data format (“serialisation”).
 - *protocol*:
 - frame layout convention: headers and payloads separate
 - message layout: a map of frames.

the multi-part trouble

- FairMQ API follows ZMQ API (historically, i guess)
- Its multipart semantics are weird:

- first part(s) (nothing is sent):

```
int FairMQChannel::SendPart(const unique_ptr<FairMQMessage>& msg)
```

- final part (all parts are queued for transport):

```
int FairMQChannel::Send(const unique_ptr<FairMQMessage>& msg)
```

- similar on ::Recv, need to recv until end of message signalled.

- NOT a problem, just an observation.

- we live with it (ZMQ semantics) in HLT
 - have it abstracted away.
 - can do a similar thing with FairMQ.

an example: STL

- implementation fully “on top of” FairMQ, thin layer.
- STL, iterators, nothing for us to maintain here, all standard concepts.
- soon on github.
- **this is semi-pseudo-code to illustrate:**

```
typedef std::pair<FairMQMessage*, FairMQMessage*> o2frame;
typedef std::vector<o2frame> o2message;

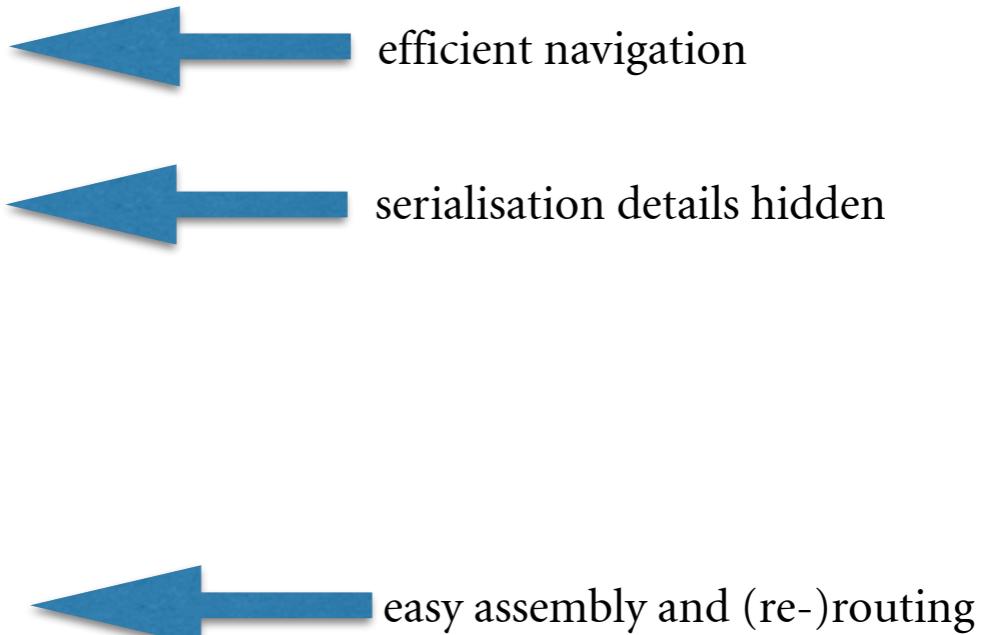
o2message timeframe;
o2message_recv(inputChannel, timeframe);

for (auto input = timeframe.begin(); input!=timeframe.end(); ++input) {
    if (frame_is(input,kDataTypeTPCClusters)) {
        o2frame rootQA;
        frame_init(rootQA, new TH1F(), kDataTypeQA);
        timeframe.push_back(rootQA);

        o2frame output;
        frame_init(output, size, kDataTypeTPCtracks);

        if (DoStuff( input, output )) {
            timeframe.push_back(output); //e.g. add output to timeframe
        } else frame_close(output);
    }
}

o2message_send(outputChannel, timeframe);
```



make multi-part nicer in FairMQ

- the CWG4 way, e.g.:

```
typedef std::pair<FairMQMessage*,FairMQMessage*> frame;
typedef std::vector<frame> message;
int FairMQChannel::Send(message& msg) const;
int FairMQChannel::Recv(message& msg) const;
```

- would make our thin layer thinner.
- assumes the header-payload concept is universal enough.

-
- the POSIX-like way (scatter-gather):

```
int FairMQChannel::Send(vector<FairMQMessage>& msg) const; (or maybe with a vector of "iovec" structs?)
int FairMQChannel::Recv(vector<FairMQMessage>& msg) const;
```

- fits better with POSIX transport semantics (sendv/recv, nanomsg, etc.)
- no assumption of bi-part model.
- if order preserved on recv (as in e.g. zmq): “protocol” easy to reconstruct in layer above.

addendum

- The lower level (FairMQMessage) interface still there
- Each FairMQMessage still nicely independent, access to the contents (buffers, headers) contained in simple API (functions, i'd avoid writing a wrapper class).
 - This API is only a convenience device, BUT:
 - code structure clearly shows intent -> readable.
- The transport layer can do whatever it wants:
 - if we use something nice (zmq,nanomsg,<?>): we utilise the features.
 - if we use raw stream sockets (or sth similarly silly):
 - we just do what would be done anyway in such a case: construct a single buffer, possibly with meta data (scatter/gather list) to convey part structure.
 - use multiple sockets and handle the sync ourselves,.....etc
- what we don't get for free (e.g. proper shared mem) - we implement (or not) above the underlying transport lib, but below FairMQ.