FastTS: Étude d'un outil de monitoring et routeur de métriques en Elixir

Janvier 2016

Mickaël Rémond - @mickael

Introduction

Le but de cette présentation est de démontrer certaines fonctionnalités d'Elixir à travers un projet de développement complet.

Les sujets abordés s'intéresseront:

- À la création de processus légers en Elixir.
- À l'architecture d'une application OTP, à base de worker et de superviseur.
- Aux macros Elixir, permettant de créer un "Domain Specific Language" permettant de configurer le projet élégamment.
- Au packaging avec exrm et au déploiement d'une application Elixir avec Docker.

Outil de monitoring / routeur de métrique

Ce projet s'inspire de **Riemann**

"Riemann aggregates events from your servers and applications with a powerful stream processing language. Send an email for every exception in your app. Track the latency distribution of your web app. See the top processes on any host, by memory and CPU. Combine statistics from every Riak node in your cluster and forward to Graphite. Track user activity from second to second."

Riemann est configurable par du code pour permettre aux sysadmins de configurer et centraliser de manière très flexible le traitement de leurs métriques et alertes.

Pourquoi développer un outil de traitement de métrique en Elixir ?

Les fonctionnalités d'Elixir sont excellentes pour du traitement de flux de "time series":

- Processus légers permettant de paralléliser le traitement des règles.
- Supervision intégrée pour gérer les erreurs.
- Capacité de créer des DSL permettant de configurer son système de manière plus intuitive.

Domain Specific Language: Avantage Elixir

Configuration de Riemann en Clojure:

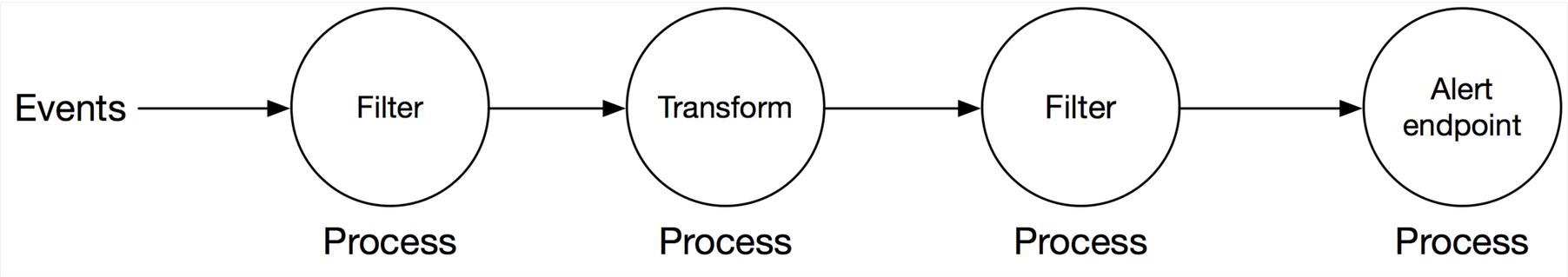
```
(tcp-server)

(let [client (tcp-client)
      index (update (index))]

  (streams
    (with {:metric_f 1 :host nil :service "events/sec"}
      (rate 5 index))

    (where (service #"^per")
      (percentiles 5 [0 0.5 0.95 0.99 1]
        index))

    index
))
```

Configuration possible pour FastTS avec Elixir:

```
defmodule HelloFast.Router do
  use FastTS.Router

  @mail %{to: "mremond@test.com"}

  pipeline "Basic pipeline" do
    under(12)
    mail
  end

  pipeline "Second pipeline" do
    rate(5)
    stdout
  end

end
```

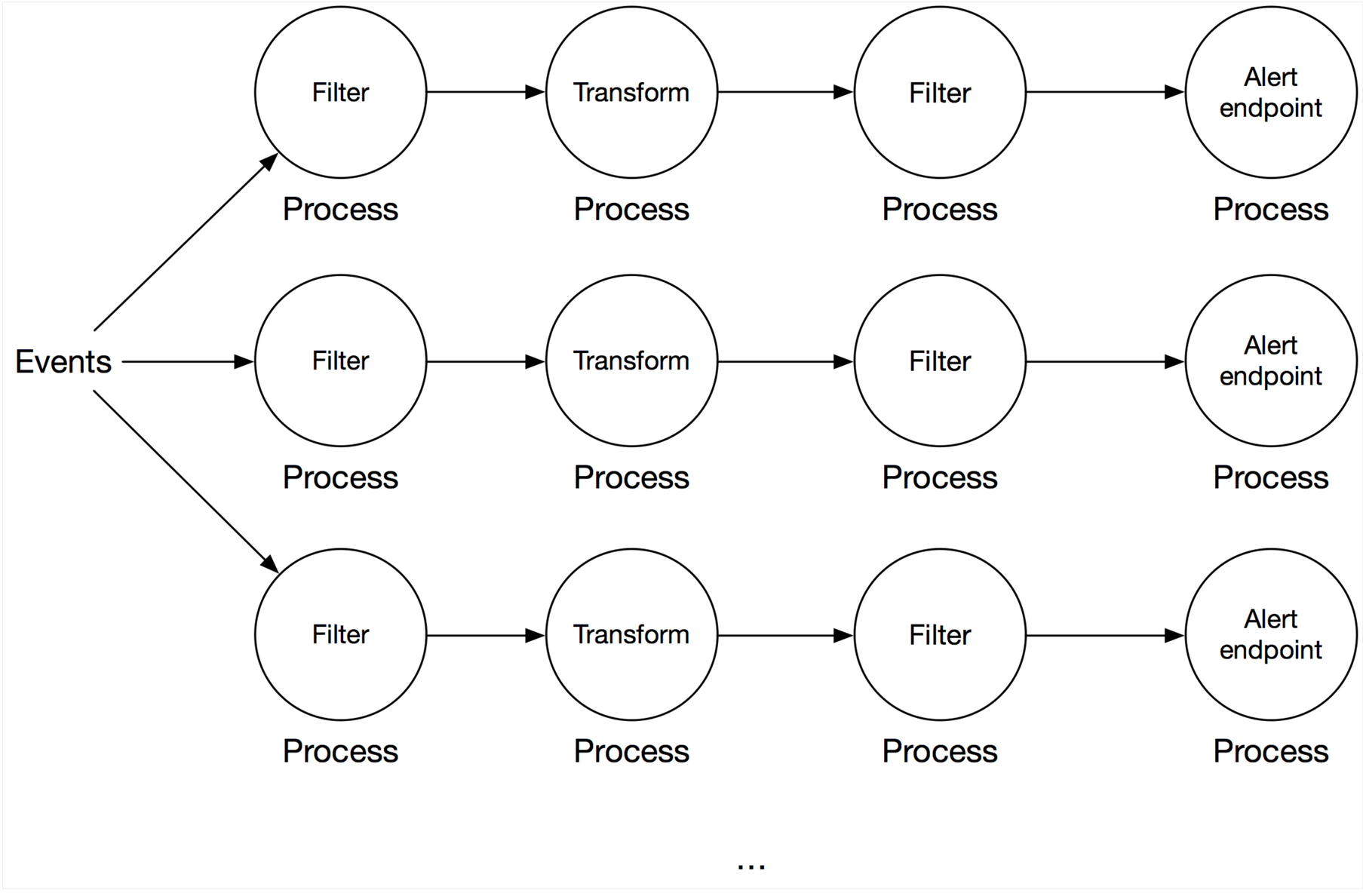
Elixir et les processus légers

Elixir permet de concevoir le traitement des streams comme un ensemble de pipelines dans lequel chaque étape est un process.

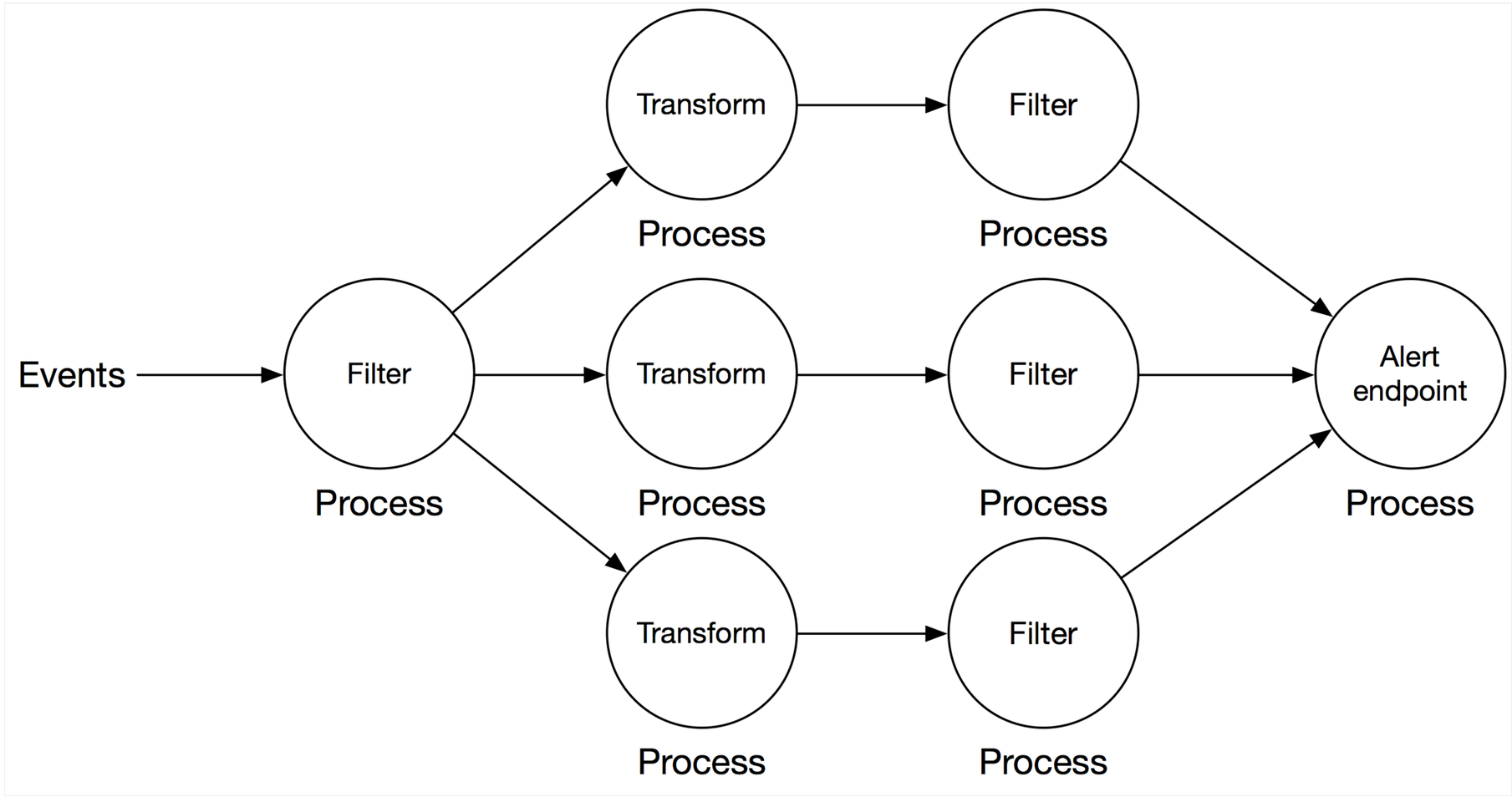


Cela signifie qu'en Elixir:

- Tous les pipelines sont exécutés en parallèle.



- Au sein d'un même pipeline, il sera possible de définir certains traitements comme pouvant être parallèles.



FastTS doit, par construction, exploiter l'ensemble des resources de traitement du serveur.

Le pipeline FastTS en Elixir

Les principes:

- Le coeur du pipeline démarre un process par pipeline.
- Le pipeline lui même est identifié par un identifiant de process. Lorsqu'un événement est reçu, le serveur l'envoi à chaque PID de pipeline.
- Chaque étape est exécutée en recevant un événement et en retournant un événement.
- Chaque étape du pipeline (process) conserve en mémoire l'identifiant du prochain process. Une fois le traitement terminé, il envoie l'événement modifié au process suivant dans le pipeline.
- La fonction de traitement associée à chaque étape peut transformer, dropper ou retarder l'événement.
- La fonction de traitement associée à chaque étape peut être avec ou sans état. L'état permet par exemple l'aggrégation d'événement avant le broadcast à l'étape suivante.

Le code pipeline.ex:

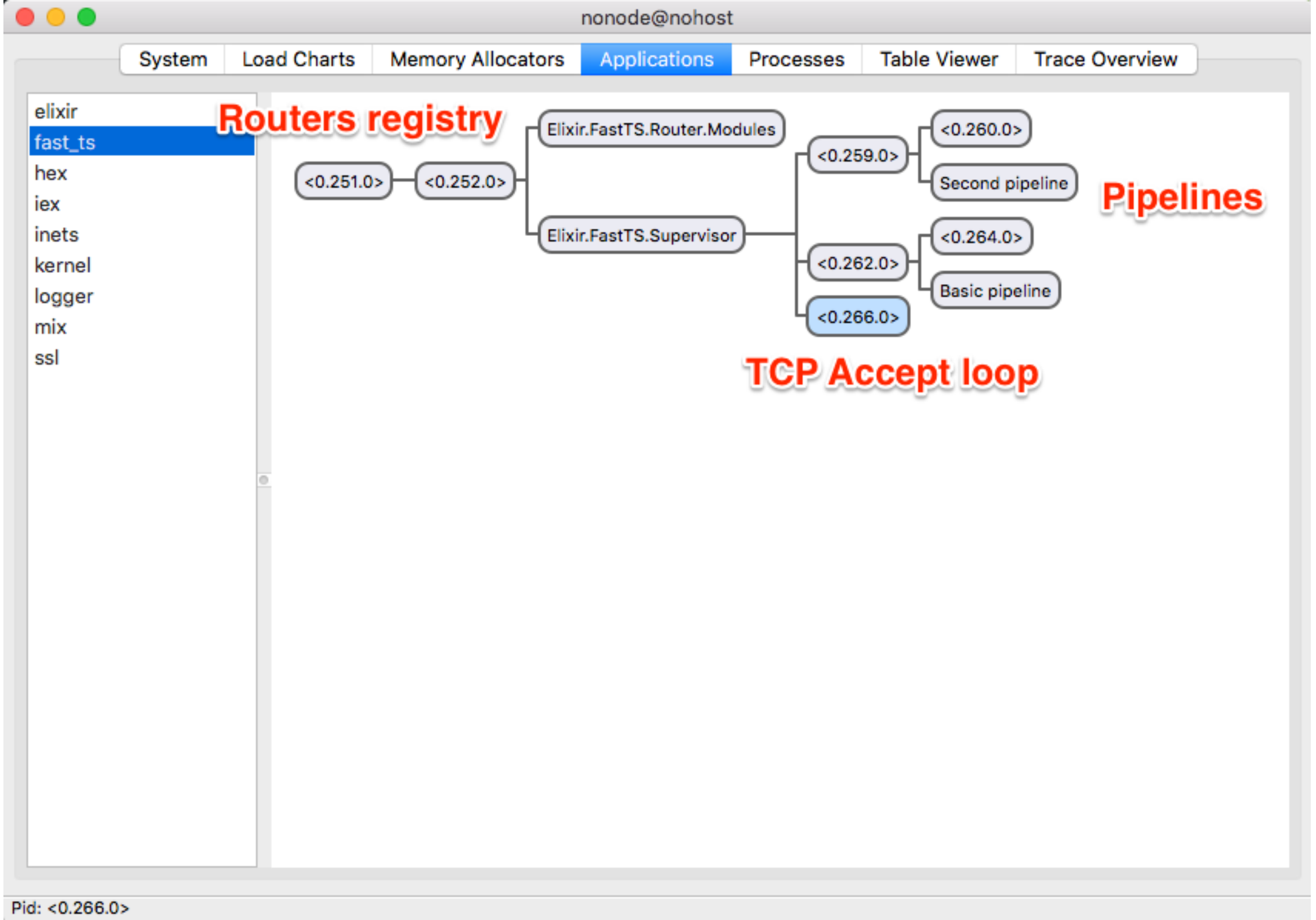
```
defmodule FastTS.Stream.Pipeline do
  use GenServer

  @type pipeline :: list({:stateful, fun})
  @spec start_link(name :: string, pipeline :: pipeline) :: :ignore | {:error,
any} | {:ok, pid}
  def start_link(_name, []), do: :ignore
  def start_link(name, pipeline) do
    GenServer.start_link(__MODULE__, [name, pipeline])
  end

  # Create a chained list of process to pass events through the pipeline
  def init([name, pipeline]) do
    process = process_name(name)
    pipeline
    |> Enum.reverse
    |> Enum.reduce(
      nil,
      fn({:stateful, start_fun}, pid) ->
          spawn_link( fn -> set_loop_state(start_fun, pid) end )
        ({:stateless, add_event_fun}, pid) ->
          spawn_link( fn -> do_loop(add_event_fun, pid) end)
      end)
    |> Process.register(process)

    state = [process, pipeline]
    {:ok, state}
  end

  def next(_result, nil), do: :nothing
  def next(:defer, _pid), do: :nothing
  def next(nil, _pid), do: :nothing
  def next(result, pid), do: send(pid, result)

  # We start one loop per steps in the pipeline. Each pipeline step is a
process.
  def set_loop_state(start_fun, pid) do
    # TODO: for now the name is fix, so it will not work for many pipelines
    table = :ets.new(:pipeline_stage_data, [:public])
    add_event_fun = start_fun.(table, pid)
    do_loop(add_event_fun, pid)
  end

  def do_loop(fun, pid) do
    receive do
      event ->
        event
        |> fun.()
        |> next(pid)
    end
    do_loop(fun, pid)
  end

  # Helper
  defp process_name(name) when is_atom(name), do: name
  defp process_name(name) when is_binary(name), do: String.to_atom(name)

end
```

L'architecture OTP de l'application

L'application doit être robuste pour permettre à du code de traitement tiers de s'exécuter et potentiellement de crasher de temps en temps (bug) sans casser le service lui-même.

Elixir et OTP fournisse des patterns permettant d'organiser des applications robustes, qui résiste et des bugs survenant dans de rares situations.

La commande `:observer.start/0` permet de faire un rendu de l'arbre de supervision de l'application:


nonode@nohost
System
Load Charts
Memory Allocators
Applications
Processes
Table Viewer
Trace Overview
elixir
fast_ts
hex
iex
inets
kernel
logger
mix
ssl
Routers registry
Elixir.FastTS.Router.Modules
<0.251.0>
<0.252.0>
Elixir.FastTS.Supervisor
<0.259.0>
<0.260.0>
Second pipeline
Pipelines
<0.262.0>
<0.264.0>
Basic pipeline
<0.266.0>
TCP Accept loop
Pid: <0.266.0>

A noter:

- Les connections clientes ne sont pas supervisées, car les redémarrer n'a pas d'intérêt (La connexion est perdue)
- Si un process crash dans un pipeline, tout le pipeline est redémarré car sinon la cohérence des données et de la configuration du pipeline n'est pas garantie.
- La registry de routers doit avoir son propre superviseur car le démarrage du superviseur principal de FastTS à besoin de la Registry pour pouvoir démarrer les pipelines.

Autres éléments:

- Nous supportons le protocole Riemann, basée sur ***protocol buffer***.
- Nous utilisons le descriptif Riemann.proto pour générer automatiquement le code de décodage / encodage du protocole.

Les macros Elixir pour améliorer le language de configuration des pipelines

Tout l'intérêt du ***router*** de ***Time Series*** est de pouvoir être totalement programmable pour répondre aux besoins de l'utilisateur.

Elixir offre la possibilité de répondre à ce besoin de configuration en permettant la création d'un langage customisé pour décrire cette configuration.

L'ensemble de la syntaxe d'Elixir repose sur la méta-programmation et les macros. Ces techniques permettent de façonner le language Elixir.

Ces techniques, utilisées dans le language, sont également à disposition des programmeurs et permettent de créer des langages adaptés aux tâches spécifiques. Ce sont les ***Domain Specific Languages*** ou ***DSL***.

Un exemple de macro: Ajouter une syntaxe if

Implémenter une macro custom if (myif) est le 'Hello World' de la macro. L'exercice consiste à supposer que nous devons écrire myif à partir de la structure case.

Nous voulons pouvoir écrire par exemple:

```
My.myif 1==2, do: (IO.puts "1 == 2"), else: (IO.puts "1 != 2")
```

Sans macro et sans possibilité de retarder l'exécution des blocs do et else, ce n'est pas possible.

Les macros permettent de retraiter des blocs de code et de les injecter dans un autre contexte grâce aux fonctions quote et unquote.

Voici le fichier myif.exs:

```
defmodule My do
  defmacro if(condition, clauses) do
    do_clause  = Keyword.get(clauses, :do, nil)
    else_clause = Keyword.get(clauses, :else, nil)
    quote do
      case unquote(condition) do
        val when val in [false, nil] -> unquote(else_clause)
        _                            -> unquote(do_clause)
      end
    end
  end
end

defmodule Test do
  require My
  My.if 1 == 2 do
    IO.puts "1 == 2"
  else
    IO.puts "1 != 2"
  end
end
```

Simplifier la configuration du routeur FastTS grâce aux macros Elixir

Voici le code de départ:

```
defmodule HelloFast.Router do

  alias RiemannProto.Event
  alias FastTS.Stream


  # previous code should generate:

  # TODO: Check if pipeline length is < 0 and do not generate stream
  def streams do
    stream1 = {:localhost, [Stream.rate(5), Stream.stdout]}
    [stream1]
  end

  def stream(event) do
    streams |>
      Enum.each( fn({name, _pipeline}) -> do_stream(name, event) end)
  end

  def do_stream(name = :localhost, event = %Event{host: "localhost"}) do
    send name, event
  end
  # Catch all case: We have no stream matching that event
  def do_stream(_, _), do: :do_nothing

end
```

Ce code:

- Reste complexe, même s'il ne configure qu'un seul ***pipeline*** avec deux étapes: Difficile de voir l'objet du code et les traitements effectués.
- nécessite de connaître les détails de l'implémentation.
- est peu tolérant aux erreurs: Difficile de guider l'utilisateur configurant FastTS.

Étape 1: Définir un module de macro Elixir et l'utiliser

Il suffit d'insérer en début de notre module:

```
defmodule HelloFast.Router do
  use FastTS.Router
...
```

et de définir un module de macro appelé FastTS.Router:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
  end
end
```

Étape 2: Ajouter les alias dans notre macro FastTS.Router

Notre module FastTS.Router devient:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
    quote do
      alias RiemannProto.Event
      alias FastTS.Stream
    end
  end
end
```

Grâce à cela, nous pouvons supprimer les lignes suivantes dans notre script de configuration de router:

```
  alias RiemannProto.Event
  alias FastTS.Stream
```

Elles sont injectées directement dans notre code par la macro __using__/1.

Étape 3: Supporter la construction 'pipeline' dans notre code

Nous pouvons ajouter la construction suivante dans notre module router:

```
defmodule HelloFast.Router do
  use FastTS.Router

  pipeline "Calculate Rate and Broadcast" do
    IO.puts "We are running pipeline CRaB"
  end
...
```

Cette construction doit être remplacée par:

```
...
  def :"Calculate Rate and Broadcast" do
     IO.puts "We are running pipeline CRaB"
  end
...
```

C'est possible en changeant notre module de macro FastTS.Router comme suit:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
    quote do
      alias RiemannProto.Event
      alias FastTS.Stream

      # Needed to be able to inject pipeline macro in the module using
FastTS.Router:
      import unquote(__MODULE__)
    end
  end

  defmacro pipeline(description, do: pipeline_block) do
    pipeline_name = String.to_atom(description)
    quote do
      def unquote(pipeline_name)(), do: unquote(pipeline_block)
    end
  end

end
```

En compilant notre projet, nous pouvons maintenant appeler la fonction générée depuis la console Elixir:

```
iex(1)> HelloFast.Router."Calculate Rate and Broadcast"
We are running pipeline CRaB
:ok
```

Étape 4: Nous utilisons les module attributes pour stocker l'état de compilation et pouvoir lister tous les pipelines

Notre module de routeur contient maintenant deux pipelines:

```
defmodule HelloFast.Router do
  use FastTS.Router

  pipeline "Calculate Rate and Broadcast" do
    IO.puts "We are running pipeline CRaB"
  end

  pipeline "Second pipeline" do
    IO.puts "We can have more than one pipeline"
  end
...
```

Nous modifions notre module de macro FastTS.Router comme suit:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
    quote do
      alias RiemannProto.Event
      alias FastTS.Stream

      # Needed to be able to inject pipeline macro in the module using
FastTS.Router:
      import unquote(__MODULE__)

      # Define pipelines module attribute as accumulator:
      Module.register_attribute __MODULE__, :pipelines, accumulate: true



      # Delay the generation of some method to a last pass to allow getting
the
      # full result of the pipeline
      # accumulation (it will be done in macro __before_compile__/1)
      @before_compile unquote(__MODULE__)
    end
  end

  defmacro __before_compile__(_env) do
    quote do
      def list_pipelines do
        IO.puts "Defined pipelines in the router: #{inspect @pipelines}"
      end
    end
  end

  defmacro pipeline(description, do: pipeline_block) do
    pipeline_name = String.to_atom(description)
    quote do
      @pipelines unquote(pipeline_name)
      def unquote(pipeline_name)(), do: unquote(pipeline_block)
    end
  end
end
```

Nous pouvons alors lister les pipelines comme suit depuis la console Elixir:

```
 iex(1)> HelloFast.Router.list_pipelines
 Defined pipelines in the router: [:"Second pipeline", :"Calculate Rate and
Broadcast"]
 :ok
```

Étape 5: Chaque fonction pipeline retourne chaque appel de fonction comme un élément d'une liste

Nous voulons pouvoir écrire notre pipeline comme:

```
pipeline "Second pipeline" do
  Stream.rate(5)
  Stream.stdout
end
```

et recevoir comme retour de la fontion sous-jacente:

```
[Stream.rate(5), Stream.stdout]
```

Notre macro pipeline devient:

```
defmodule FastTS.Router do
...
  defmacro pipeline(description, do: pipeline_block) do
    pipeline_name = String.to_atom(description)

    # Transform the block call in a list of function calls
    block = case pipeline_block do
              nil ->
                nil
              {:__block__, [], block_sequence} ->
                block_sequence
              single_op ->
                [single_op]
            end
    IO.puts "==== injected block = #{inspect block}"

    case block do
      nil ->
        IO.puts "Ignoring empty pipeline '#{description}'"
      _ ->
        quote do
          @pipelines unquote(pipeline_name)
          def unquote(pipeline_name)(), do: unquote(block)
        end
    end
  end
end
```

Étape 6: Générer la fonction streams, renvoyant l'ensemble des pipelines

C'est une fonction supplémentaire à rajouter à la fin du processus de génération du code:

```
defmodule FastTS.Router do
...
  defmacro __before_compile__(_env) do
    quote do
      def streams do
        Enum.map(@pipelines,
          fn(name) ->
            {name, apply(__MODULE__, name, [])}
          end)
      end
    end
  end
...
```

Étape 7: Générer une fonction dispatchant les événements à tous les streams du router

Il suffit de générer une fonction stream(event) qui appelle notre précédente fonction streams et envoie un message à tous les processes enregistrés avec un nom de pipeline donné:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
 ...
      def stream(event) do
        streams |>
          Enum.each( fn({name, _pipeline}) -> send(name, event) end)
      end

    end
  end
...
```

Étape 8: Import du module Stream pour simplifier les appels aux commandes de stream processing

Dans notre module de macro, nous remplaçons l'alias de FastTS.Stream par un import:

```
defmodule FastTS.Router do
  defmacro __using__(_options) do
    quote do
      alias RiemannProto.Event
      import FastTS.Stream
...
```

Cela nous permet d'utiliser les fonctions de stream dans les pipelines sans utiliser le nom du module:

```
  pipeline "Second pipeline" do
    rate(5)
    stdout
  end
```

Le module de configuration de route final

```
defmodule HelloFast.Router do
  use FastTS.Router

  pipeline "Basic pipeline" do
    under(12)  # We discard event if metric is higher
    stdout
  end

  pipeline "Second pipeline" do
    rate(5)
    stdout
  end
end
```

Le module est maintenant simple et se lit de manière purement déclarative.

FastTS est compatible Docker

FastTS propose directement une chaine de compilation qui:

- Prépare une release OTP (exrm)
- Package cette release dans un container Docker minimaliste (Alpine Linux)

Le Dockerfile est:

```
FROM msaraiva/erlang:18.1

RUN apk --update add erlang-crypto erlang-sasl && rm -rf /var/cache/apk/*

ENV APP_NAME fast_ts
ENV APP_VERSION "0.0.1"
ENV PORT 5555
ENV FTS_ROUTE_DIR /$APP_NAME/routes

RUN mkdir -p /$APP_NAME
ADD rel/$APP_NAME/bin /$APP_NAME/bin
ADD rel/$APP_NAME/lib /$APP_NAME/lib
ADD rel/$APP_NAME/releases/start_erl.data
/$APP_NAME/releases/start_erl.data
ADD rel/$APP_NAME/releases/$APP_VERSION/$APP_NAME.boot
/$APP_NAME/releases/$APP_VERSION/$APP_NAME.boot
ADD rel/$APP_NAME/releases/$APP_VERSION/$APP_NAME.rel
/$APP_NAME/releases/$APP_VERSION/$APP_NAME.rel
ADD rel/$APP_NAME/releases/$APP_VERSION/$APP_NAME.script
/$APP_NAME/releases/$APP_VERSION/$APP_NAME.script
ADD rel/$APP_NAME/releases/$APP_VERSION/$APP_NAME.sh
/$APP_NAME/releases/$APP_VERSION/$APP_NAME.sh
ADD rel/$APP_NAME/releases/$APP_VERSION/start.boot
/$APP_NAME/releases/$APP_VERSION/start.boot
ADD rel/$APP_NAME/releases/$APP_VERSION/sys.config
/$APP_NAME/releases/$APP_VERSION/sys.config
ADD rel/$APP_NAME/releases/$APP_VERSION/vm.args
/$APP_NAME/releases/$APP_VERSION/vm.args

RUN mkdir -p /$APP_NAME/routes
ADD config/routes/route.exs /$APP_NAME/routes/route.exs

EXPOSE $PORT

CMD trap exit TERM; /$APP_NAME/bin/$APP_NAME foreground & wait
```

La séquence pour construire le container est la suivante:

```
mix deps.get
MIX_ENV=prod mix compile
MIX_ENV=prod mix release
docker build -t fast_ts .
```

Pour démarrer le container avec une configuration de route custom:

```
docker run -v "$PWD/config/docker":/opt/routes -e
"FTS_ROUTE_DIR=/opt/routes" --rm -p 5555:5555 fast_ts
```

FastTS: La suite

- Disponible aujourd'hui sur Github: https://github.com/processone/fast_ts
- Open Source: Apache v2

Prochaines étapes:

- Tuner le coeur et le langage.
- Documenter le développement de function de traitement.
- Développer de nouvelles commandes avec la communauté.