From sockets to services

Reactive distributed software in Perl 6

Jonathan Worthington | EDUMENT

What things mean

Distributed system

A system where the answer to "is it working" is, "some of it"…

Distributed system

A system where the answer to "is it working" is, "some of it"…

…and the answer to "which bits of it are broken" is, "we can't tell"

Or, more technically…

A system consisting of more than one process (defined as something with independent memory, and that may fail independently), potentially spread over multiple containers, VMs, machines, data centers, countries, planets…

Inherently asynchronous, inherently unreliable

When will data sent between processes arrive?

Inherently asynchronous, inherently unreliable

When will data sent between processes arrive?

Whenever it arrives.

Inherently asynchronous, inherently unreliable

When will data sent between processes arrive?

Whenever it arrives.

(If it arrives.)

Chained request/response

Becomes an anti-pattern more than a level or two deep

First services in the chain spend a long time waiting, and their availability is tied to the services they call (so not autonomous)

Interactive vs. reactive

Interactive programming: ask for something, block until we have it (typified in the iterator pattern)

Reactive programming: subscribe, react whenever things happen (typified in the observer pattern)

Reactive: a better fit for distributed systems

Means we aren't tying ourselves to getting timely responses, just handling things as they happen

Makes it easier to bring the time dimension into our programs

Reactive programming and Perl 6

Interactive and reactive

Let's fill out this table to discover the Perl 6 interactive/reactive types

	One value	Many values
Interactive		
Reactive		

Interactive and reactive

Individual interactive values are obtained just by running a piece of code that produces the value

	One value	Many values
Interactive	`Int`, `Order`, …	
Reactive		

Interactive and reactive

A Seq represents a (perhaps infinite) sequence of values, which are produced on request (blocking)

	One value	Many values
Interactive	`Int`, `Order`, …	`Seq`
Reactive		

Interactive and reactive

A `Promise` represents a single value that will be produced, or fail to be produced, in the future

	One value	Many values
Interactive	`Int`, `Order`, …	`Seq`
Reactive	`Promise`	

Keeping Promises

Anything can be put behind a `Promise`. It can be kept explicitly:

```
> my $p = Promise.new
> $p.status
Planned

> $p.keep(42)
Nil

> $p.status
Kept
> $p.result
42
```

Breaking Promises

Or broken explicitly:

```
> my $p = Promise.new
> $p.break("I just couldn't do it man!")
Nil

> $p.status
Broken
> $p.result
Tried to get the result of a broken Promise
 in block <unit> at <unknown file> line 1
Original exception:
 I just couldn't do it man!
 in block <unit> at <unknown file> line 1
```

Typical `Promise` usage

A `Promise` will typically be kept by an operation that runs concurrently

That may be by code running on another thread, or some kind of asynchronous I/O (running a process, a network connection, etc.)

Interactive and reactive

A `Supply` represents a potentially infinite sequence of values that will be produced asynchronously

	One value	Many values
Interactive	`Int`, `Order`, …	`Seq`
Reactive	`Promise`	`Supply`

Basic publish/subscribe

```
> my $source = Supplier.new
> my $supply = $source.Supply;

> my $t1 = $supply.tap: { say "Got $_" }
> $source.emit("chili")
Got chili

> my $t2 = $supply.map(*.uc).tap: { say "OH WOW $_" }
> $source.emit("beef")
Got beef
OH WOW BEEF

> $t1.close
> $source.emit("noodles")
OH WOW NOODLES
```

Live vs. on-demand

A `Supplier` makes a live `Supply`. We tap into the stream of values at its current point; the past is gone

Most Supplies are on-demand; they start producing values at the point that they are tapped

The interval Supply factory

When the `Supply` returned by `interval` is tapped, it emits values at the specified time interval

```
> my $ticks = Supply.interval(0.5)
> my $tap = $ticks.tap: { say now }; sleep 3; $tap.close;
Instant:1498686115.539947
Instant:1498686116.040888
Instant:1498686116.541719
Instant:1498686117.042902
Instant:1498686117.543302
Instant:1498686118.044487
```

Why put these in the Perl 6 core language?

They provide a standard way to represent asynchronous data

This means that modules producing or processing asynchronous data can be used together

Sockets

Minimal HTTP client

First `connect`, which returns a `Promise`, which we may `await`

```
my $socket = await IO::Socket::Async.connect:
    'moarvm.org', 80;
```

Minimal HTTP client

Then, `print` the HTTP request to the socket:

```
my $socket = await IO::Socket::Async.connect:
    'moarvm.org', 80;
await $socket.print:
    "GET / HTTP/1.0\r\nHost: moarvm.org\r\n\r\n";
```

Minimal HTTP client

Finally, `react` to data `whenever` it arrives by printing it

```
my $socket = await IO::Socket::Async.connect:
    'moarvm.org', 80;
await $socket.print:
    "GET / HTTP/1.0\r\nHost: moarvm.org\r\n\r\n";
react {
    whenever $socket -> $chars {
        print $chars;
    }
}
```

Minimal "HTTP server"

React on incoming connections

```
react {
    whenever IO::Socket::Async.listen('0.0.0.0', 8080)
      -> $conn {

    }
}
```

Minimal "HTTP server"

Wait to receive something

```
react {
    whenever IO::Socket::Async.listen('0.0.0.0', 8080)
      -> $conn {
        whenever $conn {

        }
    }
}
```

Minimal "HTTP server"

Send a response and close the socket

```
react {
    whenever IO::Socket::Async.listen('0.0.0.0', 8080)
      -> $conn {
        whenever $conn {
            whenever $conn.print:
                    "HTTP/1.0 200 OK\r\n" ~
                    "Content-type: text/plain\r\n\r\n" ~
                    "Wow a HTTP response!\n"; {
                $conn.close;
            }
        }
    }
}
```

SSL

IO::Socket::Async::SSL

A drop-in replacement for clients (unless you need a custom CA)

```
use IO::Socket::Async::SSL;
my $conn = await IO::Socket::Async::SSL.connect:
    'moarvm.org', 443;

# The rest of the code is the very same
```

IO::Socket::Async::SSL

For server, just need to supply a key and certificate to `listen`:

```
my %ssl-config =
    certificate-file => 'server-crt.pem',
    private-key-file => 'server-key.pem';
my $server = IO::Socket::Async::SSL.listen:
    'localhost', 4433, |%ssl-config;
react {
    whenever $server -> $conn {
        # Same as for IO::Socket::Async here
    }
}
```

SSH

SSH::LibSSH

An asynchronous binding to libssh

Client side only, but can run commands, do single-file SCP, and do both port forwarding and reverse port forwarding

Run an SSH command (1)

Create an SSH session (which does server and client authentication), and open a command channel

```
my $session = await SSH::LibSSH.connect:
    :$host, :$user, :$port, :$private-key-file;
END $session.close;

my $channel = await $session.execute('ls');
END $channel.close;
```

Run an SSH command (2)

Collect/reflect output and exit code

```
my $exit-code;
react {
    whenever $channel.stdout(:enc<utf8>) -> $chars {
        $*OUT.print: $chars;
    }
    whenever $channel.stderr(:enc<utf8>) -> $chars {
        $*ERR.print: $chars;
    }
    whenever $channel.exit -> $code {
        $exit-code = $code;
    }
}
exit $exit-code;
```

Port forwarding

```
react {
    whenever IO::Socket::Async.listen('127.0.0.1',
            $local-port) -> $connection {
        …
    }
}
```

Port forwarding

```
react {
    whenever IO::Socket::Async.listen('127.0.0.1',
            $local-port) -> $connection {
        whenever $session.forward($remote-host,
                $remote-port, '127.0.0.1', $local-port)
                -> $channel {
            …
        }
    }
}
```

Port forwarding

```
react {
    whenever IO::Socket::Async.listen('127.0.0.1',
            $local-port) -> $connection {
        whenever $session.forward($remote-host,
                $remote-port, '127.0.0.1', $local-port)
                -> $channel {
            whenever $connection.Supply(:bin) {
                $channel.write($_);
                LAST $channel.close;
            }
            …
        }
    }
}
```

Port forwarding

```
react {
    whenever IO::Socket::Async.listen('127.0.0.1',
            $local-port) -> $connection {
        whenever $session.forward($remote-host,
                $remote-port, '127.0.0.1', $local-port)
                -> $channel {
            whenever $connection.Supply(:bin) {
                $channel.write($_);
                LAST $channel.close;
            }
            whenever $channel.Supply(:bin) {
                $connection.write($_);
                LAST $connection.close;
            }
        }
    }
}
```

Reactive Pipelines

Doing application protocols (such as HTTP and web sockets) properly is more complex

Want to break the problem down into isolated, re-usable components

Want to be able to add middleware

Need insight into what's happening

A while back, I had the idea for a set of distributed systems libraries that are centered around building up a `Supply` pipeline to provide reactive services

So I dug in, and in the last couple of months have worked on it together with a colleague at Edument

Today, I'm going to share what we've been building

We've called it Cro.

Soon you'll see why.

We've called it Cro.

Soon you'll see why.

And then you'll groan.

TCP

ROT13 TCP service

Let's build a TCP service that will apply the ROT13 algorithm to everything it receives, and then send the result back to the client

Pull in what we need

The Cro component model and pipeline builder, and the Cro TCP components

```
use Cro;
use Cro::TCP;
```

Create a transform

```
class Rot13 does Cro::Transform {
    method consumes() { Cro::TCP::Message }
    method produces() { Cro::TCP::Message }
    method transformer(Supply $messages --> Supply) {
        …
    }
}
```

Create a transform

```
class Rot13 does Cro::Transform {
    method consumes() { Cro::TCP::Message }
    method produces() { Cro::TCP::Message }
    method transformer(Supply $messages --> Supply) {
        supply {
            whenever $messages {
                …
            }
        }
    }
}
```

Create a transform

```
class Rot13 does Cro::Transform {
    method consumes() { Cro::TCP::Message }
    method produces() { Cro::TCP::Message }
    method transformer(Supply $messages --> Supply) {
        supply {
            whenever $messages {
                emit Cro::TCP::Message.new: data =>
                    .data.decode('latin-1')
                    .trans('a..mn..z' => 'n..za..m', :ii)
                    .encode('latin-1')
            }
        }
    }
}
```

Compose a service…

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
```

...and run it

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
$rot13.start;
react whenever signal(SIGINT) { $rot13.stop; done }
```

...and run it

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
$rot13.start;
react whenever signal(SIGINT) { $rot13.stop; done }
```

my Cro::Service

`my Cro::Service`

Hmmm…."microservice"!

But wait…

Where is the connection management happening?

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
```

Connection manager insertion

`Cro::TCP::Listener` produces a `Cro::TCP::ServerConnection`

Our transform consumes a `Cro::TCP::Message`

Connection manager insertion

When the pipeline composer spots this kind of mismatch, it takes the second part of the pipeline, instantiates a connection manager component with it, and composes that into the pipeline

But also…

Where is the pipeline "sink" that sends back the response?

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
```

Cro::Replyable

A Cro::TCP::ServerConnection does the Cro::Replyable role, and provides a Cro::Sink that sends the replies

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
```

CRO_TRACE=1

Set this environment variable to get a trace of the pipeline

```
my Cro::Service $rot13 = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Rot13;
```

<demo>

HTTP

HTTP applications in Cro

Just a `Cro::Transform` consuming `Cro::HTTP::Request` and producing `Cro::HTTP::Response`

A HTTP transform

```
use Cro::HTTP::Request;
use Cro::HTTP::Response;
class MyApp does Cro::Transform {
    method consumes() { Cro::HTTP::Request }
    method produces() { Cro::HTTP::Response }
    method transformer(Supply $reqs) {
        …
    }
}
```

A HTTP transform

```
use Cro::HTTP::Request;
use Cro::HTTP::Response;
class MyApp does Cro::Transform {
    method consumes() { Cro::HTTP::Request }
    method produces() { Cro::HTTP::Response }
    method transformer(Supply $reqs) {
        supply whenever $reqs -> $request {
            …
        }
    }
}
```

A HTTP transform

```
use Cro::HTTP::Request;
use Cro::HTTP::Response;
class MyApp does Cro::Transform {
    method consumes() { Cro::HTTP::Request }
    method produces() { Cro::HTTP::Response }
    method transformer(Supply $reqs) {
        supply whenever $reqs -> $request {
            given Cro::HTTP::Response.new(
                    :$request, :200status) {
                .append-header('Content-type',
                        'text/plain');
                .set-body("Hello from Cro\n");
                .emit;
            }
        }
    }
}
```

Compose it into a service, run it until SIGINT

```
use Cro::TCP;
use Cro::HTTP::RequestParser;
use Cro::HTTP::ResponseSerializer;

my Cro::Service $http-hello = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Cro::HTTP::RequestParser.new,
    MyApp,
    Cro::HTTP::ResponseSerializer.new;
$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

Persistent connections

HTTP/1.1 persistent connections (many requests over one connection) are automatically handled

The `Supply` of requests just emits each request on that connection

Reactive middleware

Want a logger? Pop it in the pipeline!

```
use Cro::TCP;
use Cro::HTTP::RequestParser;
use Cro::HTTP::ResponseSerializer;
use Cro::HTTP::Log::File;

my Cro::Service $http-hello = Cro.compose:
    Cro::TCP::Listener.new(:host<0.0.0.0>, :port<10000>),
    Cro::HTTP::RequestParser.new,
    MyApp,
    Cro::HTTP::Log::File.new,
    Cro::HTTP::ResponseSerializer.new;
$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

HTTPS?

Swap out the listener for an SSL one

```
use Cro::SSL;

my %ssl-config = :host<0.0.0.0>, :port<10000>,
    private-key-file => 'server-key.pem',
    certificate-file => 'server-crt.pem';
my Cro::Service $http-hello = Cro.compose:
    Cro::SSL::Listener.new(|%ssl-config),
    Cro::HTTP::RequestParser.new,
    MyApp,
    Cro::HTTP::Log::File.new,
    Cro::HTTP::ResponseSerializer.new;
$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

But…

Isn't this a lot of boilerplate for every HTTP service?

Cro::HTTP::Server

Removes the pipeline boilerplate

```
use Cro::HTTP::Server;
use Cro::HTTP::Log::File;

my Cro::Service $http-hello = Cro::HTTP::Server.new:
    :host<0.0.0.0>, :port<10000>,
    :ssl{
        private-key-file => 'server-key.pem',
        certificate-file => 'server-crt.pem';
    },
    application => MyApp,
    after => Cro::HTTP::Log::File.new;
$http-hello.start;
react whenever signal(SIGINT) { $http-hello.stop; done; }
```

And uh…

Isn't there some kind of nicer way to write my application?

Cro::HTTP::Router

```
my $application = route {
    get -> {
        content 'text/plain', "Hello from Cro\n";
    }
}
```

Cro::HTTP::Router

Uses Perl 6 signatures to specify how to perform the routing

```
my $app = route {
    # GET /catalogue/products/42
    get -> 'catalogue', 'products', Int $id {
        ...
    }

    # GET /catalogue/search/saussages
    get -> 'catalogue', 'search', $term {
        ...
    }
}
```

Cro::HTTP::Router

Of course, can use subset types to perform stronger validation

```
my $app = route {
    my subset UUIDv4 of Str where /^
        <[0..9a..f]> ** 12
        4 <[0..9a..f]> ** 3
        <[89ab]> <[0..9a..f]> ** 15
        $/;

    get -> 'user-log', UUIDv4 $id {
        ...
    }
}
```

Cro::HTTP::Router

Optional parameters (with defaults if needed) will work out too

```
my $app = route {
    # GET /products/by-tag
    # GET /products/by-tag/sparkly
    get -> 'products', 'by-tag', $tag-name? {
        ...
    }
}
```

Cro::HTTP::Router

Slurpy parameters are perhaps most useful for serving up static content

```
my $app = route {
    get -> 'css', *@path {
        static 'static-content/css', @path;
    }

    get -> 'js', *@path {
        static 'static-content/js', @path;
    }
}
```

Cro::HTTP::Router

Named parameters access the query string (can use subset types here too)

```
my $app = route {
    get -> 'search', :$term! {
        ...
    }

    get -> 'category', $category-name, :$min-price,
                                       :$max-price {
        ...
    }
}
```

Cro::HTTP::Router

Produces correct HTTP error codes:

404 for route not matching

405 for method not matching

400 for query string not matching

Cro::HTTP::Router

Pluggable body parsers; built-in ones for url-encoded, multi-part, JSON

```
post -> 'log' {
    request-body
        -> (:$level where 'error', :$message!) {
            # Process errors specially
        },
        -> (:$level!, :$message!) {
            # Process other levels
        };
}
```

Cro::HTTP::Router

And a whole bunch more features I don't have time to cover today

Including support for producing various kinds of HTTP response, and pluggable body serializers also

HTTP/2

Sorry to be a bit anti-climactic, but…

…the previous example already was HTTP/2 enabled. ☺

`Cro::HTTP::Server` enables it by default for HTTPS, using ALPN to negotiate its use

Clients that don't support it just get HTTP/1.1 as usual

Actual slight anti-climax: we didn't get push promises in place yet

Low-level plumbing is there; will work out the high level API in the coming month or so

Web Sockets

Yup, we do those too…

They work out rather nicely with Perl 6's reactive features

Integrated into the HTTP router

Super-simple chat backend

```
my $chat = Supplier.new;

get -> 'chat' {
    web-socket -> $incoming, $close {
        supply {
            whenever $incoming -> $message {
                $chat.emit(await $message.body-text);
            }
            whenever $chat -> $text {
                emit $text;
            }
            whenever $close {
                $chat.emit("A user left the chat");
            }
        }
    }
}
```

Clients

Also build around the Cro reactive pipeline concept

Have a `Cro::Connector` at the center (for a client, the network is in the middle of the pipeline rather than at either end)

Cro::HTTP::Client

Persistent connections

HTTP and HTTPS

Uses HTTP/2 if agreed with server

Pluggable body parsers/serializers

Supports middleware

Optional cookie jar

`Cro::WebSocket::Client`

So you can use web sockets for communication between services

Or any other situation when you want to connect to a web socket, really

ZeroMQ

Cro::ZeroMQ

A module providing support for building Cro pipelines using ZeroMQ sockets

Rather new, but looks promising for work distribution, pub/sub, etc.

Tooling

`cro stub`

Stubs a basic service, to provide a starting point

Try to establish some decent practices (such as taking host/port from the environment)

`cro run`

Runs one or more services, assigning them non-colliding ports

Watches for changes to the services, and automatically restarts them

Uses a .cro.yml (not for deployment)

In closing...

We're releasing Cro today, as an early BETA; you'll find it shortly at:

https://github.com/croservices

Along with a site at:

http://mi.cro.services/

The asynchronous programming support in Perl 6 makes it an interesting option for building distributed systems

We hope that Cro will be a useful contribution towards enabling this

Questions?

And in case you missed it:
http://cro.services/