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

CHEP-2018

AliceO2

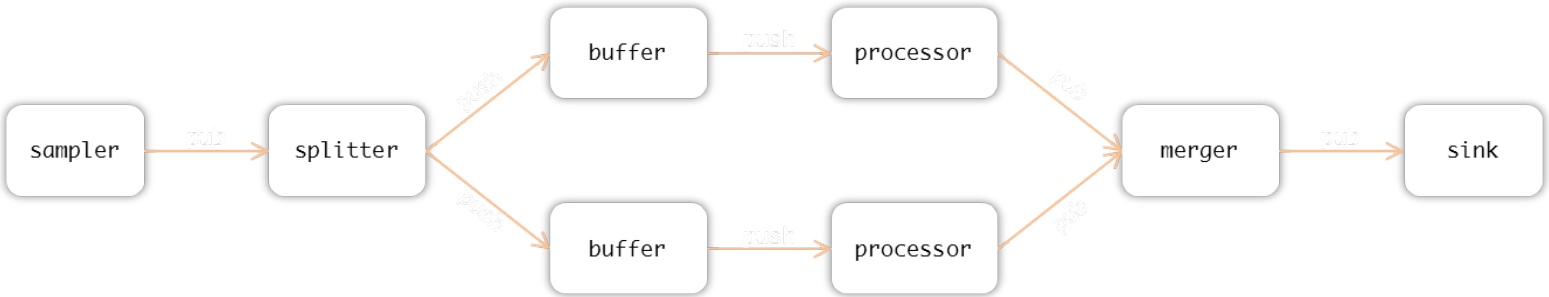
ALICE is working on a new software framework for the O2 project:

- 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

FairMQ (#454) 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()`.

FairMQ

Topologies can be created and described via JSON files (or XML, or via DDS (see #407)):

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

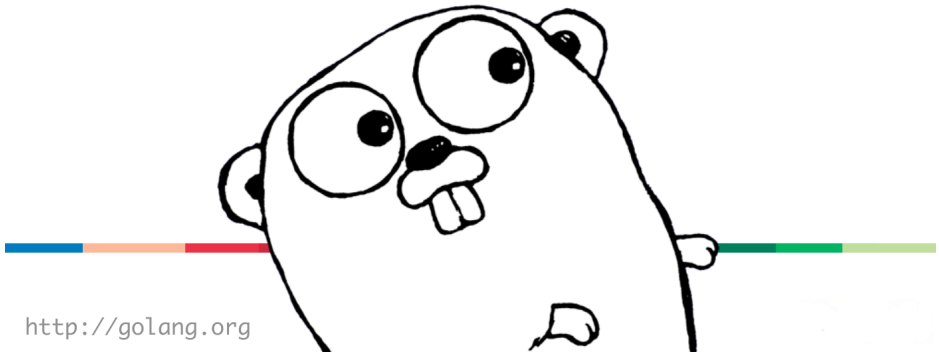
fer

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

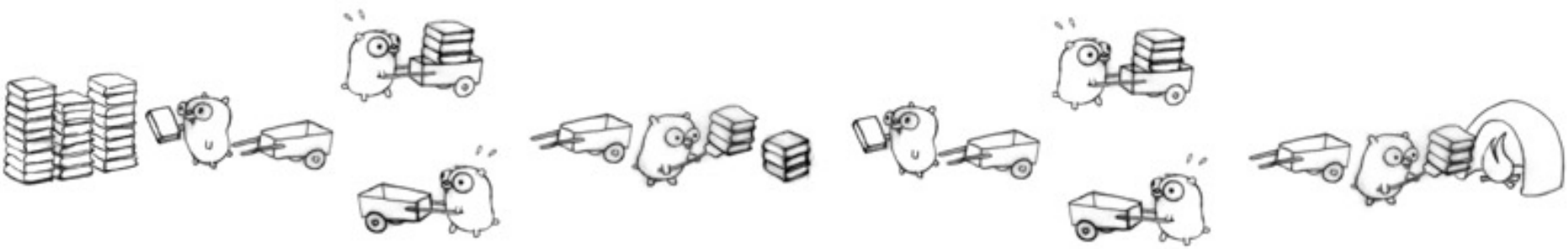


Elements of Go

- Russ Cox, Robert Griesemer, Ian Lance Taylor, Rob Pike, Ken Thompson
- **Concurrent**, **garbage-collected**
- An Open-source general progamming 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`, ...)

Concurrency



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

fer

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

fer - device

```
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 Controler) 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 Controler) error
}
type DevPauser interface {
        Pause(ctl Controler) error
}
type DevReseter interface {
        Reset(ctl Controler) error
}
```

fer - controler

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

`Controler` 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:

```
// 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
```

`fer` tutorial

fer - processor

```
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
}
```

fer - processor

```
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
}
```

fer - processor

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

Build and run like so:

```
$> 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

fer features

- `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`:

  ```
  $> 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 `$LANG1` to `$LANG2`, adiabatically.

Thank you

Sebastien Binet
CNRS/IN2P3/LPC-Clermont
github.com/sbinet
@0xbins
sebastien.binet@clermont.in2p3.fr

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

Extra material

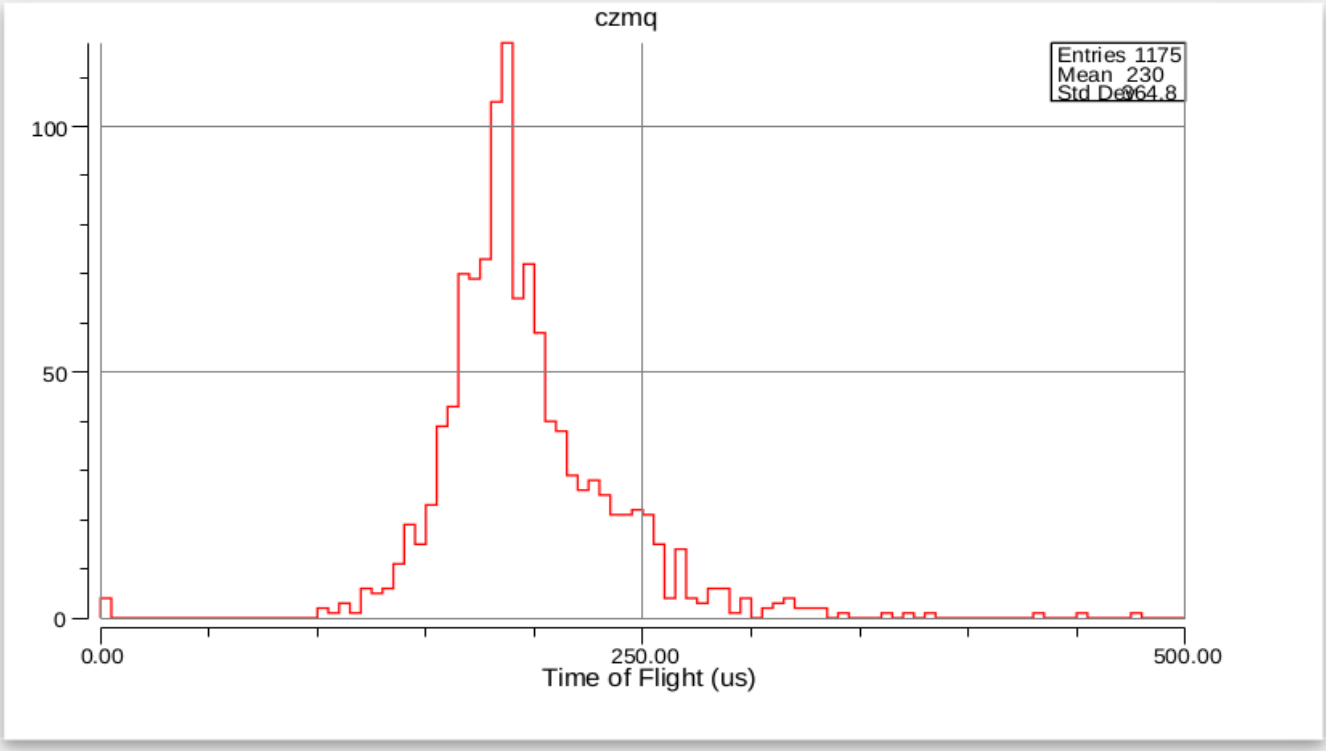
fer non-features

- 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

Launch

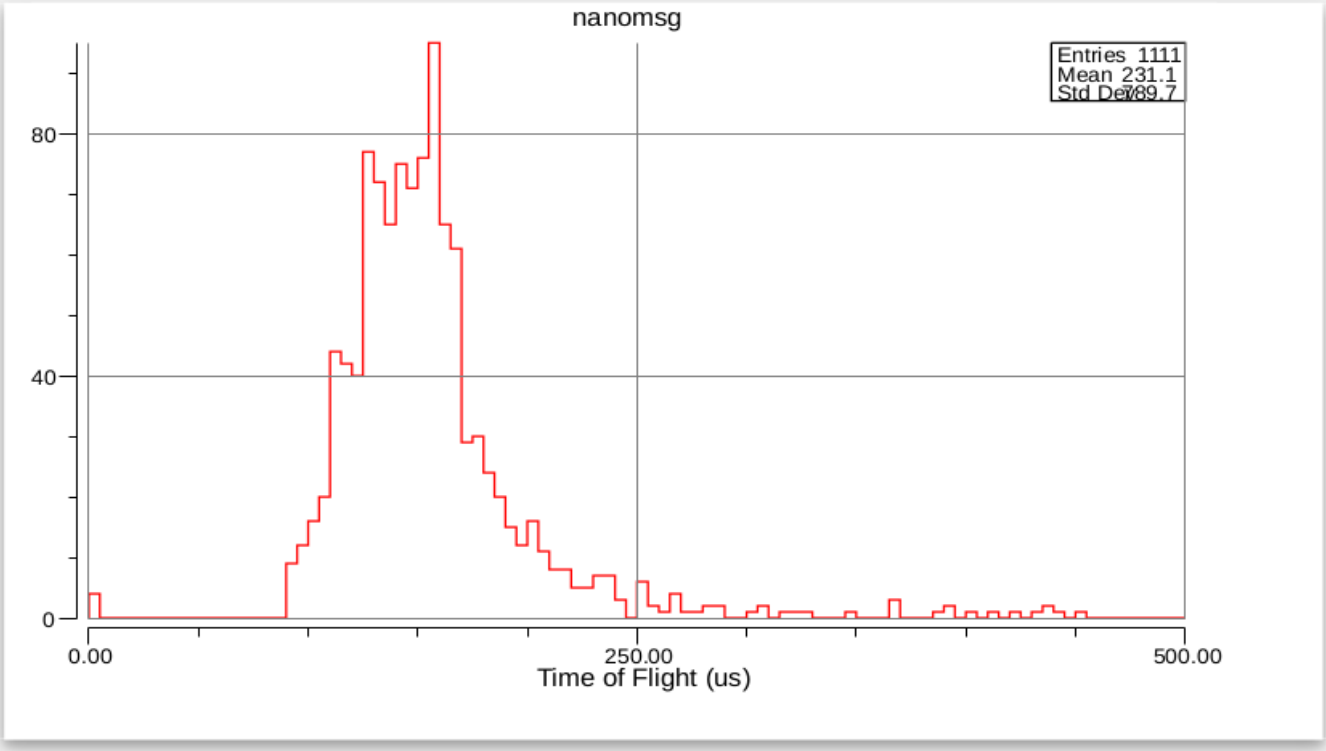
2018-06-14 19:14:38: "HELLO (modified by processor)"
[...]
2018-06-14 19:15:36.9: "HELLO (modified by processor)"
2018-06-14 19:15:37: "HELLO (modified by processor)"
2018-06-14 19:15:37: "HELLO (modified by processor)"
2018-06-14 19:15:37.1: "HELLO (modified by processor)"
2018-06-14 19:15:37.1: "HELLO (modified by processor)"
2018-06-14 19:15:37.2:

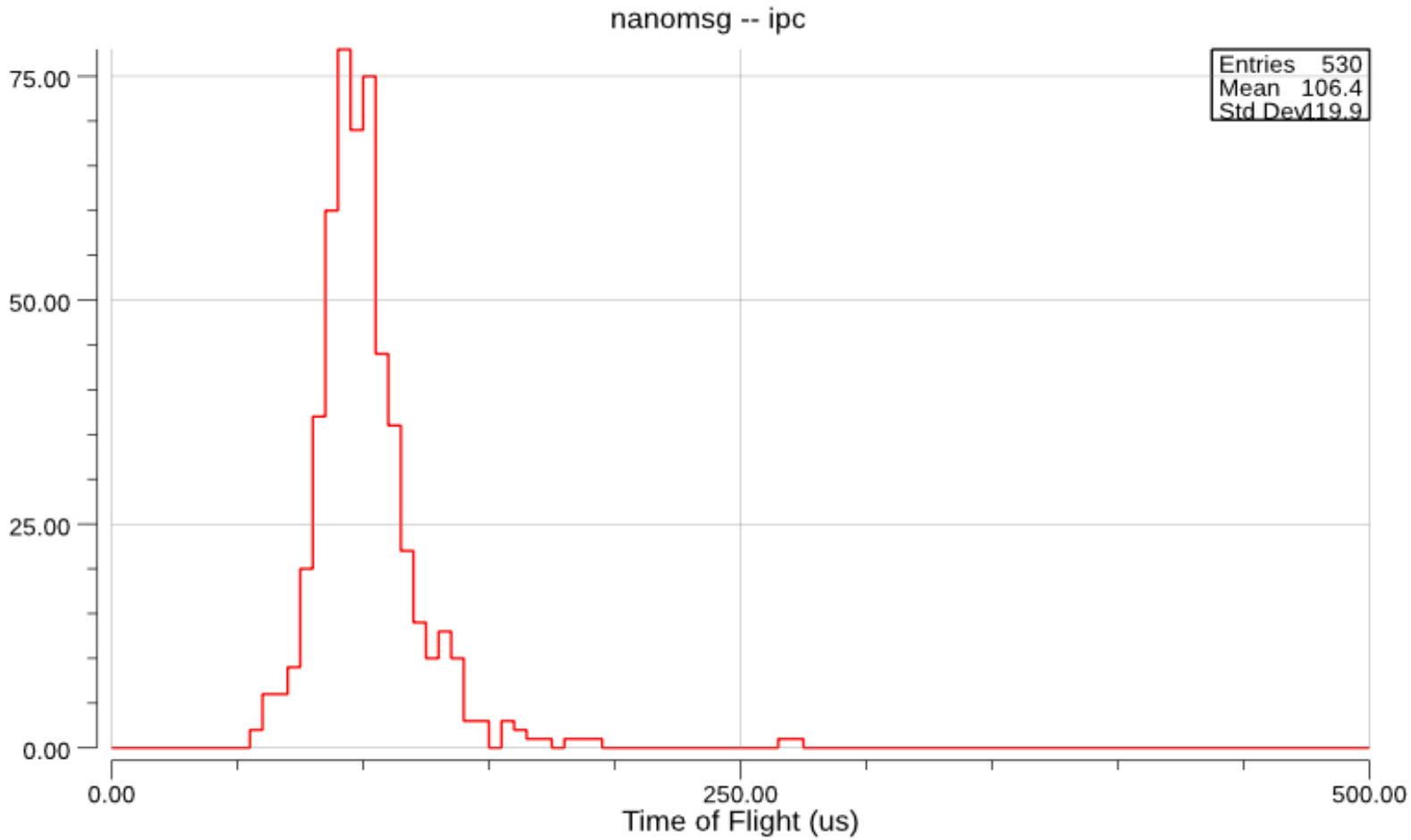


Control panel

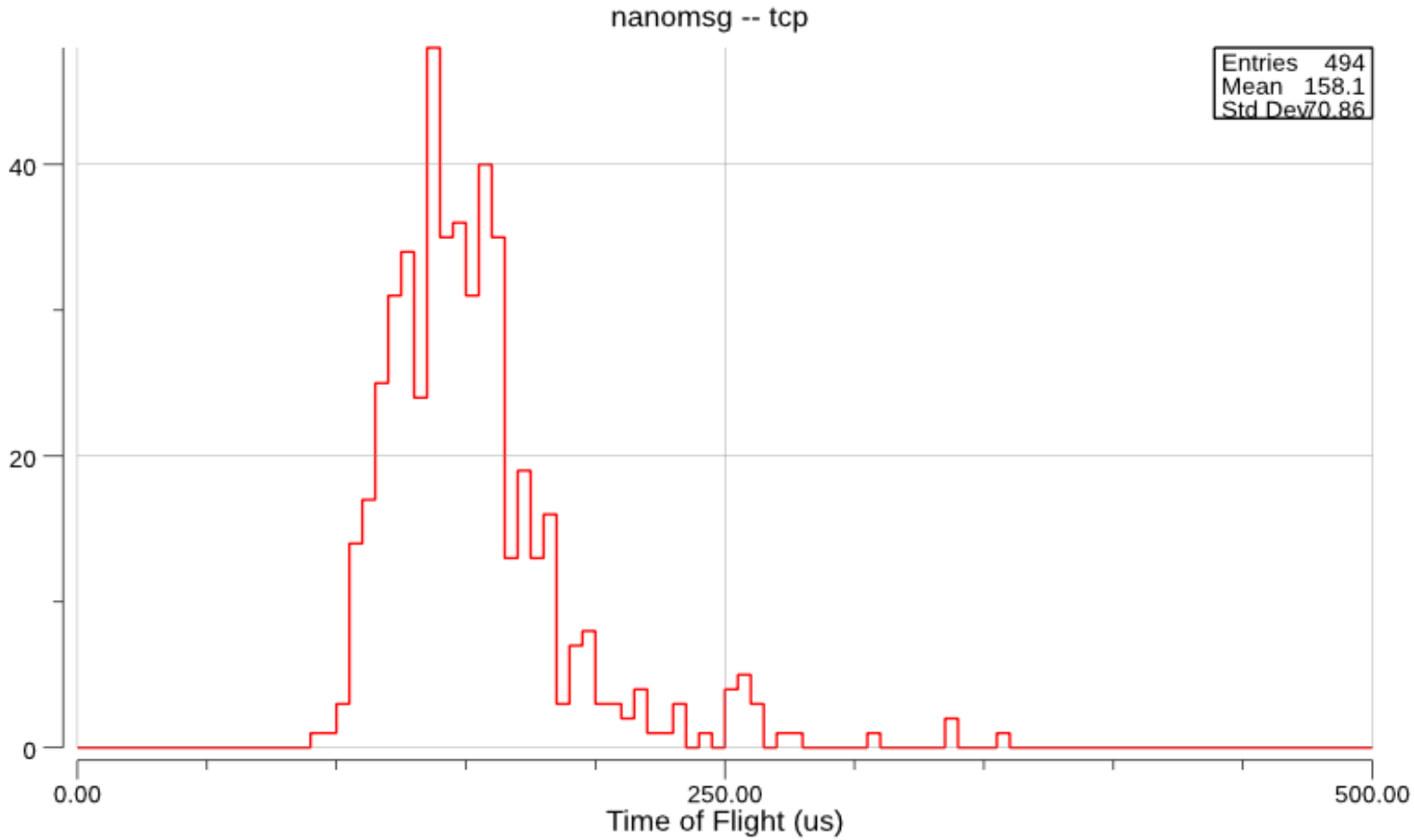
Launch

2018-06-14 19:16:01.7: "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 19:17:00.7: "HELLO (modified by processor)"
2018-06-14 19:17:00.8: "HELLO (modified by processor)"
2018-06-14 19:17:00.8:

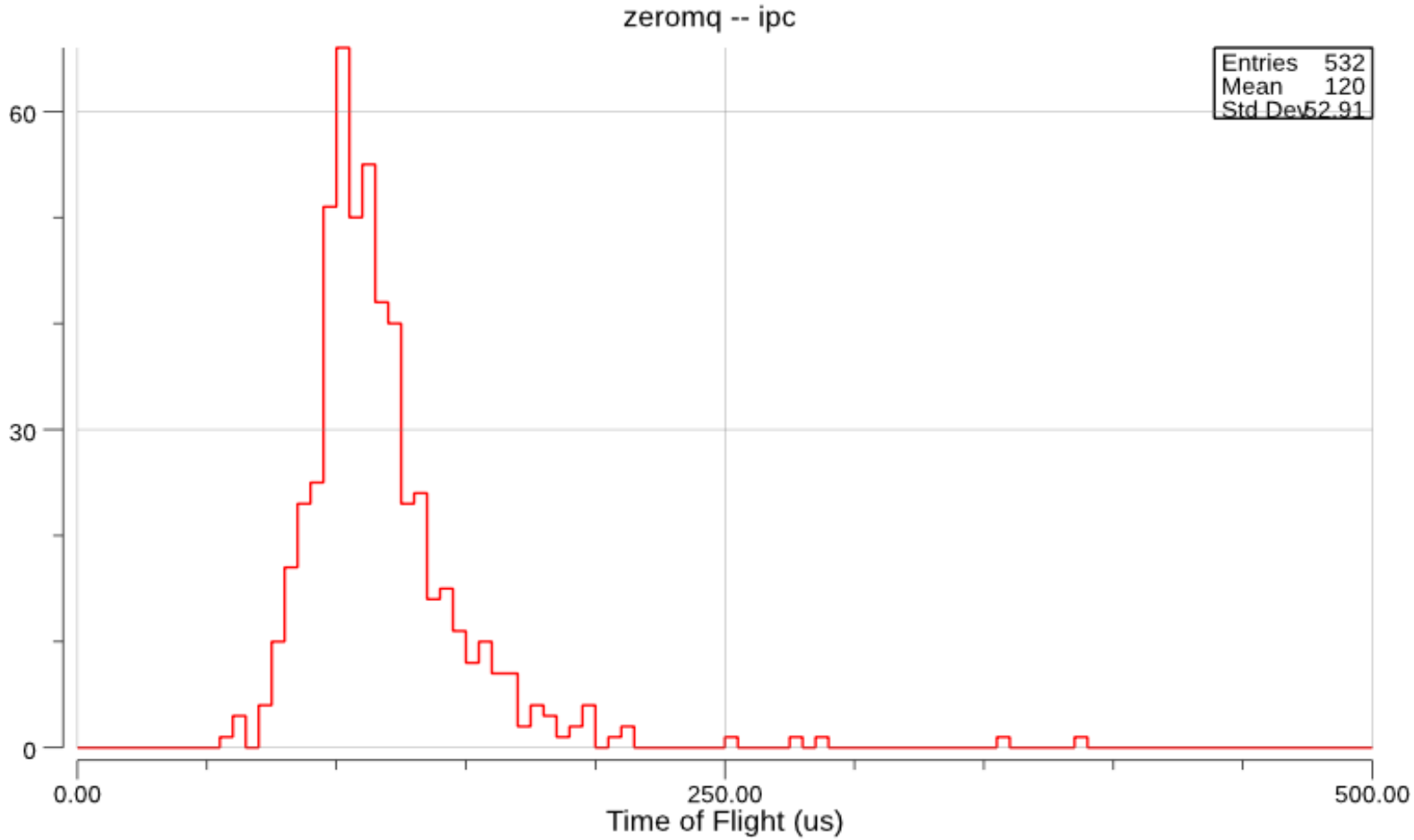




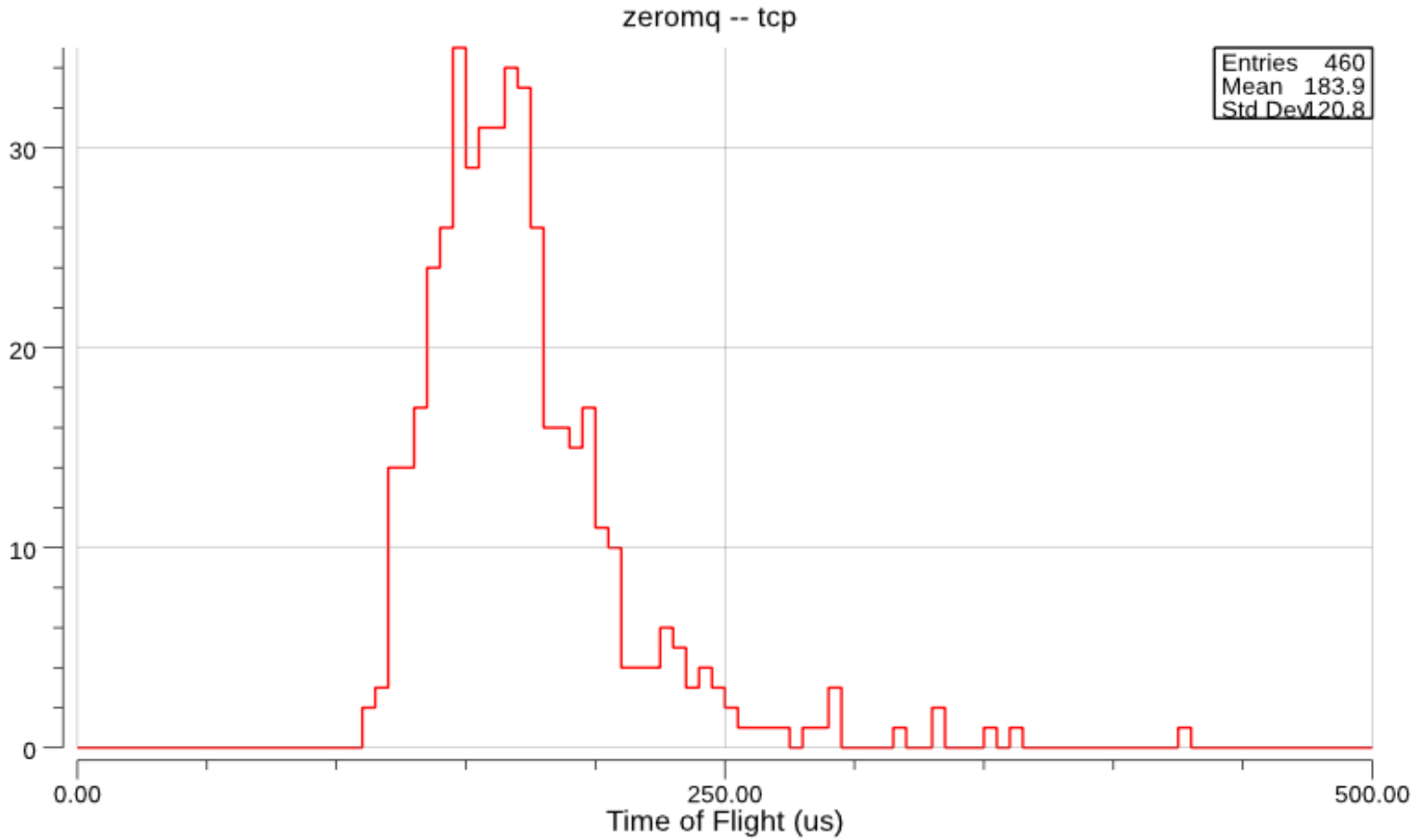

```
$> ./fer-ex-raw-ctl -timeout=20s -transport=nanomsg -protocol=ipc
real=20.06 user=23.77 sys=17.52 CPU=205% MaxRSS=18808 I/O=0/176
```




```
$> ./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
```