EPJ Web of Conferences **214**, 05029 (2019) https://doi.org/10.1051/epjconf/201921405029
CHEP 2018

Shared Memory Transport for ALFA

Alexey Rybalchenko[1*], *Dennis* Klein[1], *Mohammad* Al-Turany[1], and *Thorsten* Kollegger[1]

[1]GSI - Helmholtzzentrum für Schwerionenforschung, Darmstadt, Germany

Abstract. The high data rates expected for the next generation of particle physics experiments (e.g.: new experiments at FAIR/GSI and the upgrade of CERN experiments) call for dedicated attention with respect to design of the needed computing infrastructure. The common ALICE-FAIR framework ALFA is a modern software layer, that serves as a platform for simulation, reconstruction and analysis of particle physics experiments. Beside standard services needed for simulation and reconstruction of particle physics experiments, ALFA also provides tools for data transport, configuration and deployment. The FairMQ module in ALFA offers building blocks for creating distributed software components (processes) that communicate between each other via message passing.
The abstract "message passing" interface in FairMQ has at the moment three implementations: ZeroMQ, nanomsg and shared memory. The newly developed shared memory transport will be presented, that provides significant performance benefits for transferring large data chunks between components on the same node. The implementation in FairMQ allows users to switch between the different transports via a trivial configuration change. The design decisions, implementation details and performance numbers of the shared memory transport in FairMQ/ALFA will be highlighted.

1 Introduction

ALFA[1] is a modern C++ software framework for simulation, reconstruction and analysis of particle physics experiments. ALFA extends FairRoot[2] to provide building blocks for highly parallelized and data flow driven processing pipelines required by the next generation of experiments, such as the upgraded ALICE detector or the FAIR experiments.

FairMQ[3] is a component in ALFA that provides a C++ Message Queuing Framework that integrates standard industry data transport technologies and provides building blocks for simple creation of data flow actors and pipelines. FairMQ hides transport details behind an abstract interface and ensures best utilization of the underlying transports (zero-copy, high throughput). The framework does not impose any format on the messages.

The next generation particle physics experiments, such as FAIR/GSI experiments and the upgrade of ALICE at CERN, will chop their output data into manageable pieces called time frames. The time frame size can reach up to 11 GB [4]. The large size of the time frame calls for a inter-process transport via shared memory, that will avoid copying the data. This will not only improve throughput, but also relax the memory size requirement for the processing nodes.

*e-mail: a.rybalchenko@gsi.de

Table 1 shows an overview of transports provided by FairMQ. Network transports based on ZeroMQ[5] and nanomsg[6] involve at least one copy of the data for inter-process transfer. Also the new OFI transport[7] that exploits RDMA techniques for network transfers will copy the data between peers by design.

In the following sections we will briefly describe the core concepts of FairMQ, focusing on the transport part. Subsequent section will layout the design and the implementation of the shared memory transport, followed by a section describing the *unmanaged region* feature, which is important for special shared memory use cases (e.g. detector readout hardware). In the last three sections we will show what happens when different transports are combined, what the performance looks like and how we ensure a proper cleanup of the memory.

Table 1. FairMQ transports. Legend: '++' - supported with zero-copy, '+' - supported without zero-copy, '-' - not supported, 'n/a' - combination makes no sense.

node	process	address format	ZeroMQ	nanomsg	shmem	OFI
intra-	intra-	`inproc://endpoint`	++	++	n/a	n/a
intra-	inter-	`ipc://endpoint`	+	+	++	-
		`tcp://host:port`	+	+	++	+
inter-	inter-	`tcp://host:port`	+	+	n/a	+
		`verbs://host:port`	-	-	n/a	++

2 Concepts

FairMQ allows the user to create *devices*. Devices are independent processes with an internal state machine. The devices are controlled and configured via plugins. The transport part of FairMQ follows these general concepts:

- hide all transport-specific details from the user code
- provide clean, unified interface to different data transports
- allow easily combining different transports in one device in a transparent way
- allow switching transport via configuration only, without modifying device/user code -> same API for all transports

Additionally FairMQ follows these data ownership concepts:

- transport message owns data
- user code passes ownership of the data to the framework with the send call
- framework transfers data to next device, passing ownership to receiver (no physical copy of the data with shared memory transport)
- no ownership sharing between different devices – if the same message is needed by more than one receiver it has to be copied.

3 Design and implementation

Internally the shared memory transport consists of two parts. The first part takes care of the memory management - allocations, deallocations and all the controlling related to the created memory objects. The second part is responsible for transfer of the meta data between FairMQ devices - this meta data is used by the devices to locate the memory buffers and

to know how to handle them. The memory management part is implemented on top of the boost::interprocess library[8], which provides cross-platform primitives for shared memory allocation, management and synchronization mechanisms. The meta data transfer part is implemented via ZeroMQ library, which offers flexible, cross-platform and high performance IPC and network communication. The meta data does not have the zero-copy requirement, since it is very small in size and can be handled very efficiently by ZeroMQ. Figure 1 outlines what happens in a simple 1-to-1 device communication via the shared memory transport.

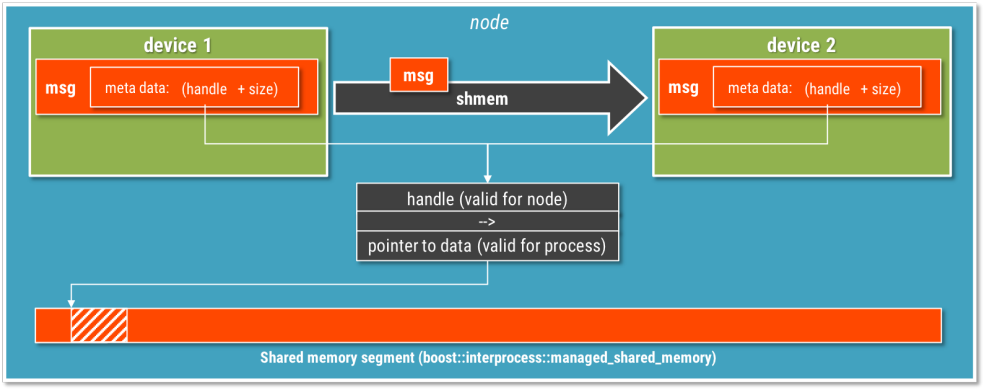

Figure 1. Shared memory transport implementation overview. Sender device (device 1) allocates its data within the shared memory segment and creates a handle to this data that is readable by any process on the node. This handle, together with further meta information is then transfered via ZeroMQ to any receiver, which uses the handle to locate the data.

4 Unmanaged region

The default message creation in FairMQ hides all the memory allocation and management details from the user and simply provides a ready to use buffer for every message. However, in a few use cases, the user might have very specific requirements for the memory layout – typically where hardware needs to write to that memory (e.g. detector readout hardware). Ideally this memory should still be usable with no/minimal copy by further devices in the pipeline. For such cases FairMQ provides UnmanagedRegion component, that allocates memory via the transport allocator and provides it to the user to manage. Messages can then be created out of subset of this region. The framework will do no additional management for the region, except destroy it entirely when the region object goes out of scope. With the shared memory implementation, messages created for this region can be given to devices on the same node without any copy of the data. As additional argument in the region creation, one can provide a callback that will be called once the last user of the message on the node no longer needs the buffer (see Figure 2). This can then be used to cleanup and reuse the memory of the region part. For all devices except the region creator device, the region messages appear as regular messages and no special care has to be taken in the user code to handle region messages, which keeps the usage simple.

5 Connection to other transports

In addition to connecting several devices on a single node via shared memory transport, an efficient mechanism is needed to connect different transports to each other. The most

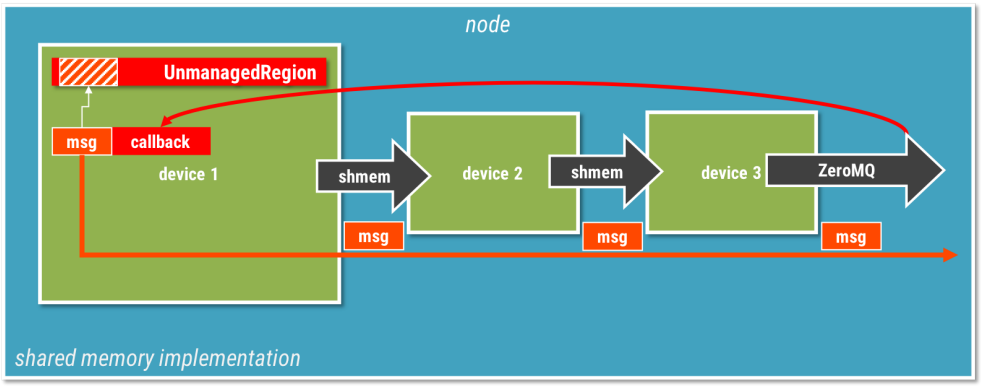

Figure 2. Shared memory unmanaged region.

typical example for this is connecting network transport to shared memory. Ideally such connection should involve no additional copy. Each channel in a FairMQ device can either use the default device transport or define its own. This allows simple logical connection of different transport together. Furthermore, no specific transport knowledge is needed in the device code – it knows only channel names – the actual transport of a channel is decided by the configuration. Whether a transport combination can be zero-copy or not depends on the capabilities of each transport. The main requirement to allow zero-copy is for the transport to be able to read/write directly to a provided buffer. Not all transport technologies support such scenario without a copy. For example both ZeroMQ and nanomsg will write to their own memory upon receiving data, so moving the data to shared memory will require additional copy. The OFI transport and shared memory transport are designed to support a zero-copy data transfer between themselves. The FairMQ interface hides such details internally, so that the user doesn't have to care in their code about different transport combinations - the framework will encure zero-copy connection if it is possible, or will hide the copy if one has to be done.

6 Performance

Beyond relaxing the memory size requirements for the processing nodes, shared memory transport also has potential to improve the data throughput by avoiding expensive copies and reducing CPU usage needed for such copies. Figure 3 demonstrates the transfer rate between two devices using shared memory transport with varying message size. The transport can maintain a transfer rate of about 1 MHz with 1-to-1 connections independent of the message size. Because both the allocation within a memory segment and the meta data transfer are very fast, the effective performance with the rising message size depends on the amount of the available memory and how fast the consumers will release it.

Table 2 shows the CPU usage of the shared memory transport compared to the one of the ZeroMQ/TCP transport. For this test both transports have been given a rate limit of 2.5 GB/s to see how the CPU usage compares with the same work size. The table shows that the shared memory transport has significantly lower CPU usage.

Figure 3. Shared memory performance: transfer rate between two FairMQ devices with varying message size. Hardware used for the test: dual Intel Xeon E5-2660v3 @ 2.6 GHz, 128GB RAM, 60GB shared memory segment.

	ZeroMQ: TCP	shmem
sender	68.5%	1.1%
receiver	89.1%	0.9%

Table 2. CPU usage of shared memory transport compared to TCP via ZeroMQ. For the comparison, the transfer rate has been limited to 2.5 GB/s for each transport. Hardware used for the test: dual Intel Xeon E5-2660v3 @ 2.6 GHz, 128GB RAM, 60GB shared memory segment.

7 Memory cleanup

Shared memory transport obtains resources from the system that can outlive the requesting process. Therefore it is important to ensure a proper release of these resources when they are no longer needed, but also in cases when devices shut down incorrectly (e.g. due to a crash). The transport handles the cleanup behind the scenes and does so on multiple levels. First level is taking care of a proper cleanup when devices shutdown properly - resources will be released as they go out of scope and the last user will release all the resources upon shutdown. Should one or more devices crash and fail to release their resources orderly, a shared memory monitoring deamon will take over the cleanup and will release the resources once the last user has stopped/crashed. The shared memory resources are associated to the current user and to a session id, such that many users can run several sessions with independent shared memory resources without them (or their cleanup) conflicting with each other.

8 Conclusion

With the described shared memory transport FairMQ extends its offer of transport implementations with one that can significantly optimize inter-process device communication and reduce memory and CPU requirements for data transfers. The transport can be connected to other efficient inter-node transports, such as RDMA-capable transports, while maintaining zero-copy data handling. The provided interface hides the low level details from the user and offers a clearly defined ownership model for the transfered data. Special cases, where very specific memory layout is necessary, are supported via an advanced interface. Cleanup of the shared memory resources is ensured for both orderly shutdown or in case of a crash.

References

[1] M. Al-Turany et al., "ALFA: The new ALICE-FAIR software framework", Journal of Physics: Conference Series, Volume 664 (2015)

[2] https://github.com/FairRootGroup/FairRoot (accessed 2018-10-22)

[3] https://github.com/FairRootGroup/FairMQ (accessed 2018-10-22)

[4] P. Buncic, M. Krzewicki, P. Vande Vyvre, et al.: Technical Design Report for the Upgrade of the Online-Offline Computing System, 2015, CERN, Geneva, The LHC experiments Committee.

[5] http://zeromq.org/ (accessed 2018-10-22)

[6] https://nanomsg.org/ (accessed 2018-10-22)

[7] D. Klein, "RDMA-accelerated data transport in ALFA", CHEP (to be published)

[8] https://www.boost.org/doc/libs/1_68_0/doc/html/interprocess.html/ (accessed 2018-10-22)