N2O

Most Powerful Erlang Web Framework

@5HT

How do I shot Web?

Micro REST

Python Flask

Ruby Sinatra

PHP Silex

Scala Scalatra

Concurrency in Mind

Ruby Celluloid

PHP React

PHP phpDaemon

Java+Scala Play

SPA

Angular Meteor Ember
Chaplin Brunch
D3 Knockout React
Backbone jQuery

Functional DSL

Scala Lift

Erlang Nitrogen

Haskell BlazeHtml

OCaml Ocsigen

F# WebSharper

Clojure

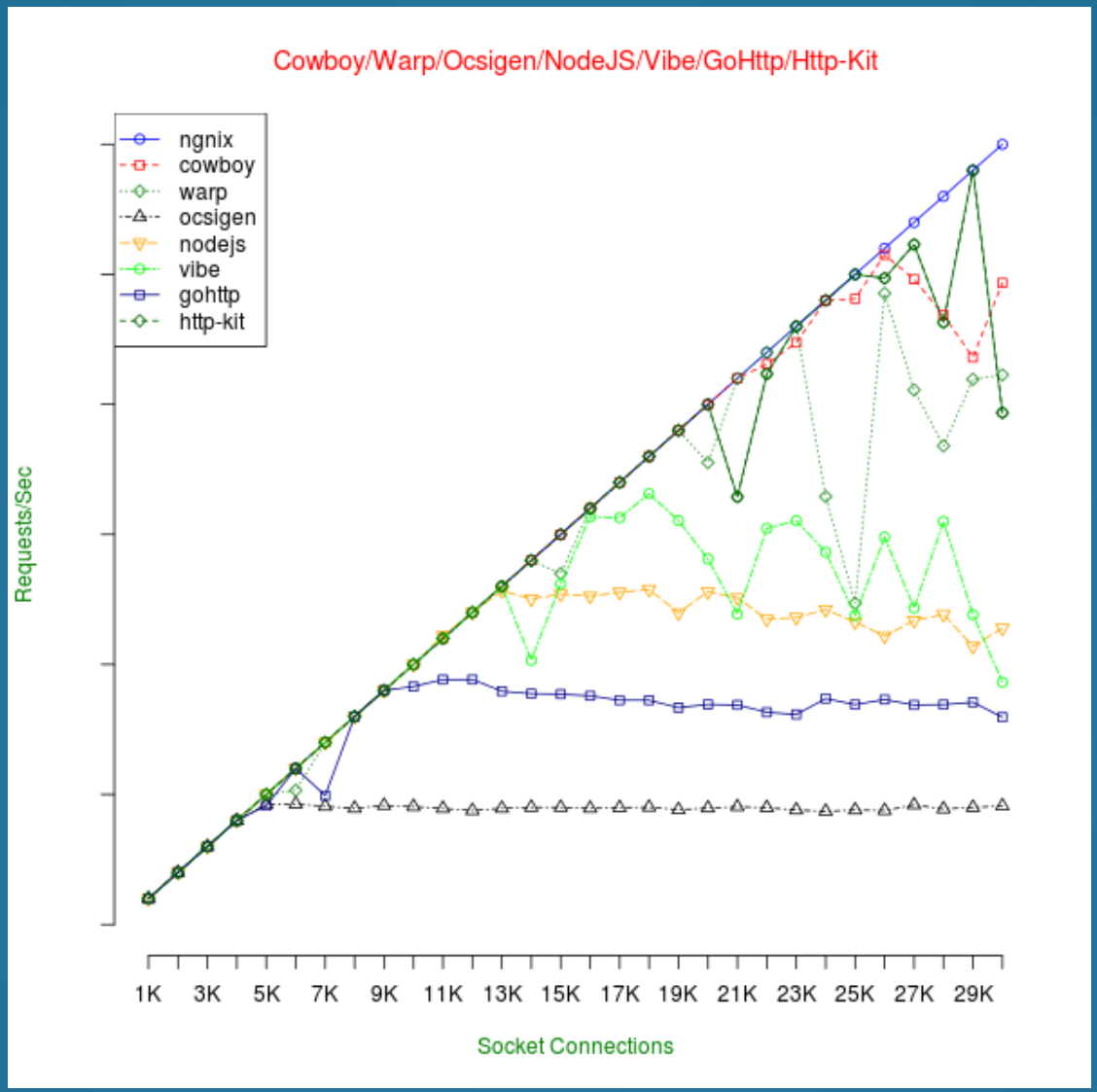
Laser Enlive

Compojure Ring

Hiccup ClojureScript Om

http-kit aleph noir

JVM


Cowboy/Warp/Ocsigen/NodeJS/Vibe/GoHttp/Http-Kit
ngnix
cowboy
warp
ocsigen
nodejs
vibe
gohttp
http-kit
Requests/Sec
1K
3K
5K
7K
9K
11K
13K
15K
17K
19K
21K
23K
25K
27K
29K
Socket Connections

Elixir Weber

```
        def action(_, conn) do
              {:render, [project: "simpleTodo"], []}
        end

def add([body: body], _conn) do
    {:json, [response: "ok"], [{"Content-Type", "application/json"}]}
end

            def wrong(_, _) do
                  {:redirect, "/"}
            end
```

Erlang ChicagoBoss

DTL Engine

Database Connectivity

PubSub

Ruby on Rails like

Nitrogen

N2O ~2000 LOC

One Process per Connection

Binary Page Construction

Zero Bridge

GProc Pub/Sub

WebSockets, KVS DB

Tuned Layers

static and dynamic Routing Path
cleaned Query Parser
Session Cookies stored in ETS
cached DTL Templates
optimized Nitrogen DSL rendering
HTML Elements binaries
JavaScript Actions deferred

Layers Performance

components		TCP conn (K)
PHP5 FCGI	Simple Script <?php ?>	5
Nitrogen	No session, No DSL, DTL	1
N2O	Sessions, DSL, DTL	7
N2O	Sessions, no DSL, DTL	10
N2O	No Sessions, no DSL, DTL	13

On same machine
raw webserver performance
measured with wrk: NGINX -- 60K
Cowboy -- 30K

Measuring Tools

	requests	pages/sec	latency (ms)
wrk	**15K**	**13628.86**	**18.88**
ab	**10K**	**5464.63**	**190**
httperf	**10K**	**3623.50**	**200**
siege	**1K**	**884.51**	**430**

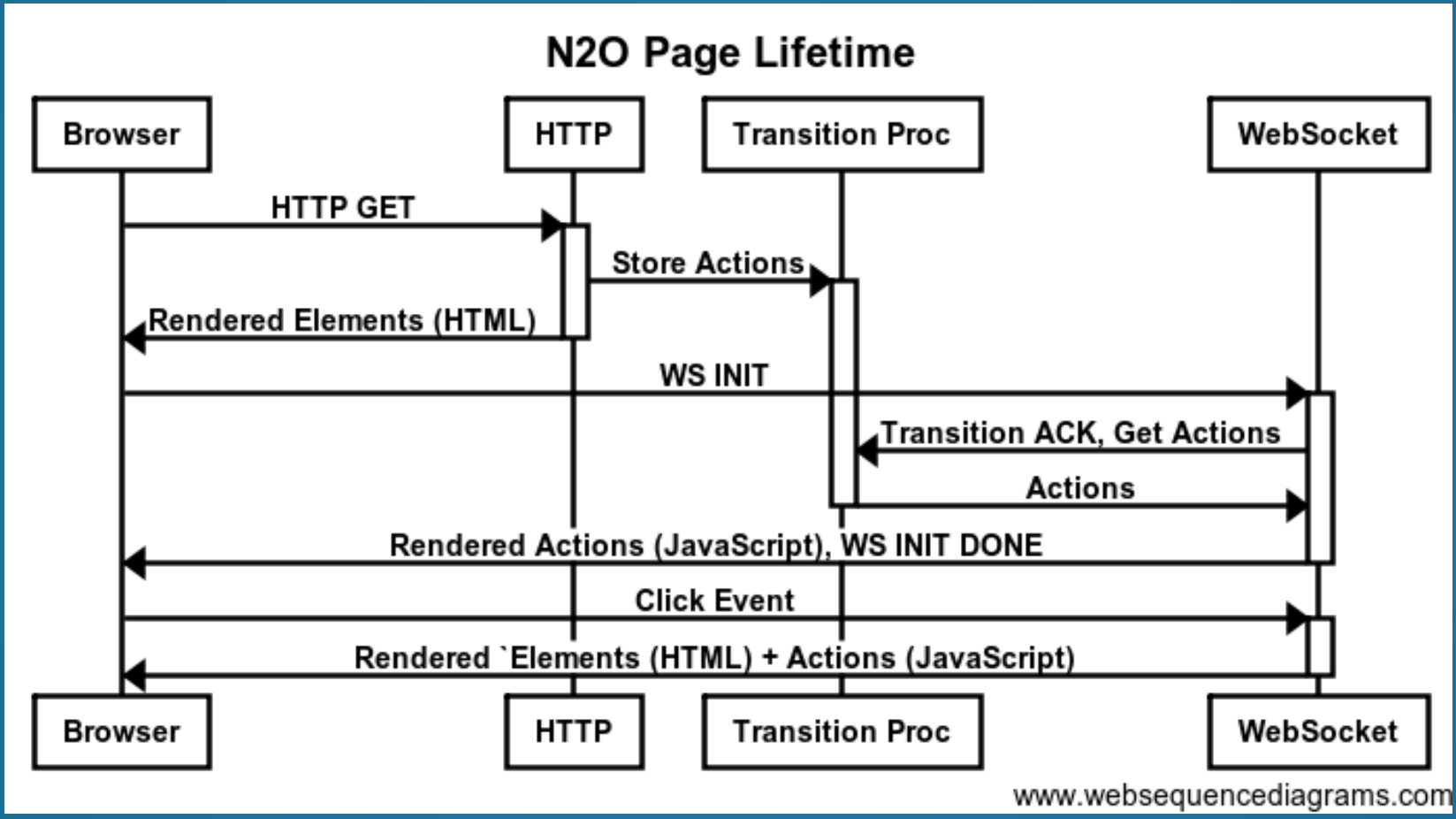
On same machine
raw webserver performance
measured with wrk: NGINX -- 60K
Cowboy -- 30K

Decreasing Latency From 2x to ∞

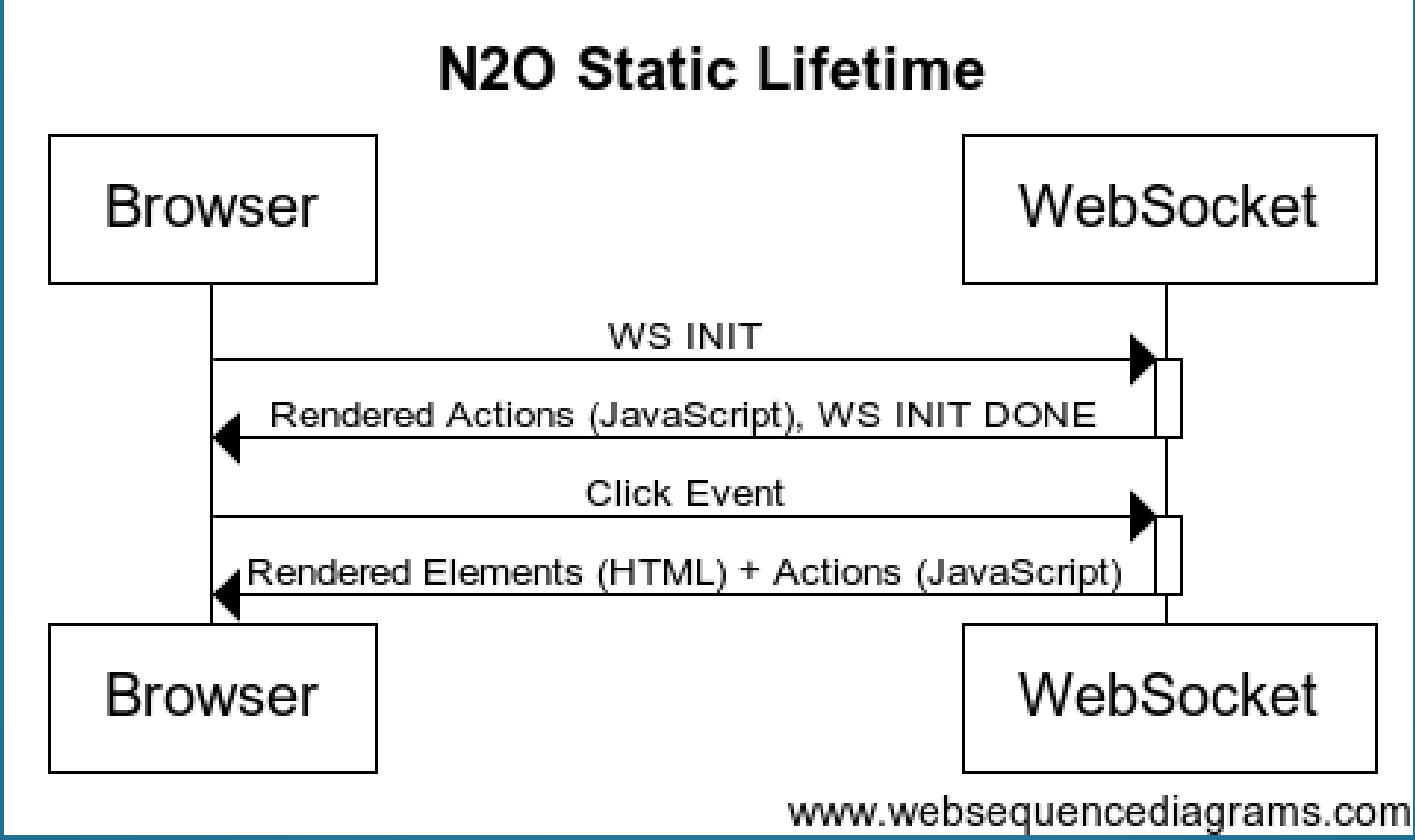
Deliver HTML ASAP
Deferred JavaScript delivery
after WebSocket
connection established

<script> TransitionProcess = '<0.7780.5>' </script>
socket.send(["N2O",TransitionProcess]).

Dynamic HTML cow



Static HTML nginx



SPA and DTL

Render both HTML and JavaScript using DSL :

```
#button { postback = login, source = [ user, pass ] }
```

Or render only JavaScript ..

```
#button { render = script,
          id = ElementId,
          postback = login,
          source = [ user, pass ] }
```

.. when you develop SPA

Render

```
<input value="Login" id= "temp563149" type="button"/>

$( '#temp563149' ).bind( 'click' ,
    function anonymous(event)
        ws.send(Bert.encodebuf({
            source: Bert.binary( 'temp563149' ),
            pickle: Bert.binary('g2gCaINoAmgEZAA...'),
              linked: [ Bert.tuple(Bert.atom( 'user' ),
                                                    utf8.toByteArray($(
'#user' ).val())),
                          Bert.tuple(Bert.atom( 'pass' ),
                                utf8.toByteArray($( '#pass' ).val())) ]
     })); });
```

Elements

Plug your JavaScript controls

```
body() -> [ #textboxlist { },
            #grid { },
            #diagram { },
            #histogram { },
            #monitor { } ].
```

and talk with them using BERT through WebSockets

Actions

Client Updates are sent as Actions

which are evaluated as JavaScript strings

```
render_action(Record) ->
        wf:f("alert(\"~s\");",  [wf:js_escape(Record#alert.text)]).

wf:wire(#alert{text="Hello, World!"}).
```

Element render could create Actions

Action could include rendered Elements

Triggered Action may send Events

JS Compiler 200 LOC

Preserves Semantics

```
-compile({parse_transform,shen}).
M = lists:foldl(fun(X, Acc) -> Acc + X end, 0, [1,2,3,4]),
```

```
$ erlc sample.erl
```

```
m = [1,2,3,4].reduce( function(x,acc) { return acc + x; }, 0);
```

1. Human Readable Output

2. No Runtimes

3. Strong Dynamic Typing

4. Compact Language, Sane Syntax

Synrc Shen, JScala	**[1,2,3,4]**
TypeScript	**[1,2,4]**
CoffeeScript, LiveScript	**[1,2,3]**
Ur, Fay, Roy, Elm, Haxe, F*, WebSharper	**[3,4]**
ClojureScript	**[4]**
OberonJS	**[1,3,4]**
Wisp	**[1,2,4]**

Why Erlang to JS ?

Validation Rules

Business Rules

Prototyping

Static Content Generation

Single Environment

SPA React ERL

```
User = #react{
    render = fun(This) -> #h1{body=value(email,This)} end
        },

CommentList = #react{
    props = [{data,[]}],
    render = fun(This) ->
        Users = lists:map(fun(Item) ->
            User#react{props=Item} end, value(data,This)),
        Users
    end },
```

SPA React JS

```
var user = React.createClass({
        render: function() {
                return React.DOM.h1(null,value('email',this));
        }});

var commentlist = React.createClass({props: {data:[]},
        render: function() {
         var users = value('data',this).map(function(item) {
                return user({props: item});
         });
                return users;
        }});
```

Erlang AST CMPTBL

Brace (Joxa /w Erlang AST)
Elixir, Erlang

Synrc Shen JavaScript Compiler
can handle each Erlang language that
compiles to Erlang AST*

DTL Sample

```
main () ->
    Body = wf : render ( body () ) ,
    [ #dtl { file =  "login" ,
                bindings = [ { body , Body } ] } ].

body () ->
    [  #span { text  = "Login: " },
       #textbox  { id  = user },
       #span  {  text  =  "Pass: "  },
       #password  {  id  = pass },
       #button  {  text  = "Login" ,  postback =  login , source =  [ user ,
pass ] } ].

event ( login ) ->  wf : user ( wf : q ( user ) ), wf : redirect ( ).
```

N2O in Elixir

```
defmodule Sample.Index do

  def  event(:init) do : wf.reg(:room) end
  def  event({:chat,pid}) do pid <- {: message, : wf.q(:message)} end
  def  event(e) do : wf.info('Unknown Event ~p',[e]) end
  def  main() do : dtl.new(file: "index", bindings: [title:  "Elixir N2O", body: body()]) end
  def  body() do { : ok, pid } = : wf.async(fn() -> loop() end)
            [ :    div.new(id: :history),
              :    textbox.new(id: :message),
              :    button.new(id: :send, body:  "Chat", postback: {: chat,pid}, source: [: message]) ] end

  def  loop() do receive do
       {:   message, message} ->   : wf.insert_bottom(:history, [ : span.new(body: message), : br.new() ])
          :   wf.flush(:room)
       unknown -> :   wf.info('Unknown Looper Message ~p',[unknown]) end
     loop() end  end
```

N2O in Joxa

```
(ns lispchat (require erlang joxa-core n2o wf))
(defn+ body() [(n2o/panel   {:id :history})
               (n2o/textbox {:id :message})
               (n2o/button  {:id :send}
                            {:body "Chat"}
                            {:postback :chat}
                            {:source [:message]}) ])
(defn+ main() [(n2o/dtl {:app :n2o_sample}
                        {:file "index"}
                        {:bindings [{:title "Title"} {:body (body)}]})])
(defn+ speak(m) (wf/insert_bottom :history (n2o/span {:body m})))
(defn+ event(body)
   (case body
      (:init (speak ["Hello" (n2o/br)]))
      (:chat (speak [(wf/q :message) (n2o/br)]))))
(defn+ test() (do
               (event :init)
               (wf/info "hello ~p~n" [(main)])))
```

JSON

proplists <-> JSON 243x

Library	Type	Enc	Dec	
jsonx	C NIF	1048	998	
jiffy	C NIF	2652	2234	
yaws2	Erlang	14208	12653	Xen Ready
jsonerl	Erlang	14549	14899	
mochijson2	Erlang	16787	16512	
jsx	Erlang	27188	18333	

JSON PT

proplists <=> #rec{}

```
-module(users).
-behaviour(n2o_rest).
-compile({parse_transform, n2o_rest}).
-include("users.hrl").
-rest_record(user).

from_json(Proplist, #user{}) -> #user{}.
to_json(Record) -> proplist().
```

REST Sample

```
-compile({parse_transform, n2o_rest}).
-include("users.hrl").
-rest_record(user).

init() -> ets:new(users, [public, named_table, {keypos, #user.id}]).
populate(Users) -> ets:insert(users, Users).
exists(Id) -> ets:member(users, wf:to_list(Id)).
get() -> ets:tab2list(users).
get(Id) -> [User] = ets:lookup(users, wf:to_list(Id)), User.
delete(Id) -> ets:delete(users, wf:to_list(Id)).
post(#user{} = User) -> ets:insert(users, User);
post(Data) -> post(from_json(Data, #user{})).
```

KVS REST API

```
POST/PUT kvs:add(#user{}).
  DELETE  kvs:remove(user,2).
PUT/PATCH kvs:put(#user{id=2}).
     GET  kvs:entries(Feed,user).
     GET  kvs:get(user,2).
     GET  kvs:all(user).
```

Events

From JavaScript

Everything is sent as an Event

Postback event/1

Control control_event/2

API api_event/3

Handle events in Erlang...

Events Nature

Events are enveloped by BERT as #ev{}

```
-record ( ev,
          { name :: api_event | control_event | event | atom () ,
            payload ,
            trigger ,
            module :: atom () } ).
```

You can define your own entry points

Postback Events

Create Page Logic with Postback Events event/1

```
main() -> [ #button { postback = login } ].

event(login) -> wf:info("Logged User: ~p", [wf:user()]).
```

Control Events

Handle Data from Controls with Control Events control_event/2

```
render_element( E = #myelement { } ) -> do_render( E ).

control_event(Trigger, Data) ->
    wf:info("Received Data ~p from JavaScript Control ~p",
        [ Data, Trigger ] ).
```

... when you cteate your own Elements

API Events

Wire Erlang and JavaScript with API Events api_event/3

```
main() -> wf:wire( #api { name = notify, tag = api3  } ), [ ].

api_event(Event, Args, Req) ->
        wf:info( "Received from JavaScript: ~p", [ Args ] ).
```

```
document.notify('Hello from JavaScript').
```

Community

Sources

https://github.com/5HT/n2o

HTML and TeX Handbook

http://synrc.com/framework/web/

IRC FreeNode

#N2O