

# Reactor Netty

Stephane Maldini

0.8.20.RELEASE

# Table of Contents

Asynchronous TCP, UDP and HTTP .....	1
Overview .....	2
Channels .....	4
Channel Handlers .....	4
Specifications .....	4
Client Specification .....	5
Server Specification .....	5
Backpressure .....	5
TCP 101 .....	6
Start and Stop .....	6
Writing Data .....	6
Flushing Strategies .....	6
Consuming Data .....	6
Backpressure Strategies .....	6
Closing the Channel .....	6
HTTP 101 .....	7
Start and Stop .....	7
Routing HTTP .....	7
Routing WebSocket .....	7
Writing Data .....	7
Flushing Strategies .....	7
Consuming Data .....	7
Backpressure Strategies .....	7
Closing the Channel .....	7

# Asynchronous TCP, UDP and HTTP

Nothing travels faster than the speed of light, with the possible exception of bad news, which obeys its own special laws.

— Douglas Noel Adams, Mostly Harmless (1992)

*Head first with a Java 8 example of some Net work*

```
import reactor.ipc.netty.tcp.TcpServer;
import reactor.ipc.netty.tcp.TcpClient;

//...

CountDownLatch latch = new CountDownLatch(10);

TcpServer server = TcpServer.create(port);
TcpClient client = TcpClient.create("localhost", port);

final JsonCodec<Pojo, Pojo> codec = new JsonCodec<Pojo, Pojo>(Pojo.class);

//the client/server are prepared
server.start( input ->

    //for each connection echo any incoming data

    //return the write confirm publisher from send
    // >>> close when the write confirm completed

    input.send(

        //read incoming data
        input
            .decode(codec) //transform Buffer into Pojo
            .log("serve")
            .map(codec)    //transform Pojo into Buffer

        , 5) //auto-flush every 5 elements
    ).await();

client.start( input -> {

    //read 10 replies and close
    input
        .take(10)
        .decode(codec)
        .log("receive")
        .subscribe( data -> latch.countDown() );

    //write data
```

```
input.send(
    Flux.range(1, 10)
        .map( it -> new Pojo("test" + it) )
        .log("send")
        .map(codec)
    ).subscribe();

//keep-alive, until 10 data have been read
return Mono.never();

}).await();

latch.await(10, TimeUnit.SECONDS);

client.shutdown().await();
server.shutdown().await();
```

# Overview

## How is Reactor Net module working ?

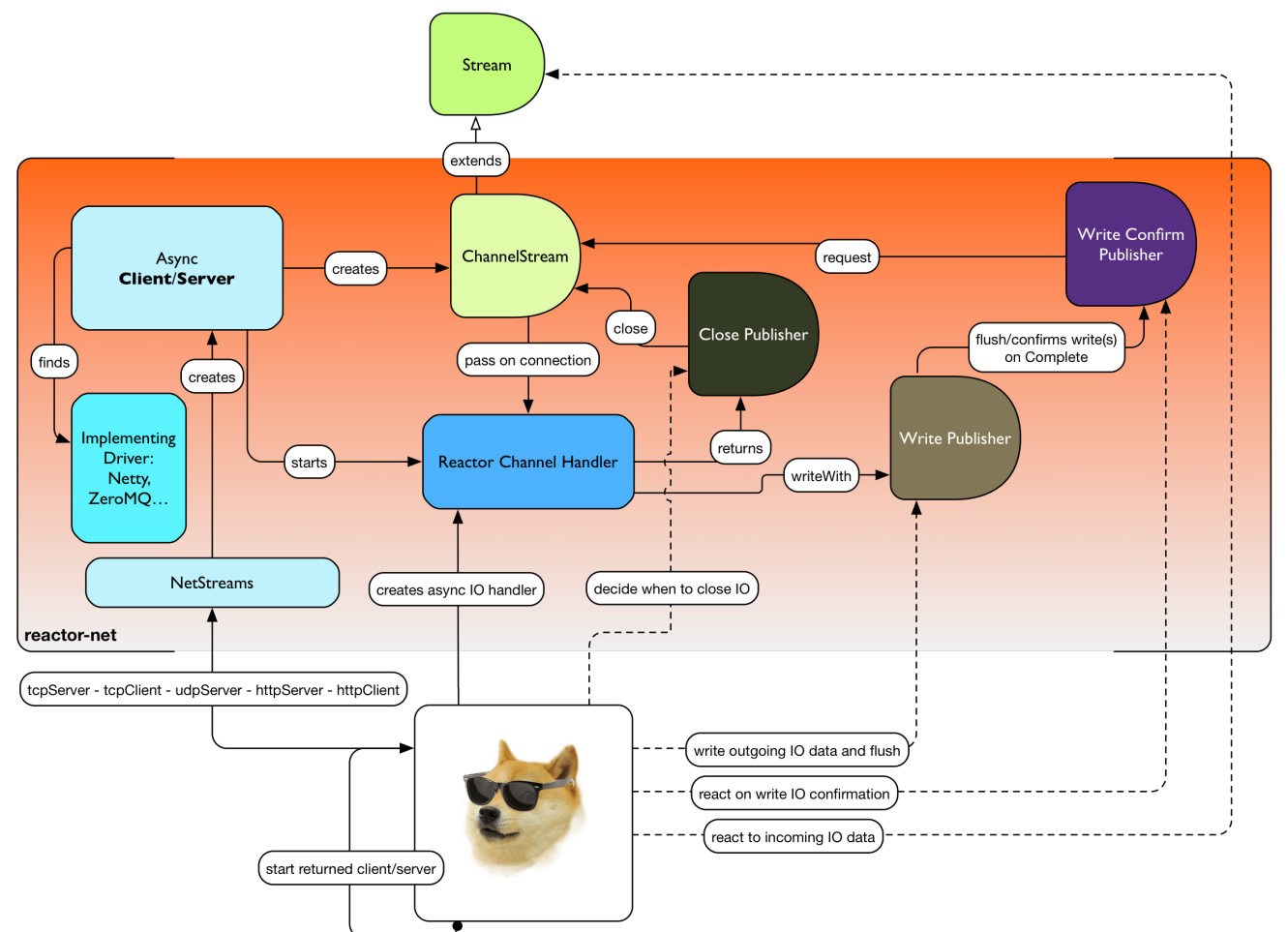
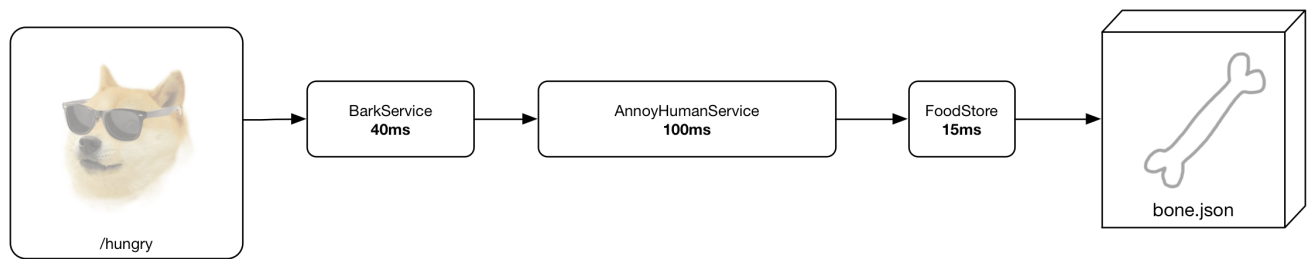


Figure 1. How Doge can use Reactor-Net

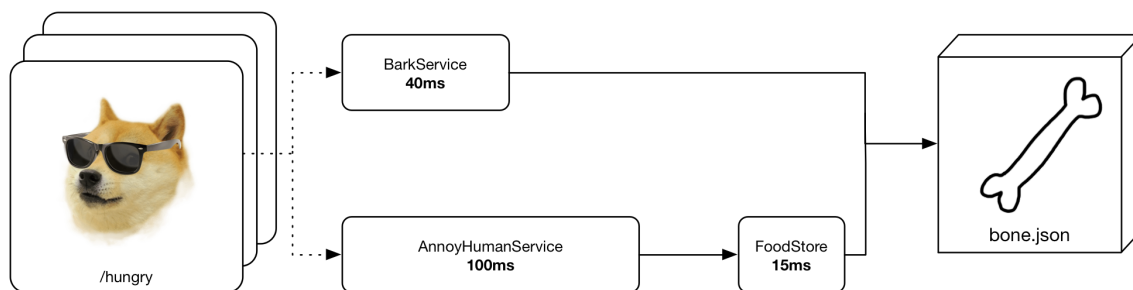
So why should you care about an asynchronous runtime to deal with network operations ? As seen in the [Microservice with Streams](#) section, it is preferred to not block for a service reply. Non-

Blocking write over network will slightly be more costly than blocking ones in terms of resources, however it might be perceived **more responsive** to the producer. Responsiveness all along the request flow impacts various systems and eventually, 1 or N users waiting their turn to push new requests.



Blocking request, send + wait reply on each service and store access  
**155ms Latency, 20ms CPU use**

---



Non Blocking Scatter/Gather request with concurrent service calls, coordination and store access  
**115ms Latency, 30ms CPU use, & Concurrent Access Affinity**

---

Figure 2. Doge trades off CPU for Latency for better responsivity and to leave the service available to his friends

Blocking Read or Write become more like a nightmare for concurrent services use over long-living connections such as TCP or WebSocket. Apart from network routing component which might timeout a too long connection, little can be done with a blocking socket in the application locking the thread on read or write IO methods.

Of course there is always the choice to provide for a pool of threads or any *Async Facade* such as a **Core Processor** to mitigate the blocking read/write contention. The problem is there won't be many of these threads available in a **Reactive** world of non blocking dispatching, so blocking behind 4/8/16 async facades is a limited option. Again the thread pool with a large queue or even many threads won't necessarily solve the situation neither.

*Instead why not invoking callbacks on different IO operations: connection, read, write, close... ?*

**Reactor Net** aims to provide an **Asynchronous IO** runtime that supports **Reactive Streams** backpressure for client or server needs over a range of protocols and drivers. Some drivers will not implement every protocol but at least one, **Netty**, implements all current protocols. At the moment, Reactor Net is **supporting Netty 4.x** and **ZeroMQ** through **jeroMQ 0.3.+** and you must add explicitly one of them in the application classpath.

**Reactor Net** has the following artifacts:

- **ReactorChannel** and its direct implementations **Channel** and **HttpChannel**
  - Represents a direct connection between the application and the remote host
  - Contains non blocking IO write and read operations
  - Reactor drivers will directly expose **Channel** to access the **Stream** functional API for read operations
- **ReactorPeer** and **ReactorChannelHandler** for common network component (client/server) contract
  - Provides for **start** and **shutdown** operations
  - Binds a **ReactorChannelHandler** on **start** to listen to the requesting **Channel**
  - **ReactorChannelHandler** is a function accepting **Channel** requests and returning a **Publisher** for connection close management
- **ReactorClient** for common client contract
  - Extends **ReactorPeer** to provide a *reconnect* friendly start operation
- **NetStreams** and **Spec** to create any client or server
  - Looks like **Streams**, **BiStreams** and other **Reactor Flux** Factories
  - **NetStreams** factories will accept **Function<Spec,Spec>** called **once** on creation to customize the configuration of the network component.
- **HTTP/WS/UDP/TCP** protocol **ReactorPeer** implementations
  - **HttpServer** & **HttpClient** will provide routing extensions
  - **DatagramServer** will provide multicast extensions
  - **TcpServer** & **TcpClient** will provide additional TCP/IP context informations
- **Netty** and **ZeroMQ** drivers



**Reactor Net** implements a model discussed under the [Reactive IPC](#) initiative. As we progress we will align more and eventually depend on the specified artefacts likely over 2016. We give you a chance to experiment as of today with some of the principles and make our best to prepare our users to this next-generation standard.

## Channels

## Channel Handlers

## Specifications

## **Client Specification**

## **Server Specification**

## **Backpressure**

Using Reactor and Reactive Stream standard for flow-control with TCP network peers.

# TCP 101

Using Reactor's TCP support to create high-performance TCP clients and servers.

## Start and Stop

## Writing Data

From a Server perspective

From a Client perspective

## Flushing Strategies

## Consuming Data

From a Server perspective

From a Client perspective

## Backpressure Strategies

## Closing the Channel



# HTTP 101

Using Reactor's HTTP support to create high-performance HTTP clients and servers.

## Start and Stop

## Routing HTTP

SSE

## Routing WebSocket

## Writing Data

Adding Headers and other Metadata

From a Server perspective

From a Client perspective

## Flushing Strategies

## Consuming Data

Reading Headers, URI Params and other Metadata

From a Server perspective

From a Client perspective

## Backpressure Strategies

## Closing the Channel