Buffer Allocation for Shared Buffers without Copying

David Rohr, drohr@cern.ch
Frankfurt Institute for Advanced Studies
CERN, 26.2.2016

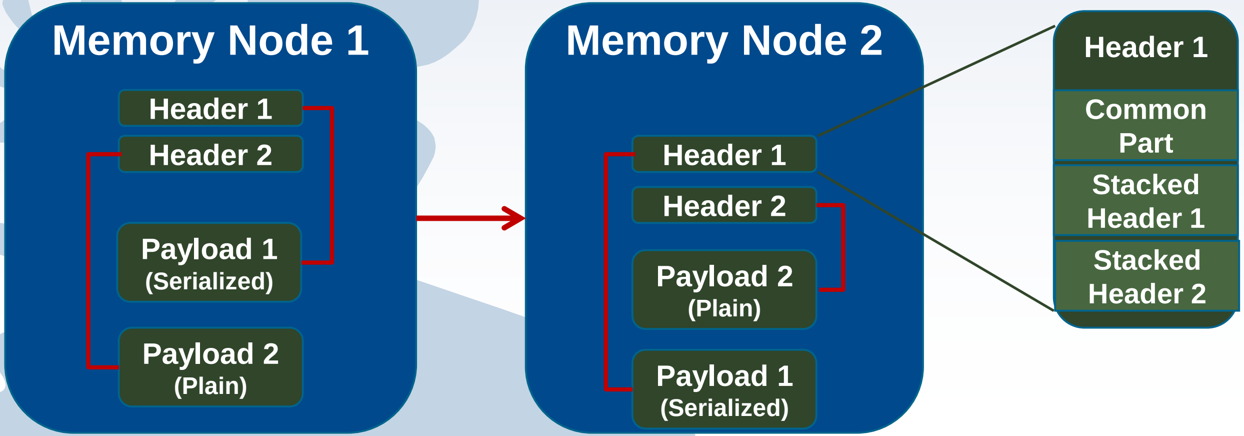
SUMMARY FROM CWG4: DATA HEADERS

Data headers

- **In CWG 4, we foresee multiple input and output data blocks per device.**
  - The payload of each data block is continuous in memory (and has variable size).
  - The payload can be serialized. We support multiple serialization methods. Serialization / Deserialization should be automatized.
    - Plain (non-serialized) data is just one trivial serialization method.
    - Deserialization simply returns the plain pointer to the payload.
  - Each payload data block has a header.
  - The header size is variable.
  - Each header begins with a fixed common header structure, and can contain additional stacked headers with additional header information for this block (e.g. trigger information for data sources of triggered detectors).
    - The stacked headers can be serialized objects as well. We want to support the same mechanisms as for the payload, with the same automation for serialization / deserialization.
  - Access over multiple input buffers via iterators (or similar):
    - The device queries an iterator for all payload blocks with a certain data type and data origin (e.g. TPC Raw).
    - Via the iterator, the device gets meta information from the header, and a pointer to the data buffer (for a plain data buffer), or a pointer to the deserialized object (for a serialized buffer).
    - Analogously for additional stacked headers.

What does this mean for the transport layer

- The transport layer must transfer an amount of pairs: payload buffers and headers.
- The relation of header to buffer must be maintained.
- The size of each payload buffer and of each header is variable.
- The positions of headers and payload in memory is arbitrary.
  - But: each header and each payload on its own is in continuous memory (ongoing discussion in CWG4, but this is the goal):
- The device starts processing only after all input data blocks are available → They must be gathered beforehand.
- The receiving device can select input data depending on the data type / origin in the common part of the header. Data not matching the selection should not be transferred.



- To the transport layer, the header is just one chunk of data.
- The header contains only meta data for the payload
- There is no pointer to the real payload!
  - This pointer is different on every compute node and is maintained by the framework.

Interface Feature Example Pseudocode

- We can use an iterator concept to iterate over input data and stacked headers.
- For instance, we could use STL (see Mikolaj's example later)
- It might be convenient to foresee a method to forward data buffers.
- See below some generic pseudocode:
  - Terminology: FRAME: one data buffer plus its header.
    MESSAGE: a vector of frames

```
iter = Framework.GetInputMessageIterator(DATA_TYPE_RAW, DATA_ORIGIN_TPC);        //PAYLOAD ITERATOR
while (frame = iter->Next())
{
        stackedHeaderIterator = frame->GetStackedHeaderIterator(HEADER_TYPE);  //STACKED HEADER ITERATOR
        if (stackedHeader = stackedHeaderIterator->Next())
        {
                if (…) Framework.ForwardFrame(frame);
        }
        dataPtr = frame->GetPayload(); //Returns a pointer to (possibly) deserialized payload
}

Framework.AppendOutputFrame(…);
```

AVOID COPYING

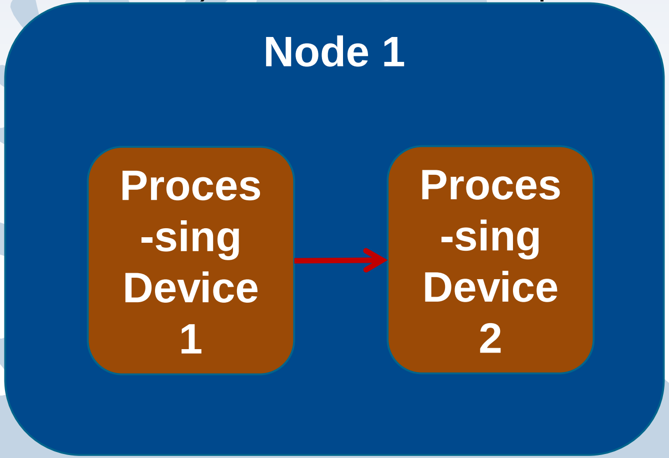
Sending Messages without Copying

- **We want a message-queuing based approach.**
- **The approach should be such, that no copies are needed.**
  - Messages must be passed between the processes on one node.
  - And between processes on different nodes.
    - Our interface should be such, that we can make use of efficient network DMA transfer offered by the interconnect.
  - We must get data from the CRU FPGA PCIe card.

- **How should this work behind the scenes?**

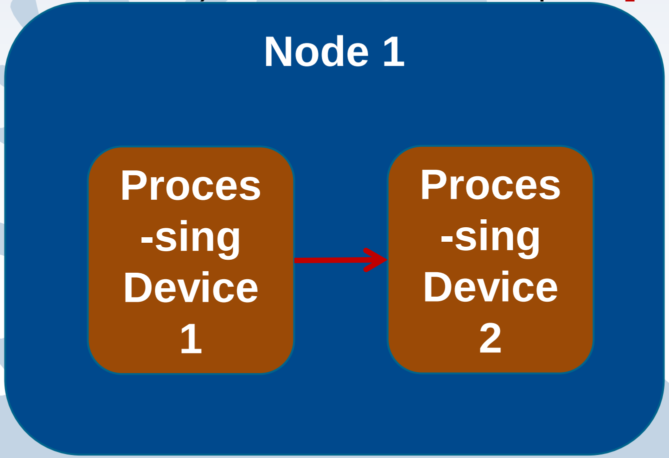
Basic Example 1

- **The most basic example: Two components on the same node:**
  - Probably, this is the most frequent case: Most data transport will be between the individual processing devices processing one timeframe, and these will reside on the same EPN node.
  - Device 1 produces its output data, which is used by device 2.
  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until device 2 is finished.
    - B) Device 2 takes ownership of the buffer and handles the deallocation.
    - C) The data must be copied.



Basic Example 1

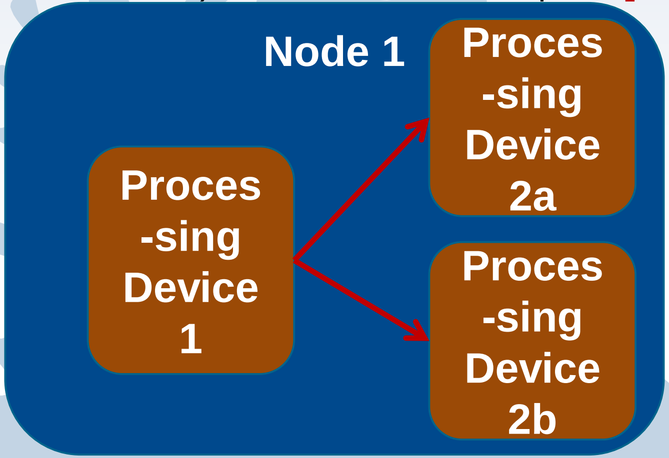
- **The most basic example: Two components on the same node:**
  - Probably, this is the most frequent case: Most data transport will be between the individual processing devices processing one timeframe, and these will reside on the same EPN node.
  - Device 1 produces its output data, which is used by device 2.
  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until device 2 is finished.
    - B) Device 2 takes ownership of the buffer and handles the deallocation.
    - C) The data must be copied. **[We want to avoid this]**



Basic Example 2: Two consumers

- **The most basic example: Two components on the same node:**
  - Probably, this is the most frequent case: Most data transport will be between the individual processing devices processing one timeframe, and these will reside on the same EPN node.
  - Device 1 produces its output data, which is used by device 2.
  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until all consumers are finished.
    - B) The consumers take ownership of the buffer and handles the deallocation.
    - C) The data must be copied. **[We want to avoid this]**

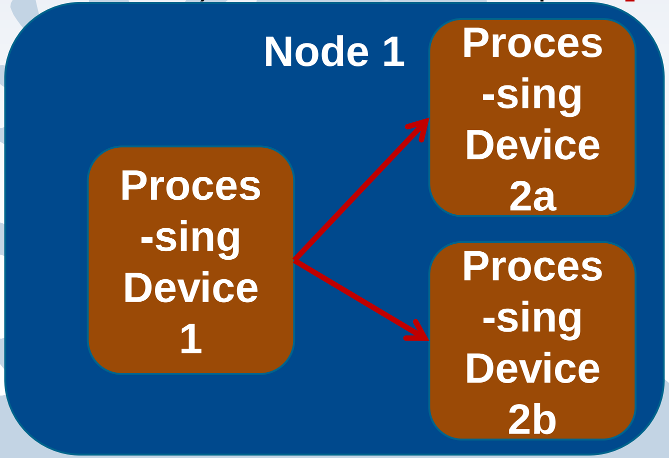
All consumers incur dependencies!



- **There is also the case, that there is more than one consumer.**
  - This makes methods A and B more complicated.
    - A introduces many dependencies.
    - In B, not a single device would take ownership, but multiple devices. Deallocation would happen after all of them have finished.

Basic Example 2: Two consumers

- **The most basic example: Two components on the same node:**
  - Probably, this is the most frequent case: Most data transport will be between the individual processing devices processing one timeframe, and these will reside on the same EPN node.
  - Device 1 produces its output data, which is used by device 2.
  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until all consumers are finished. **[Complicated]**
    - B) The consumers take ownership of the buffer and handles the deallocation. **[→ Shared Buffer]**
    - C) The data must be copied. **[We want to avoid this]**

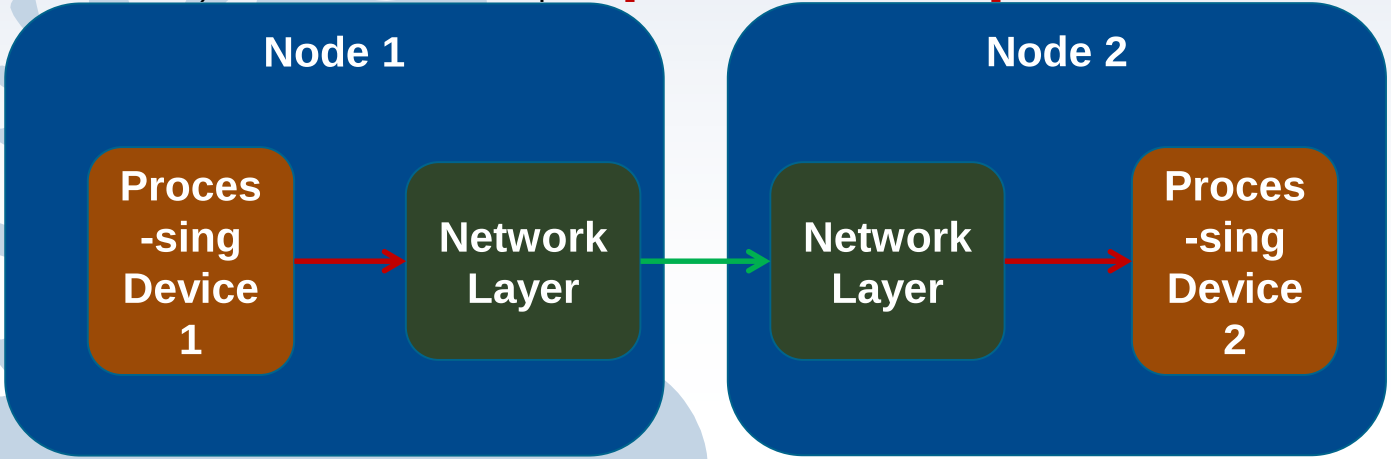


- **There is also the case, that there is more than one consumer.**
  - This makes methods A and B more complicated.
    - A introduces many dependencies.
    - In B, not a single device would take ownership, but multiple devices. Deallocation would happen after all of them have finished.
      - → This is in principle a shared buffer.
- **The situation for a transfer over network is similar.**

Basic Example 3: With network

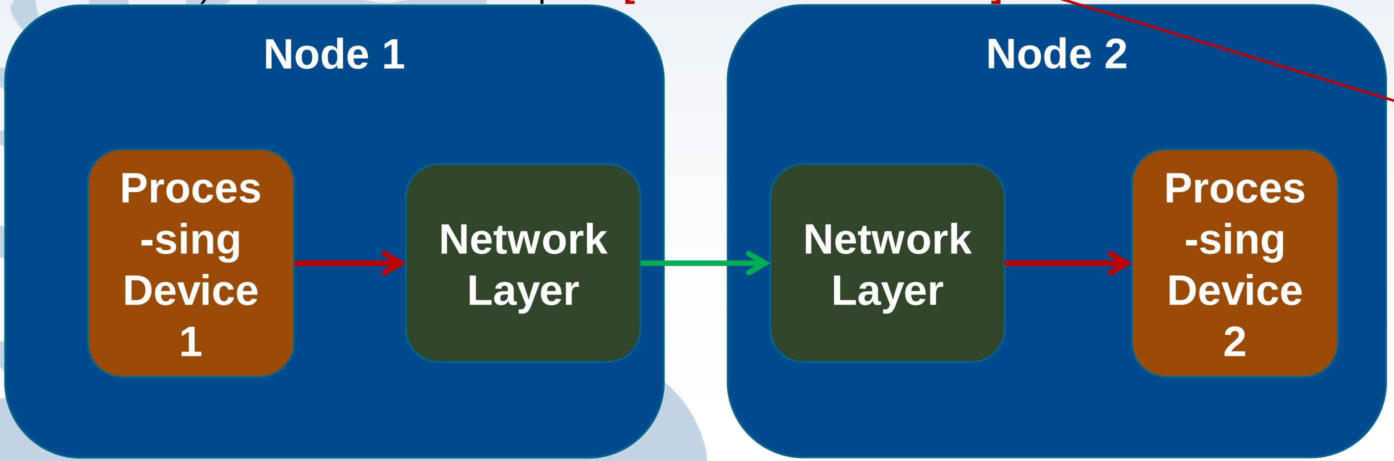
- **The most basic example: Two components on the same node:**
  - **Device 1 must pass its output data to the network transport layer library (in whatever form). → Buffer deallocation must only happen after the network transfer.**

  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until all consumers are finished. **[Complicated]**
    - B) The consumers take ownership of the buffer and handles the deallocation. **[→ Shared Buffer]**
    - C) The data must be copied. **[We want to avoid this]**



Basic Example 3: With network

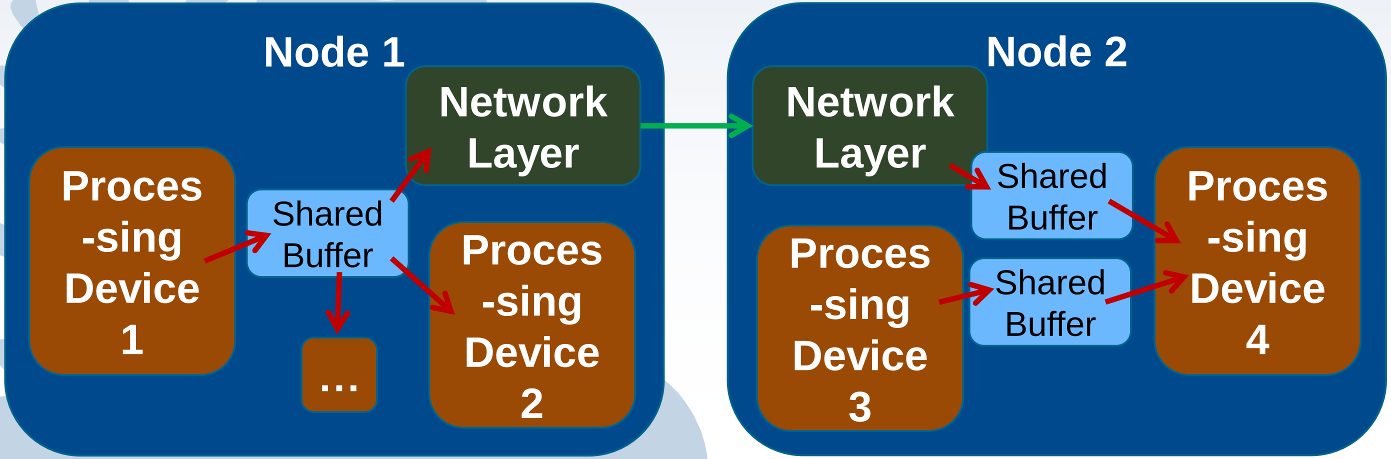
- **The most basic example: Two components on the same node:**
  - **Device 1 must pass its output data to the network transport layer library (in whatever form). → Buffer deallocation must only happen after the network transfer.**
  - **Some networks need registered memory. A device-1-allocated buffer that is freed, must be re-registered!**
  - If device 1 allocates the buffer for its output data, then there are three possibilities:
    - A) Device 1 has to wait with the deallocation, until all consumers are finished. **[Complicated]**
    - B) The consumers take ownership of the buffer and handles the deallocation. **[→ Shared Buffer]**
    - C) The data must be copied. **[We want to avoid this]**



- It might be better to have persistent buffers, to avoid re-registering.
- It might be more efficient, if this is allocated by the transport or framework layer.

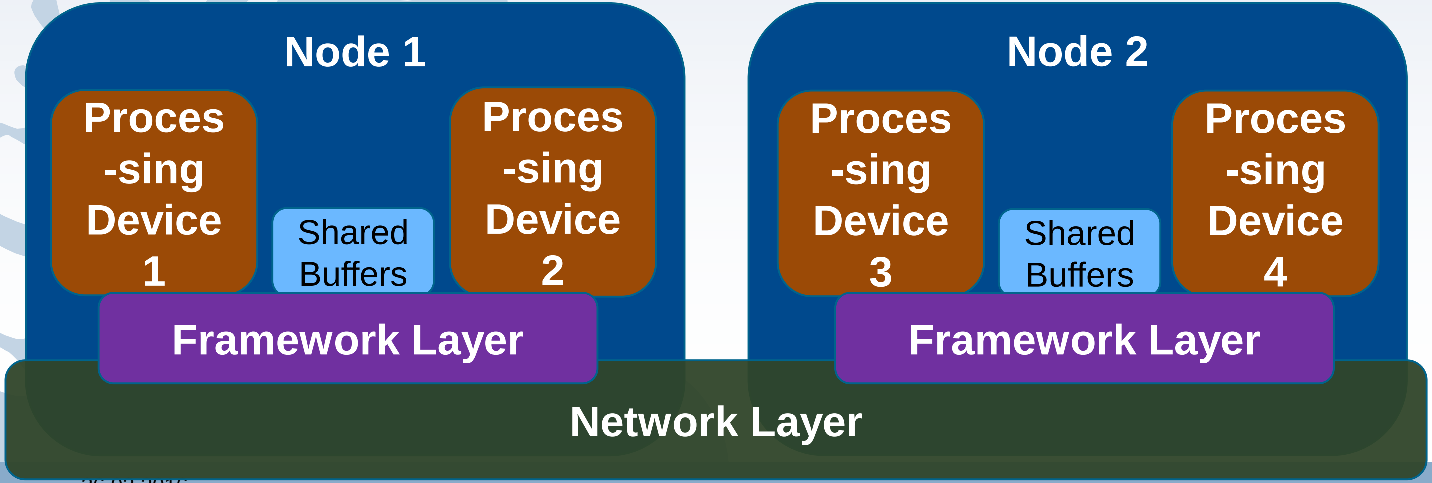
A shared buffer meets all requirements:

- **Buffers should be shared, such that all processes can access them.**
- **Shared buffers can be registered for GPU-accelerators, Network Interface Cards, and Read Out FPGA Cards.**
- **Every combination of subscribers is possible.**
- **Buffers are persistent → No need to reregister buffers.**
  - Such a shared buffer can directly be used for the CRU readout, i.e. as ring-buffer, etc.
  - Buffer management should not be done in the user code in the devices!



A Framework layer

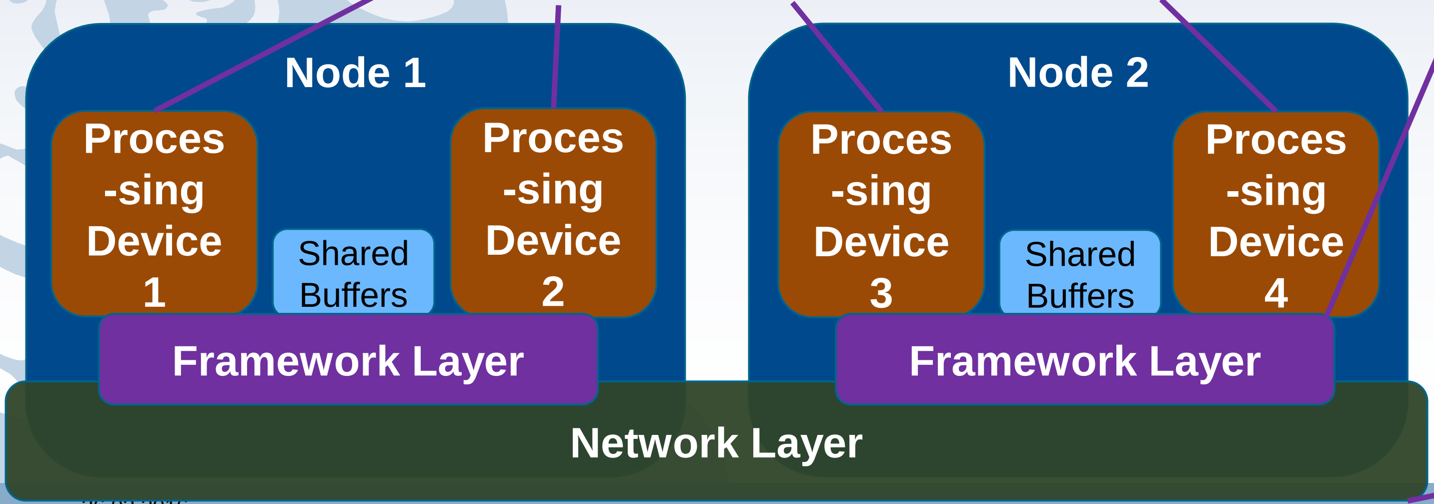
- **We need another layer between the network transport and the processing devices with user code.**
  - “Framework” is a work-name → should be replayed by something reasonable!
- **The framework layer should provide shared buffers to the network, GPU, CRU, and processing devices.**
  - There must be the possibility to use DMA-registered memory for GPUs and the network – if hardware supports it.
    - → The framework layer must provide the necessary interfaces!
    - – These interfaces must provide serialization / deserialization capabilities.



A Framework layer

- **We need another layer between the network transport and the processing devices with user code.**
  - "Framework" is a work-name → should be replayed by something reasonable!
- **The framework layer should provide shared buffers to the network, GPU, CRU, and processing devices.**
  - There must be the possibility to use DMA-registered memory for GPUs and the network – if hardware supports it.
    - → The framework layer must provide the necessary interfaces!
    - – These interfaces must provide serialization / deserialization capabilities.

Essentially FairRoot devices, perhaps with slightly different interface.



This is essentially FairRoot / ALFA

But may be with some additional layers:
- **serialization**
- **iterators**
- **light o2 layer**

Conclusion from the examples

- **We should foresee that all messages can reside in shared memory.**
- **This will allow us zero-copy message sending in any case.**
- **It depends on the underlying transport layer, whether the transfer is really zero-copy in any case, but:**
  - We should define an interface that enables zero-copy.
  - We should not limit ourselves here for the future.
- **The shared memory usage should be abstracted / hidden in the message-queuing API for the user.**

THE INTERFACE

The Interface

- **The framework must provide two interfaces:**
  - Input data for processing devices:
    - Very simple: just a pointer inside the shared buffer and the size of the data.

  - Output data for processing devices:
    - Output data must go to shared buffers, but there will be multiple output methods:

    1. Ideal: The framework provides an output buffer, the device fills the buffer, and tells how many bytes are written.

The Interface

- **The framework must provide two interfaces:**
  - Input data for processing devices:
    - Very simple: just a pointer inside the shared buffer and the size of the data.

  - Output data for processing devices:
    - Output data must go to shared buffers, but there will be multiple output methods:
    1. Ideal: The framework provides an output buffer, the device fills the buffer, and tells how many bytes are written.

- **The above is unfortunately insufficient in some cases:**
  - The device might output an object, that needs to be serialized.
    - Serialization should happen directly into the shared output buffer.
    - This serialization should happen in the framework not in the user code.
    - Serialization must also be taken into account for data input.
  - The device could have the output data in its own memory. (This should be avoided, but there are use cases.)
    - In this case, passing the buffer might be better than copying manually, and it simplifies the code.
  - There might be multiple input / output buffers.

The Interface

- **The framework must provide two interfaces:**
  - Input data for processing devices:
    1. Very simple: just a pointer inside the shared buffer and the size of the data.
    2. Alternatively: the shared buffer contains an object which is deserialized and then passed to the device.
  - Output data for processing devices:
    - Output data must go to shared buffers, but there will be multiple output methods:
    1. Ideal: The framework provides an output buffer, the device fills the buffer, and tells how many bytes are written.
    2. If the device sends an object: It provides object and object type and the framework serializes in a shared buffer.
    3. Should be avoided if possible: The device returns a pointer to a data buffer and the data size.

- **The interface must support multiple input / output buffers!**

The Interface

- **The framework must provide two interfaces:**
  - Input data for processing devices:
    1. Very simple: just a pointer inside the shared buffer and the size of the data.
    2. Alternatively: the shared buffer contains an object which is deserialized and then passed to the device.
  - Output data for processing devices:
    - Output data must go to shared buffers, but there will be multiple output methods:
    1. Ideal: The framework provides an output buffer, the device fills the buffer, and tells how many bytes are written.
    2. If the device sends an object: It provides object and object type and the framework serializes in a shared buffer.
    3. Should be avoided if possible: The device returns a pointer to a data buffer and the data size.

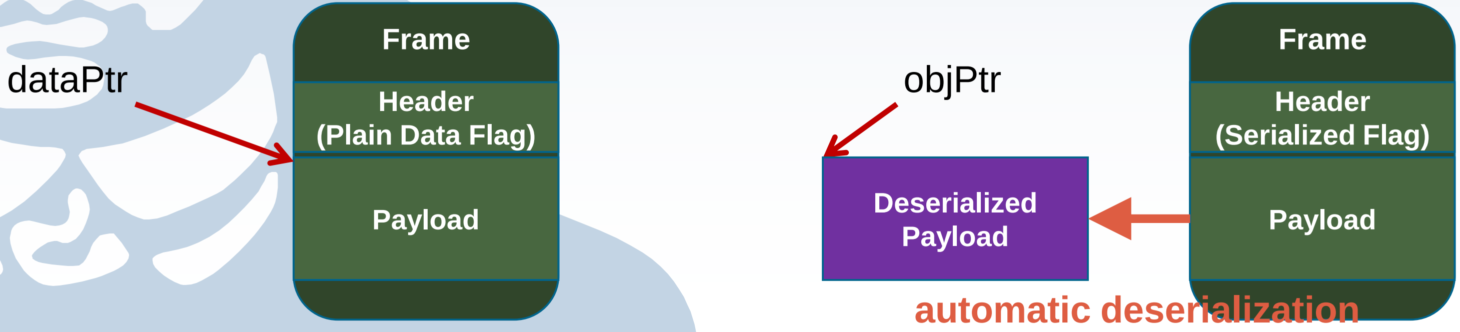
- **The interface must support multiple input / output buffers!**

- **In summary, the message sending will be more efficient, when the memory is allocated by the framework, not by the user.**
  - This works for flat buffers.
  - We try to directly serialize into such shared buffers.
  - Method 3 is only a fallback for compatibility reasons.

Interface for receiving messages

- **We use a simple iterator concept to loop over the messages.**
- **The iterator provides us with a pointer to the payload.**
  - This can be a pointer to a flat data buffer (array / struct).
  - Or this points to an object.
    - This object will be automatically deserialized by the framework.

```
dataPtr = frame->GetPayload();              //For a flat data buffer
dataObj = frame->GetPayload();              //For a serialized object
```



Interface for sending messages of type 1

- **This is the default method to send flat data (struct / array).**
  - The buffer is allocated by the framework, not by the user (can be shared memory).
  - The user fills the buffer.
  - When finished, the user tells the framework how much data was written to the buffer.
  - Then, the message can be sent.

- **It is better to define the message size after the buffer was filled.**
  - The final size might not be known in advance.
  - For the allocation, a maximum size is assumed.

```
frame = Framework.CreateFrame();
outputBuffer = frame.AllocateBuffer(MaxSize);
payloadSize = FillOutput(outputBuffer); //Internal function to fill the output in the buffer, returns size
frame.SetSize(payloadSize);
message.AddFrame(frame);
message.Send();
```

Interface for sending messages of type 2

- **This is the method to send serializable objects.**
  - This should not depend on the serialization method.
  - The framework transparently serializes the object into a shared buffer.
  - Then, the message can be sent.

- **It might be that the serialization library does not allow to serialize into an existing buffer.**
  - Again, this is treated by the framework transparently.
  - The buffer can either be copied to a shared buffer after serialization, or this can be handled otherwise.

```
TH1F histogram;
…

frame = Framework.CreateFrame();
frame.SerializeObject(SERIALIZE_TYPE_TOBJECT, histogram);
message.AddFrame(frame);
message.Send();
```

Interface for sending messages of type 3

- **This method exists only for compatibility.**
  - It should be avoided if possible, as this can incur additional copies.

- **We might have the situation, that the user code wants to ship an existing buffer.**
  - For instance, a foreign serialization library might serialize into its own buffer.

- **It should now be up to the framework to decide whether the data is copied to a shared buffer or sent directly.**
  - Thus, this is identical to messages of type 2, except that there is no serialization.
  - We could define this as SERIALIZE_TYPE_FLAT or SERIALIZE_TYPE_PLAIN.
  - Then we can use the same method.

```
void* myBuffer = malloc(messageSize);
...

frame = Framework.CreateFrame();
frame.SerializeObject(SERIALIZE_TYPE_PLAIN, myBuffer, payloadSize);
message.AddFrame(frame);
message.Send();
```

Interface Example Bottom Line

- **For the user, it does not matter which layer provides which part of the interface.**
  - There will probably anyway be some kind of serialization layer.
  - Whenenver it makes sense (for general purpose), the functionality should go to FairRoot / Alfa.
  - The rest will go to a custom light O2 layer.

- **The important point is:**
  - The message-queuing interface, which we define now / soon, must allow for all the features presented.
  - Otherwise, we might limit ourselves in the future.

ONE IMPLEMENTATION IDEA

One implementation idea

FIAS Frankfurt Institute for Advanced Studies

- **One framework / transport process per compute node.**
  - This process allocated anonymous shared SysV / POSIX buffers.
  - These buffers are registered for the network hardware via the transport layer.
  - The process forks multiple times, once per processing device on that node.
    - This decreases startup time.
    - Common libraries are only loaded once.
  - The (original) framework / transport process communicates with the worker processing processes via shared memory.
  - Data exchange happens in these shared memory segments in the way illustrated on the previous slides.
  - If one processing process dies, the framework and transport are unaffected.
    - Processing process can be restarted (forked again) easily.

- **This is just one idea out of my mind from a recent project.**
  - There are plenty of other ways to do things.
  - But perhaps this can help.