Exploring polyglot software frameworks in ALICE with `FairMQ` and `fer`

Sébastien Binet[1,*] on behalf of the ALICE collaboration

[1]CNRS/IN2P3, LPC, F-63000 Clermont-Ferrand, France

Abstract. In order to meet the challenges of the Run 3 data rates and volumes, the ALICE collaboration is merging the online and offline infrastructures into a common framework: ALICE-O^2. O^2 is based on FairRoot and FairMQ, a message-based, multi-threaded and multi-process control framework. In FairMQ, processes (possibly on different machines) exchange data via message queues either through 0MQ or nanomsg. In turn, this enables developers to write their reconstruction or analysis process in whatever language they choose or deem appropriate for the task at hand, as long as that programming language can send and receive data through these message queues. This paper introduces `fer`, a Go-based toolkit that interoperates with the C++ toolkit FairMQ, to explore the realm of polyglot distributed frameworks.

1 Introduction

The ALICE collaboration is merging the online and offline infrastructures into a common framework: ALICE-O^2 [1]. This work is needed to meet the challenges of the Run 3 data rates and volumes. O^2 is a message-based, multi-threaded and multi-process control framework based on two libraries: FairRoot [2] and FairMQ [3]. Sophisticated processing pipelines can be modeled with FairMQ: router/dealer, request/reply, publish/subscribe, client/server, etc. The nodes of these pipelines are processes (possibly on different machines) exchanging data via message queues either through `ZeroMQ` [4] or `nanomsg` [5].

From the standpoint of the FairMQ toolkit, the programming language used to implement a given process in the pipeline is irrelevant, as long as data is correctly propagated through the message queues. This enables developers to write their reconstruction or analysis process in whatever language they choose or deem appropriate for the task at hand.

This paper presents `fer`, a Go-based [6] library compatible and interoperable with FairMQ. It starts with a brief introduction of the builtin features that make Go a solid choice when dealing with I/O and concurrency. The principal components of `fer` and how they interact with C++ FairMQ will then be described. Finally, Sec. 4 will report on the performance (CPU, VMem) of `fer` and conclude with the main figures of merit of `fer`, in the context of deployment in a distributed computing setup.

*e-mail: binet@cern.ch

2 FairMQ concepts

FairMQ is a distributed processing toolkit, written in C++, with pluggable transports (`ZeroMQ`, `nanomsg`, `shmem`, InfiniBand). In FairMQ, the bulk of the processing is performed by devices: UNIX processes that are connected with other devices through message queues. Devices can be connected in various ways, through a wide variety of topologies: router/dealer, request/reply, publish/subscribe, client/server, etc. The medium used for the connection between devices can be chosen among the following options: `tcp`, `udp`, `ipc`, `inproc`, shared-memory. Fig. 1 shows a typical FairMQ processing pipeline, fanning out data read off a source (`sampler`) to a pair of processors and then fanning in the data to a single output sink.

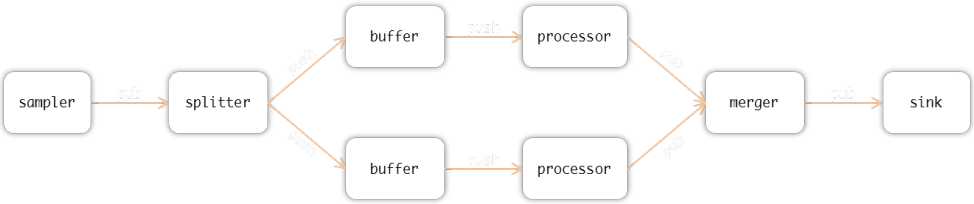

Figure 1: Example of a processing pipeline topology, implemented with FairMQ. Each box is a UNIX process, possibly on different machines, connected with other UNIX processes. Data "flows" from left to right.

FairMQ applications are currently configured via a `JSON` configuration file, declaring the devices, the ports they are listening on or sending data to, the type and topology of the message queues, etc. An example of such a configuration file is shown in listing 1.

```
{
    "fairMQOptions":
    {
        "devices":
        [{
            "id": "sampler1",
            "channels":
                    [{
                "name": "data1",
                "sockets":
                            [{
                    "type": "push",
                    "method": "bind",
                    "address": "tcp://*:5555",
                    "sndBufSize": 1000,
                    "rcvBufSize": 1000,
                    "rateLogging": 0
                }]
            }]
        },
[...]
}
```

Listing 1: Example of a JSON configuration file for a FairMQ processing pipeline.

Topologies in FairMQ can be described in JSON, like in listing 1, but other formats and mechanisms are available: XML or via the Dynamic Deployment System (DDS) [7]. Devices

are completely agnostic with regard to the technology used to modify, deploy, and push the final configuration. Once the user has written the correct configuration, declaring inputs and outputs of the whole pipeline, the C++ device is launched and provided with a configuration. It essentially runs an infinite `for` loop, waiting for data to be received, sending refined data down the pipeline. Devices are also listening for external commands that control and modify the finite state machine of a FairMQ application. An abridged declaration of the C++ base class for devices is shown in listing 2.

```
class FairMQDevice : <...> {
public:
 int
 Send(const std::unique_ptr<FairMQMessage>& msg,
      const std::string& chan, const int i) const;

 int
 Receive(const std::unique_ptr<FairMQMessage>& msg,
         const std::string& chan, const int i) const;

protected:
  virtual void Init();    virtual void InitTask();
  virtual void Run();
  virtual bool ConditionalRun();
  virtual void Pause();
  virtual void Reset();   virtual void ResetTask();
};
```

Listing 2: Excerpt of the `FairMQDevice` class, the main API for implementing a device.

Users are supposed to at least override the `FairMQDevice::Run()` method in their derived classes. This method is where data is sent to the downstream devices (using the `FairMQDevice::Send(...)` method) or received from upstream devices (using the `FairMQDevice::Receive(...)` method). As a device is usually instantiated within its own UNIX process, users are usually free to leverage any kind of library or programming paradigm inside the boundaries of their own memory address space.

With FairMQ's microservice-like architecture, processing pipelines can horizontally scale much more easily when data taking or data processing demands it. Indeed, it is just a matter of adding more processing resources to the pool and connect them via message queues to accomodate a change in the processing load. Abstracting away the minute details of how devices are connected and distributed (all in the same address space, on different machines, etc.) allows users to very easily isolate a faulty device and mitigate its impact on others, *e.g.* preventing it from corrupting memory of other devices. This also allows to use processes' boundaries to segregate components that are multithread friendly from others that are not.

On one hand, requiring that data is communicated between devices through these message queues healthly constrains the set of C++ features one can use to implement the data model. Indeed, the data being exchanged will tend to resemble plain old data (POD). On the other hand, relying on off the shelf libraries such as `ZeroMQ` or `nanomsg` to implement the actual data transport between devices, enables the implementation of devices in any language that has support for `ZeroMQ` or `nanomsg`. A FairMQ pipeline could theoretically be composed of n devices, each device implemented in a different programming language, presumably one that fits best the task at hand.

In the following, the feasability of a `Go` [6] based toolkit, interoperable with FairMQ, `fer`, is investigated.

3 Elements of Go

Go is an open source general programming language with built-in support for concurrency programming. It has been released in 2009 under the BSD-3 license. Go has a static type system with first-class functions, closures, and interfaces. The general syntax of Go is close to that of C and C++. Even if Go programs are statically typed, code verbosity is limited thanks to Go's type inference system. Go's concurrency primitives derive from Hoare's Communicating Sequential Processes (CSP) [8]: *goroutines* and *channels*. Channels are typed conduits that connect goroutines together. Goroutines are very lightweight green threads: it is possible to schedule thousands of them on regular hardware. This is achieved thanks to the Go runtime multiplexing goroutines on OS threads and the ability to grow and shrink each goroutine's stack as needed. The last element for easy concurrency programming is the garbage collector that takes care of freeing memory from the correct thread, when it is not needed anymore.

Go is available on all major platforms (Linux, Windows, macOS, Android, iOS, etc.) and for many architectures (`amd64`, `arm64`, `i386`, `s390x`, `mips64`, etc.). The Go toolchain can very easily cross-compile from one OS/Arch pair to any other pair, by just modifying two environment variables. The Go toolchain generates completely statically compiled binaries. This enables a very efficient deployment model where binaries produced on a developer laptop can be quickly provisioned on a bare metal production cluster. Finally, compiling Go code is not only fast thanks to its package and import system, it is also regular and automatically handles dependencies, recursively discovering and installing them. Indeed, installing Go packages is usually just a matter of entering the following command at the prompt:

```
%> go get -v github.com/sbinet-alice/fer
github.com/sbinet-alice/fer/mq
github.com/pkg/errors
golang.org/x/net/context
github.com/sbinet-alice/fer/config
nanomsg.org/go-mangos
[...]
nanomsg.org/go-mangos/transport/tcp
github.com/sbinet-alice/fer/mq/nanomsg
github.com/sbinet-alice/fer/mq/zeromq
```

This single command works on all platforms and operating systems supported by the Go toolchain.

4 Design and components of `fer`

As introduced in Sec. 2, implementing a toolkit compatible with FairMQ requires to be able to configure FairMQ-like devices, using *e.g.* the JSON configuration file format; to create topologies of devices according to the configuration and connected via a transport library (*e.g.* `ZeroMQ` or `nanomsg`); and finally to execute the main routines of the devices.

The requirements above are met by `fer` [9], a toolkit implemented in Go. Similar to its `C++` sibling, `fer` exposes the concept of a device by way of the `Device` interface, as shown in listing 3.

A type implements the `Device` interface as soon as it has a method named `Run()` that takes a `Controller` interface and returns an `error`. The `Controller` interface is shown in listing 4. It is used to retrieve named channels bound to incoming or outgoing data, through the `Chan()` method. An additional `Done()` method exposes transitions in the `fer` final state machine (`RUN`, `PAUSE`, `STOP`, etc.)

The input configuration is parsed by the `fer/config` package and passed to the devices via the `DevConfigurer` interface, if that device implements it.

```go
package fer

import "github.com/sbinet-alice/fer/config"

// Device is a handle to what users get to run via the Fer toolkit.
type Device interface {
        Run(ctl Controller) error
}

type DevConfigurer interface { Configure(cfg config.Device) error }
type DevIniter     interface { Init(ctl Controller) error }
type DevPauser     interface { Pause(ctl Controller) error }
type DevReseter    interface { Reset(ctl Controller) error }
```

Listing 3: Excerpt of the `fer` package, with the main interfaces it exposes. Users are only required to implement the `Device` interface, other interfaces are optional.

```go
// Controller controls devices execution and gives a device access to input and
// output data channels.
type Controller interface {
        Logger
        Chan(name string, i int) (chan Msg, error)
        Done() chan Cmd
}

// Msg is a quantum of data being exchanged between devices.
type Msg struct {
        Data []byte // Data is the message payload.
        Err  error  // Err indicates whether an error occured.
}

// Cmd describes commands to be sent to a device, via a channel.
type Cmd byte
```

Listing 4: Declaration of the `Logger` and `Controller` interfaces in the `fer` package. `Controller`s expose communication channels that can exchange data messages, `Msg`, that embed the actual payload or an error if any.

4.1 Implementing a `fer` device

This section shows how implementing a `fer` device and integrating it inside a C++ pipeline can be done. Listing 5 shows how one can implement the `processor` from Fig. 1. The `processor` retrieves two channels, the `"data1"` input channel and the `"data2"` output channel. The `Run()` method contains an infinite `for` loop. At each iteration, the device waits for data coming in from either the `idatac` or the `Done()` channel. When the input channel is ready – a message has been delivered to the device – the associated payload is modified by the device (just appending some data) and then sent downstream through the output channel `odatac`. Concurrently, if any command is sent on the `Done()` channel, the `for` loop exits and the Go runtime can reclaim resources used by the associated goroutine.

Once a type implements the `fer.Device` interface, a value of that type can be created and passed to the `fer.Main` function that takes care of the minute details of scheduling devices, establishing socket connections and forwarding external commands. This simplifies the creation of the main entry point for the whole program, as shown in listing 6.

Compiling and running this processor is done via:

```go
package mydevice

import (
        "github.com/sbinet-alice/fer"
        "github.com/sbinet-alice/fer/config"
)

type processor struct {
        cfg    config.Device
        idatac chan fer.Msg
        odatac chan fer.Msg
}

func (dev *processor) Configure(cfg config.Device) error {
        dev.cfg = cfg
        return nil
}

func (dev *processor) Init(ctl fer.Controller) error {
        idatac, err := ctl.Chan("data1", 0) // handle err
        odatac, err := ctl.Chan("data2", 0) // handle err
        dev.idatac = idatac
        dev.odatac = odatac
        return nil
}

func (dev *processor) Run(ctl fer.Controller) error {
        str := " (modified by "+dev.cfg.Name()+")"
        for {
                select {
                case data := <-dev.idatac:
                        out := append([]byte(nil), data.Data...)
                        out = append(out, []byte(str)...)
                        dev.odatac <- fer.Msg{Data: out}
                case <-ctl.Done():
                        return nil
                }
        }
}
```

Listing 5: Implementation of the `DevConfigurer`, `DevIniter` and `Device` interfaces.

```
%> go get ./my-device
%> $GOPATH/bin/my-device --id processor --mq-config ./path-to/config.json
```

4.2 Go implementation of `ZeroMQ`

At the beginning of the development of `fer`, a pure Go implementation of the `nanomsg` package was readily available but not for `ZeroMQ`. Only a so-called `cgo` based package existed, one that linked to the C/C++ `ZeroMQ` library and wrapped the C/C++ calls with Go functions and types. Calling Go functions that call C/C++ and back is not only costly – around ten times a normal function call – it also breaks the easy compilation, installation, and deployment promises that Go users are accustomed to. To correct this deficiency, a pure Go package that implements the required subset of the `ZeroMQ` protocol and sockets to provide interoperability with the C++ implementation was developed: `go-zeromq/zmq4` [10].

```go
package main

func main() {
        err := fer.Main(&processor{})
        if err != nil {
                log.Fatal(err)
        }
}
```

Listing 6: Main program for the `processor` example device. For brevity, imports are omitted.

As shown in Fig. 2, the mean value for the time of flight of a single `uint64` token exchanged between a source device, a processor device and a sink device, goes from $\sim$ 70 μs for the `czmq` C++ version (called from Go) down to $\sim$ 45 μs for the pure Go `nanomsg` and `zeromq` transports. The maximum resident set size (`MaxRSS` from `/proc/self/statm`) for all configurations was around 22 Mb.

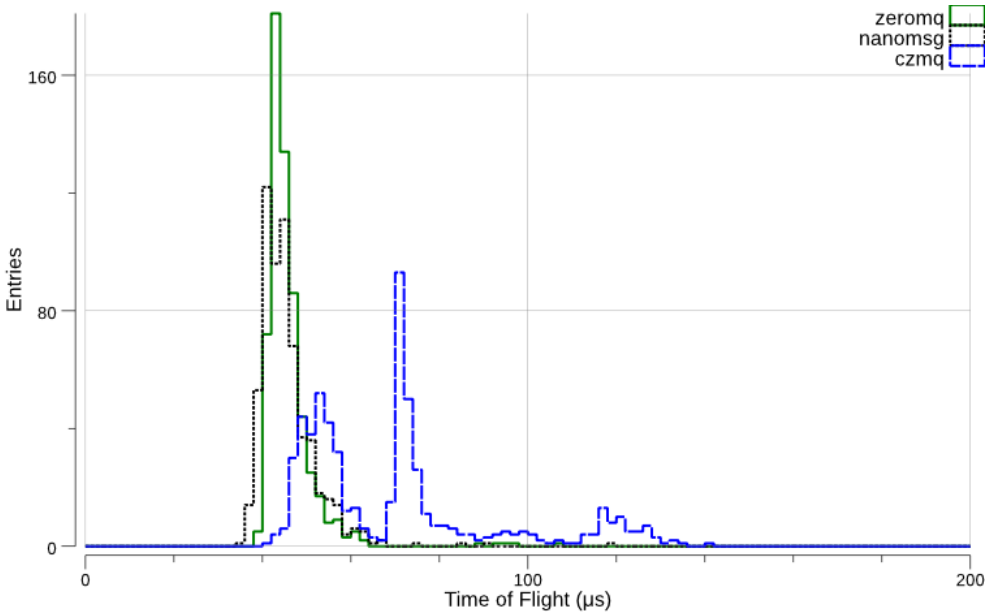

Figure 2: Distribution of the time of flight of a single token (`uint64`) through a pipeline `p = { source|processor|sink}`, for different transports and protocols: `zeromq`, `nanomsg` and `czmq`. `czmq` is the C/C++ `ZeroMQ` library called from Go.

5 Conclusions

The ALICE-O^2 framework based on the message passing infrastructure from FairMQ enables one to build applications that resemble microservices architectures. These types of architectures allow systems to scale horizontally quite easily, by just adding new processor devices and fanning out data to these new processors, in *e.g.* a round-robin fashion. FairMQ, and

microservices in general, is language agnostic: developers can thus implement their device in whatever language they deem adequate for the task at hand. This paper introduced `fer`, a Go-based toolkit that interoperates with the C++ toolkit FairMQ, to explore the realm of polyglot distributed frameworks. The `fer` toolkit provides a fast, "one-command" installation procedure that works on all platforms and operating systems supported by the Go toolchain. It also provides Grid-friendly deployments thanks to its single, statically compiled, binary. As such it can be a great testbed environment to explore new avenues.

References

[1] M. Al-Turany *et al.* ALFA: The new ALICE-FAIR software framework, Journal of Physics: Conference Series, **Volume 664**, 072001 (2015)

[2] M. Al-Turany *et al.* The FairRoot framework, Journal of Physics: Conference Series, **Volume 396**, 022001 (2012)

[3] `https://github.com/FairRootGroup/FairMQ`, visited: August 2018

[4] `http://zeromq.org/` (`v4.2.5`, visited: August 2018)

[5] `https://nanomsg.org/` (`v1.1.5`, visited: August 2018)

[6] The `Go` programming language, `https://golang.org` (`v1.11`, visited: August 2018)

[7] A. Manafov and A. Lebedev, DDS: The Dynamic Deployment System, GSI Report 2015-1, p 371 (2015) `doi:10.15120/GR-2015-1-INFRASTRUCTURE-IT-04`

[8] C. A. R. Hoare, Communications of the ACM **Volume 21** Issue 8, pp 666-677 (1978)

[9] `https://github.com/sbinet-alice/fer`, `doi:10.5281/zenodo.1470586`

[10] `https://github.com/go-zeromq/zmq4`, `doi:10.5281/zenodo.1470580`