

Scala.js networking made easy

Olivier Blanvillain

PROGRAMMING METHODS LABORATORY, EPFL

January 26, 2015

This Presentation

1. Transport library
2. Latency compensation framework
3. Example: online multiplayer game

Motivation

- Many JavaScript APIs
- Many network programming models
- Goal: cross platform networking

DIVING IN...

```
trait Transport {  
  type Address  
  def listen(): Future[Promise[ConnectionListener]]  
  def connect(remote: Address): Future[ConnectionHandle]  
  def shutdown(): Future[Unit]  
}  
  
trait ConnectionHandle {  
  def handlerPromise: Promise[MessageListener]  
  def write(message: String): Unit  
  def closedFuture: Future[Unit]  
  def close(): Unit  
}  
  
type ConnectionListener = ConnectionHandle => Unit  
type MessageListener = String => Unit
```

```
trait Transport {  
  type Address  
  def listen(): Future[Promise[ConnectionListener]]  
  def connect(remote: Address): Future[ConnectionHandle]  
  def shutdown(): Future[Unit]  
}  
  
trait ConnectionHandle {  
  def handlerPromise: Promise[MessageListener]  
  def write(message: String): Unit  
  def closedFuture: Future[Unit]  
  def close(): Unit  
}  
  
type ConnectionListener = ConnectionHandle => Unit  
type MessageListener = String => Unit
```

```
trait Transport {  
  type Address  
  def listen(): Future[Promise[ConnectionListener]]  
  def connect(remote: Address): Future[ConnectionHandle]  
  def shutdown(): Future[Unit]  
}  
  
trait ConnectionHandle {  
  def handlerPromise: Promise[MessageListener]  
  def write(message: String): Unit  
  def closedFuture: Future[Unit]  
  def close(): Unit  
}  
  
type ConnectionListener = ConnectionHandle => Unit  
type MessageListener = String => Unit
```

```
trait Transport {  
  type Address  
  def listen(): Future[Promise[ConnectionListener]]  
  def connect(remote: Address): Future[ConnectionHandle]  
  def shutdown(): Future[Unit]  
}  
  
trait ConnectionHandle {  
  def handlerPromise: Promise[MessageListener]  
  def write(message: String): Unit  
  def closedFuture: Future[Unit]  
  def close(): Unit  
}  
  
type ConnectionListener = ConnectionHandle => Unit  
type MessageListener = String => Unit
```


Client Example

```
val transport = new WebSocketClient()
val url = WebSocketUrl("ws://echo.websocket.org")

val futureConnection = transport.connect(url)
futureConnection.foreach { connection =>
    connection.handlerPromise.success { message =>
        print("Received: " + message)
        connection.close()
    }
    connection.write("Hello WebSocket!")
}
```

Client Example

```
val transport = new WebSocketClient()
val url = WebSocketUrl("ws://echo.websocket.org")

val futureConnection = transport.connect(url)
futureConnection.foreach { connection =>
    connection.handlerPromise.success { message =>
        print("Received: " + message)
        connection.close()
    }
    connection.write("Hello WebSocket!")
}
```

Client Example

```
val transport = new WebSocketClient()
val url = WebSocketUrl("ws://echo.websocket.org")

val futureConnection = transport.connect(url)
futureConnection.foreach { connection =>
    connection.handlerPromise.success { message =>
        print("Received: " + message)
        connection.close()
    }
    connection.write("Hello WebSocket!")
}
```

Targeted Technologies

- WebSocket
- SockJS
- WebRTC

WebSocket Support, caniuse.com

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome Android
			1 5.1					
8			2 6.1					
9	33	37	7		7.1		4.3	
10	34	38	7.1		8		4.4	
11	35	39	8	26	8.1	8	37	39
TP	36	40		27				
	37	41		28				
	38	42						

Availability: ~84%

SockJS, Supported Transports

<i>Transport</i>	<i>References</i>
websocket (rfc6455)	rfc 6455
websocket (hixie-76)	draft-hixie-thewebsocketprotocol-76
websocket (hybi-10)	draft-ietf-hybi-thewebsocketprotocol-10
xhr-streaming	Transport using Cross domain XHR streaming capability (readyState=3).
xdr-streaming	Transport using XDomainRequest streaming capability (readyState=3).
eventsourcing	EventSource .
iframe-eventsourcing	EventSource used from an iframe via postMessage .
htmlfile	HtmlFile .
iframe-htmlfile	HtmlFile used from an iframe via postMessage .
xhr-polling	Long-polling using cross domain XHR .
xdr-polling	Long-polling using XDomainRequest .
iframe-xhr-polling	Long-polling using normal AJAX from an iframe via postMessage .
jsonp-polling	Slow and old fashioned JSONP polling .

WebRTC

- Peer to peer
- Made for Video, Audio and Data
- Supports TCP, UDP and SCTP
- RTC = Real Time Communication

WebRTC Connection Establishment

- Requires a signaling channel
- Typically through a relay server
- `class WebRTCClient extends Transport {
 type Address = ConnectionHandle
 ...
}`

WebRTC Support, caniuse.com

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome Android
			5.1					
8			6.1					
9	33	37	7		7.1		4.3	
10	34	38	7.1		8		4.4	
11	35	39	8	26	8.1	8	37	39
TP	36	40		27				
	37	41		28				
	38	42						

Availability: ~54%

Transport Implementations

Platform	WebSocket	SockJS	WebRTC
JavaScript	client	client	client
Play Framework	server	server	-
Netty	both	inc.	-
Tyrus	client	-	-

NETWORK PROGRAMMING ABSTRACTIONS

The Actor Model

- Akka on the JVM
- scala-js-actors on the browser
- Let's do everything with actors!

Actor Transport Wrapper

```
class ActorWrapper[T <: Transport](t: T) {  
  type Handler = ActorRef => Props  
  def acceptWithActor(handler: Handler): Unit  
  def connectWithActor(  
    address: t.Address)(handler: Handler): Unit  
}
```

Actor Transport Wrapper

```
class ActorWrapper[T <: Transport](t: T) {  
  type Handler = ActorRef => Props  
  def acceptWithActor(handler: Handler): Unit  
  def connectWithActor(  
    address: t.Address)(handler: Handler): Unit  
}
```

Connection Handling Actor

```
class YellingActor(out: ActorRef) extends Actor {  
  override def preStart = println("Connected")  
  override def postStop = println("Disconnected")  
  def receive = {  
    case message: String =>  
      println("Received: " + message)  
      out ! message.toUpperCase  
  }  
}
```

Connection Handling Actor

```
class YellingActor(out: ActorRef) extends Actor {  
  override def preStart = println("Connected")  
  override def postStop = println("Disconnected")  
  def receive = {  
    case message: String =>  
      println("Received: " + message)  
      out ! message.toUpperCase  
  }  
}
```


Remote Procedure Calls

- Wrapper around Autowire
- Future based RPC
- Agnostic of the serialization library

```
trait Api {  
  def doThing(i: Int, s: String): Seq[String]  
}
```

```
object Server extends Api {  
  def doThing(i: Int, s: String) = Seq.fill(i)(s)  
}  
  
val transport = new WebSocketServer(8080, "/ws")  
new MyRpcWrapper(transport).serve(_.route[Api](Server))
```

```
val transport = new WebSocketClient()  
val url = WebSocketUrl("ws://localhost:8080/ws")  
val client = new MyRpcWrapper(transport).connect(url)  
val result: Future[Seq[String]] =  
  client[Api].doThing(3, "ha").call()
```

```
trait Api {  
  def doThing(i: Int, s: String): Seq[String]  
}
```

```
object Server extends Api {  
  def doThing(i: Int, s: String) = Seq.fill(i)(s)  
}  
val transport = new WebSocketServer(8080, "/ws")  
new MyRpcWrapper(transport).serve(_.route[Api](Server))
```

```
val transport = new WebSocketClient()  
val url = WebSocketUrl("ws://localhost:8080/ws")  
val client = new MyRpcWrapper(transport).connect(url)  
val result: Future[Seq[String]] =  
  client[Api].doThing(3, "ha").call()
```

```
trait Api {  
  def doThing(i: Int, s: String): Seq[String]  
}
```

```
object Server extends Api {  
  def doThing(i: Int, s: String) = Seq.fill(i)(s)  
}  
  
val transport = new WebSocketServer(8080, "/ws")  
new MyRpcWrapper(transport).serve(_.route[Api](Server))
```

```
val transport = new WebSocketClient()  
val url = WebSocketUrl("ws://localhost:8080/ws")  
val client = new MyRpcWrapper(transport).connect(url)  
val result: Future[Seq[String]] =  
  client[Api].doThing(3, "ha").call()
```

```
trait Api {  
  def doThing(i: Int, s: String): Seq[String]  
}
```

```
object Server extends Api {  
  def doThing(i: Int, s: String) = Seq.fill(i)(s)  
}  
  
val transport = new WebSocketServer(8080, "/ws")  
new MyRpcWrapper(transport).serve(_.route[Api](Server))
```

```
val transport = new WebSocketClient()  
val url = WebSocketUrl("ws://localhost:8080/ws")  
val client = new MyRpcWrapper(transport).connect(url)  
val result: Future[Seq[String]] =  
  client[Api].doThing(3, "ha").call()
```

LATENCY COMPENSATION

LIVING WITH LAG*

*when slow internet causes disruption or delay to the user




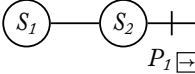
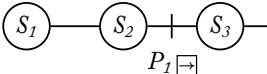
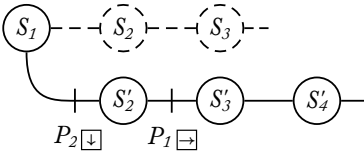
Let's see how **Google Docs** does it!

```
sudo tc qdisc add dev wlan0 root netem delay 3000ms
```

```
sudo tc qdisc del dev wlan0 root netem
```



```
nextState: (State, Set[Action]) => State
```

Received	Time	State Graph	Rendered
-	t_1		S_1
$P_1 \boxed{\rightarrow}$ at t_2	t_2		S_2
-	t_3		S_3
$P_2 \boxed{\downarrow}$ at t_1	t_4		S'_4

Latency Compensation Framework

- Peer to peer
- Zero input latency (predictive)
- Eventual consistency

Functional Interface

```
class Engine(  
    initState: State,  
    nextState: (State, Set[Action]) => State,  
    render: State => Unit,  
    broadcastConnection: ConnectionHandle) {  
  
    def triggerRendering(): Unit  
    def futureAct: Future[Action => Unit]  
}
```

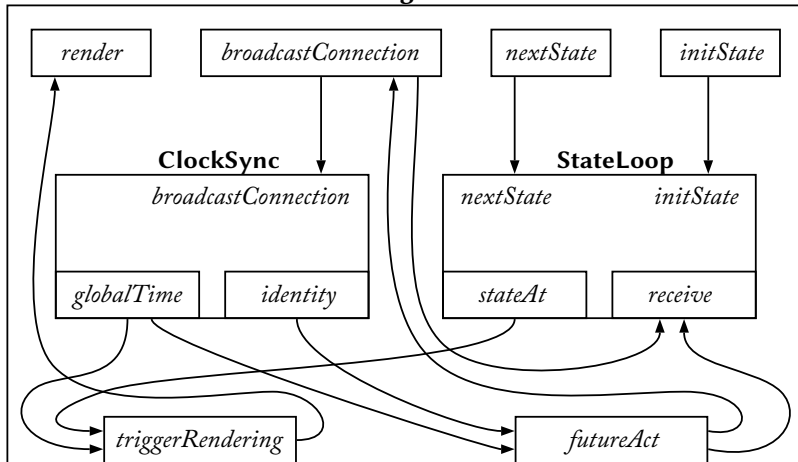
Functional Interface

```
class Engine(  
  initState: State,  
  nextState: (State, Set[Action]) => State,  
  render: State => Unit,  
  broadcastConnection: ConnectionHandle) {  
  
  def triggerRendering(): Unit  
  def futureAct: Future[Action => Unit]  
}
```

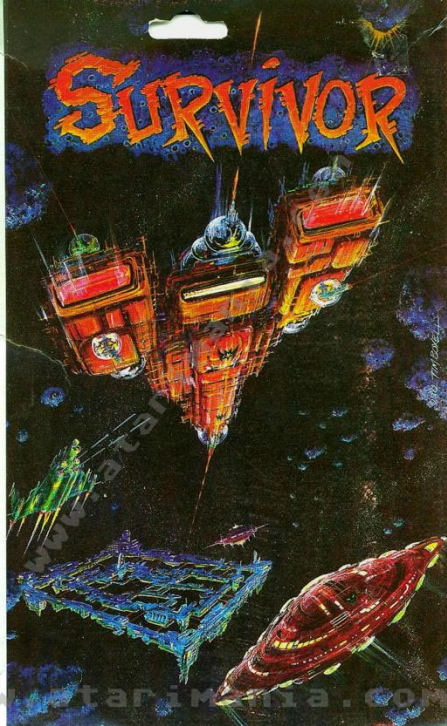
Functional Interface

```
class Engine(  
    initState: State,  
    nextState: (State, Set[Action]) => State,  
    render: State => Unit,  
    broadcastConnection: ConnectionHandle) {  
  
    def triggerRendering(): Unit  
    def futureAct: Future[Action => Unit]  
}
```

Engine



SYNAPSE SOFTWARE



Demo

1. JVM and JavaScript together
2. WebRTC, server shutdown
3. Latency compensation [in action](#)

THANKS!

BONUS SLIDES

Server Example

```
val transport = new WebSocketServer(8080, "/ws")
try {
  transport.listen().foreach { _.success { connection =>
    connection.handlerPromise.success { message =>
      connection.write(message)
    }
  }}
} finally transport.shutdown()
```

WebRTC connection establishment

```
val websocketClient = new WebSocketClient()
val webRTCClient = new WebRTCClient()
val relayURL = WebSocketUrl("ws://localhost:8080/ws")

val signalingChannel: Future[ConnectionHandle] =
    websocketClient.connect(relayURL)

val p2pConnection: Future[ConnectionHandle] =
    signalingChannel.flatMap(webRTCClient.connect(_))
```

React

- Re-render the whole application every frame
- **def** render(state: State): Html
- Virtual DOM diff algorithm