Exploring polyglot software frameworks in ALICE with FairMQ & fer

Sebastien Binet
CNRS/IN2P3/LPC
binet@cern.ch

Laurent Aphecetche
CNRS/IN2P3/Subatech
laurent.aphecetche@cern.ch

2018-07-09

ALICE

2018-07-09

CHEP-2018
ALICE is working on a new software framework for the O2 project:
- a new framework to tackle the challenges of Run-3 data taking
- a new framework that brings together Online and Offline
- a new framework that builds upon FairRoot and FairMQ

The FairRoot framework is an object oriented simulation, reconstruction and data analysis framework. It includes core services for detector simulation and offline analysis of particle physics data. FairRoot is the standard simulation, reconstruction and data analysis framework for the FAIR experiments at GSI Darmstadt.
FairMQ is a distributed processing toolkit, written in C++, with pluggable transports (ZeroMQ, nanomsg).

Each box can be a different process, possibly on a different remote machine. This architecture enables a much smoother horizontal scaling when data taking or data processing demands it. Each box may be connected to another via various protocols: tcp, udp, ipc, inproc, shared-memory.
FairMQ

FairMQ has a concept of Devices which are executed and connected together to form various topologies:

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();
};

Users are supposed to at least override FairMQDevice::Run().
Topologies can be created and described via JSON files (or XML, or via DDS (see #407)):

```json
{
    "fairMQOptions": {
        "devices": [
            {
                "id": "sampler1",
                "channels": [
                    {
                        "name": "data1",
                        "sockets": [
                            {
                                "type": "push",
                                "method": "bind",
                                "address": "tcp://*:5555",
                                "sndBufSize": 1000,
                                "rcvBufSize": 1000,
                                "rateLogging": 0
                            }
                        ]
                    }
                ]
            }
        ],
        [...]
    }
}
```
As FairMQ is distributed with each device talking over ZeroMQ or nanomsg, one can write each device in any language (which has support for nanomsg or ZeroMQ). $OTHER\_LANGUAGE$ could be:

- Java
- Perl
- Python
- Rust
- ...

or even just bash + netcat :) Let’s do that in Go.
Interlude: Go

http://golang.org
Elements of Go

- Russ Cox, Robert Griesemer, Ian Lance Taylor, Rob Pike, Ken Thompson

- **Concurrent, garbage-collected**
  - An Open-source general programming language (BSD-3)
  - feel of a **dynamic language**: limited verbosity thanks to the *type inference system*, map, slices
  - safety of a **static type system**
  - compiled down to machine language (so it is fast, goal is within $\pm 10\%$ of C)
  - **object-oriented** but w/o classes, **builtin reflection**
  - first-class functions with **closures**
  - implicitly satisfied **interfaces**

Available on all major platforms (Linux, Windows, macOS, Android, iOS, ...) and for many architectures (amd64, arm, arm64, 386, s390x, mips64, ...)

Go’s concurrency primitives - **goroutines** and **channels** - derive from Hoare’s Communicating Sequential Processes (CSP.)

Goroutines are like threads: they share memory.

But cheaper:

- Smaller, segmented stacks.
- Many goroutines per operating system thread
Ok, let’s write a FairMQ compatible, inter-operable, toolkit in Go:

```
github.com/sbinet-alice/fer
```

- create `fer.Device`
- create topologies using the same JSON files
- connect devices via ZeroMQ
- connect devices via nanomsg

All topologies supported by FairMQ should be supported by fer too.
One can mix and match C++ and Go devices, see:
```
github.com/FairRootGroup/FairRoot/tree/dev/examples/advanced/GoTutorial
```
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
}

Users need to implement this interface.

Optionally, they may also implement:

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
}
// Controller controls devices execution and gives a device access to input and output data channels.

```go
type Controller interface {
    Logger
    Chan(name string, i int) (chan Msg, error)
    Done() chan Cmd
}
```

Controller is used to give access to input/output channels to the user device. The Done() channel is used to signal user devices that processing should be somehow interrupted or paused.

Messages and commands are defined as:

```go
// 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 occurred.
}
```

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

type Cmd byte
```
fer tutorial
import {
    "github.com/sbinet-alice/fer"
    "github.com/sbinet-alice/fer/config"
}

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

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

func (dev *processor) Init(ctl fer.Controler) error {
    idatac, err := ctl.Chan("data1", 0) // handle err // HL
    odatac, err := ctl.Chan("data2", 0) // handle err // HL
    dev.idatac = idatac
    dev.odatac = odatac
    return nil
}
func (dev *processor) Run(ctl fer.Controler) error {
    str := " (modified by "+dev.cfg.Name()+")"
    for {
        select {
            case data := <-dev.idatac: // HL
                out := append([]byte(nil), data.Data...)
                out = append(out, []byte(str)...)
                dev.odatac <- fer.Msg{Data: out} // HL
            case <-ctl.Done():
                return nil
        }
    }
}

func (dev *processor) Pause(ctl fer.Controler) error {
    return nil
}

func (dev *processor) Reset(ctl fer.Controler) error {
    return nil
}
```go
func main() {
    err := fer.Main(&processor{}) // HL
    if err != nil {
        log.Fatal(err)
    }
}
```

Build and run like so:

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

Of course, fer is compatible with JSON files from FairMQ...

More docs:
godoc.org/github.com/sbinet-alice/fer
ZeroMQ (push/pull, pub/sub, ...), pure-Go go-zeromq/zmq4
nanomsg (push/pull, pub/sub, ...), pure-Go go-mangos/mangos
devices’ executables statically compiled, easily cross- compilable
tcp, inproc and ipc supported
with nanomsg: transport via WebSockets (think: monitoring via a web server)
FairMQ-compatible program options
Conclusions

fer is a ‘FairMQ‘-compatible toolkit, written in Go.
Straightforward installation:

- install Go for your platform (macOS, linux, windows, ...)
- install fer:
  
  $> \text{go get github.com/sbinet-alice/fer}

and voilà!

fer provides interoperability with FairMQ in a language that is safe, concurrent-friendly, easy to deploy, and ready for the cloud.

With FairMQ and the microservice-like architecture it enables, one can migrate each device from $\text{LANG1}$ to $\text{LANG2}$, adiabatically.
Extra material
• shared-memory "transport": not there yet
• plugins for the Dynamic Deployment System (DDS)
• state machine more limited than FairMQ's
• no support yet for AliceO2's Data Processing Layer (DPL, see #328)
Control panel

2018-06-14
19:14:38: "HELLO (modified by processor)"

[...]
2018-06-14
19:15:36.9:
"HELLO (modified by processor)"
2018-06-14
19:15:37: "HELLO (modified by processor)"
2018-06-14
19:15:37: "HELLO (modified by processor)"
2018-06-14
19:15:37:1: "HELLO (modified by processor)"
2018-06-14
19:15:37:1: "HELLO (modified by processor)"
2018-06-14
19:15:37:2:
“HELLO (modified by processor)"
2018-06-14
19:17:00.5:
"HELLO (modified by processor)"
2018-06-14
19:17:00.6:
"HELLO (modified by processor)"
2018-06-14
19:17:00.7:
"HELLO (modified by processor)"
2018-06-14
19:17:00.8:
"HELLO (modified by processor)"
$> ./fer-ex-raw-ctl -timeout=20s -transport=nanomsg -protocol=ipc
real=20.06 user=23.77 sys=17.52 CPU=205% MaxRSS=18808 I/O=0/176
$> ./fer-ex-raw-ctl -timeout=20s -transport=nanomsg -protocol=tcp
real=20.05 user=17.98 sys=24.40 CPU=211% MaxRSS=19548 I/O=0/80
$> ./fer-ex-raw-ctl -timeout=20s -transport=zeromq -protocol=ipc
real=20.06 user=27.62 sys=15.63 CPU=215% MaxRSS=19312 I/O=0/80
$> ./fer-ex-raw-ctl -timeout=20s -transport=zeromq -protocol=tcp
real=20.06 user=21.28 sys=22.28 CPU=217% MaxRSS=19976 I/O=0/80