Idea for multi-part service on top of scatter/gather in FairMQ

David Rohr, Gvozden Neskovic, Sandro Wenzel, Mikolaj Krzewicki, Giulio Eulisse, Matthias Richter
drohr@cern.ch, CERN
21.12.2017
Multi-part in FairMQ (derived from ZeroMQ) is deprecated, and anyway not supported by all backends (e.g. NanoMsg).

FairMQ will provide an atomic scatter/gather transport API.
- Initially, this might involve copies and TCP/IP transfer.
- Later, this should use zero-copy RDMA if possible (basically using whatever scatter/gather the underlying API supports).

At many places in O2, we need to send multiple related messages. Examples are:
- The O2 framework always sends a pair consisting of the O2 data header and the payload.
- To simplify merging, we could send multiple messages for one time frame atomically, maintaining the association.
- It can be used for the in-event parallelization of O2 simulation.
- Attaching additional information like streamers to the QA.

All these cases would be covered by a multi-part API.

If no such means exists on the framework side, users will (or do already) implement their own version of a multi-part API on top of FairMQ.
- It makes sense to have a common service for this.
- For us it does not really matter if it is provided by O2 or FairMQ, but if it is provided upstream, others can profit as well.
- The service we suggest would work for pairs (header / payload), but also for vectors of data (depending on the user).
Multi-part implementations

- Multi-part can be implemented in different ways.
  - Multi-part could be implemented on the transport level (not going to happen, I know).
  - The user could copy all his parts into a single buffer and ship that as a single message (inefficient, involves a copy).
  - We can implement multi-part on top of the current FairMQ transport features with some kind of protocol:
    - Send each part as a single message, and one additional message with meta-information to recover the parts. (Broken since the transfer is not atomic. Need to wait for all parts, and must make sure that all parts are received.)
    - Implement it on top of scatter/gather, which can ship multiple fragments in an atomic way. (best solution!)

- This multi-part service would sit somewhere between FairMQ and O2.
  - It is totally independent from the transport, so let’s call it a multi-part service for now.
  - Basically, we could make it either part of FairMQ or of O2.
  - There is no difference for the user or for ALICE, but if it is part of FairMQ, others can use it.
  - We could also go ahead and start implement it in O2, and FairMQ could adopt it later.
Multi-part on top of scatter/gather

- The FairMQ scatter/gather is an ideal basis for multi-part.
  - One can easily implement multi-part on top of scatter/gather without even touching the transport. (let’s call it a service for now)

The user on the receiving side does not know about the fragments
Multi-part on top of scatter/gather

- The FairMQ scatter/gather is an ideal basis for multi-part.
  - One can easily implement multi-part on top of scatter/gather without even touching the transport. (let’s call it a service for now)

- The multi-part service gets a list of parts.
  - It creates some meta-information and a scatter/gather list, which ships the meta information as first fragment.

- Data is shipped via normal FairMQ scatter/gather.
- Using the meta-information, the service restores the list of parts.
  - The scatter/gather transport is not touched and it is still atomic. The multi-part is fully decoupled and sits on top.
Multi-part on top of scatter/gather

- The FairMQ scatter/gather is an ideal basis for multi-part.
  - One can easily implement multi-part on top of scatter/gather without even touching the transport. (let’s call it a service for now)
  - For shared-memory transport, the parts are only handles to the shared memory buffers

- For shared-memory transport, the parts are only handles to the shared memory buffers
Details

• The scatter / gather aggregates all buffers from the scatter / gather list into a single buffer.
• The meta-data added by the multi-part service enables us to recover the original buffers.
• The meta-data is send as first fragment in the scatter / gather, so we find the meta-data at the beginning of the buffer on the receiver side.
• A simple meta-data format could be:
  • Number of parts in multi-part.
  • Array with size of all multi-part buffers.
  • CRC sum to validate meta-data.
• With the values, we know the sizes and can compute the offsets of all the parts in the receiving buffer.
• With the CRC, we can make sure that what we received in the scatter / gather was indeed a multi-part message.
• An interface will enable the user to navigate inside the multiple parts.
The interface should enable the user to easily send a multi-part message, and navigate through the parts on the receiving side.

- **On the sender side:** Either provide a list, or have an object to add parts one after another.
  - Nothing happens until the final send, the user must make sure the added parts persist until then. Send is atomic.
- **On the receiving side:** Getters for number of parts and the i-th part, as well as an iterator over all parts.
  - The receive is atomic. The transport has nothing to do with this iterator. All parts are in the memory of the receiving side, the iterator is just an ease-of-use for the user.

- **Ideally, we can use the same container for send / receive (simplifies forwarding).**
  - As std::vector, basically a std::vector<FairMQMessage> should be enough.
  - We could even maintain the FairMQParts interface, but take it out of the transport.

- The overall idea is: This is just an interface / service, which enables the user to send multi-part messages using the FairMQ scatter / gather. It is not a really new / different feature, but just an abstraction.
  - The performance will be in every respect identical to scatter / gather.
    - If scatter / gather does RDMA with zero-copy, this multi-part does as well. If scatter / gather needs to copy n times, the multi-part interface does as well.
In this way, the multi-part is decoupled from the transport.

We can still decide on the protocol:

- Do we want some “special channel-type” for multi-part?
- Do we simply intermix multi-part and “normal” messages on one channel?
  - It is up to the user to make sure that when he sends a multi-part, he treats the message as multi-part on the receiving side.
  - We can easily avoid a “normal” message being received as a multi-part message: the service on the receiving side could check for a checksum in the meta information and return a failure in case the scatter-gather transport was not a multi-part.
  - The service can not easily catch when a multi-part message is simple received and interpreted as normal message – which is by the way the user’s fault.
Performance

- **We must ensure that this multi-part service yields good (i.e. zero copy-overhead) performance.**
  - In case of network transfer, assuming the underlying RDMA transfer supports scatter-gather, the data should be shipped in one single transfer.
  - In case of a non-RDMA transfer, it is a bit more complicated. There is basically no way to aggregate the source buffers in a continuous destination buffer without a copy. Thus all non-RDMA transfers will need to copy.
    - However, this is only relevant for node-internal transfers.
    - In this case we use shared memory.
    - The messages (which are copied) only hold a reference to the shared memory segment.
    - The message copy is negligible, the shared memory segment is not copied.

- **This means: in both cases, good performance should be achieved.**
- **Basically, the multi-part service yields the exact same performance as the scatter-gather.**

- **However, this approach will require a copy in case the transfer is neither RDMA nor shared memory.**
  - Is this acceptable?
  - Should we provide a multi-part solution based on “normal” messages in that case?
  - Or would it be better to implement the multi-part service based on “normal” messages only in the first place?