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.onSuccess { case connection =>
  connection.write("Hello WebSocket!")
  connection.handlerPromise.success { message =>
    print("Received: " + message)
    connection.close()
  }
}
```

Client Example

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

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

Client Example

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

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

Client Example

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

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

Server Example

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

Server Example

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

Server Example

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

Server Example

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

Server Example

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

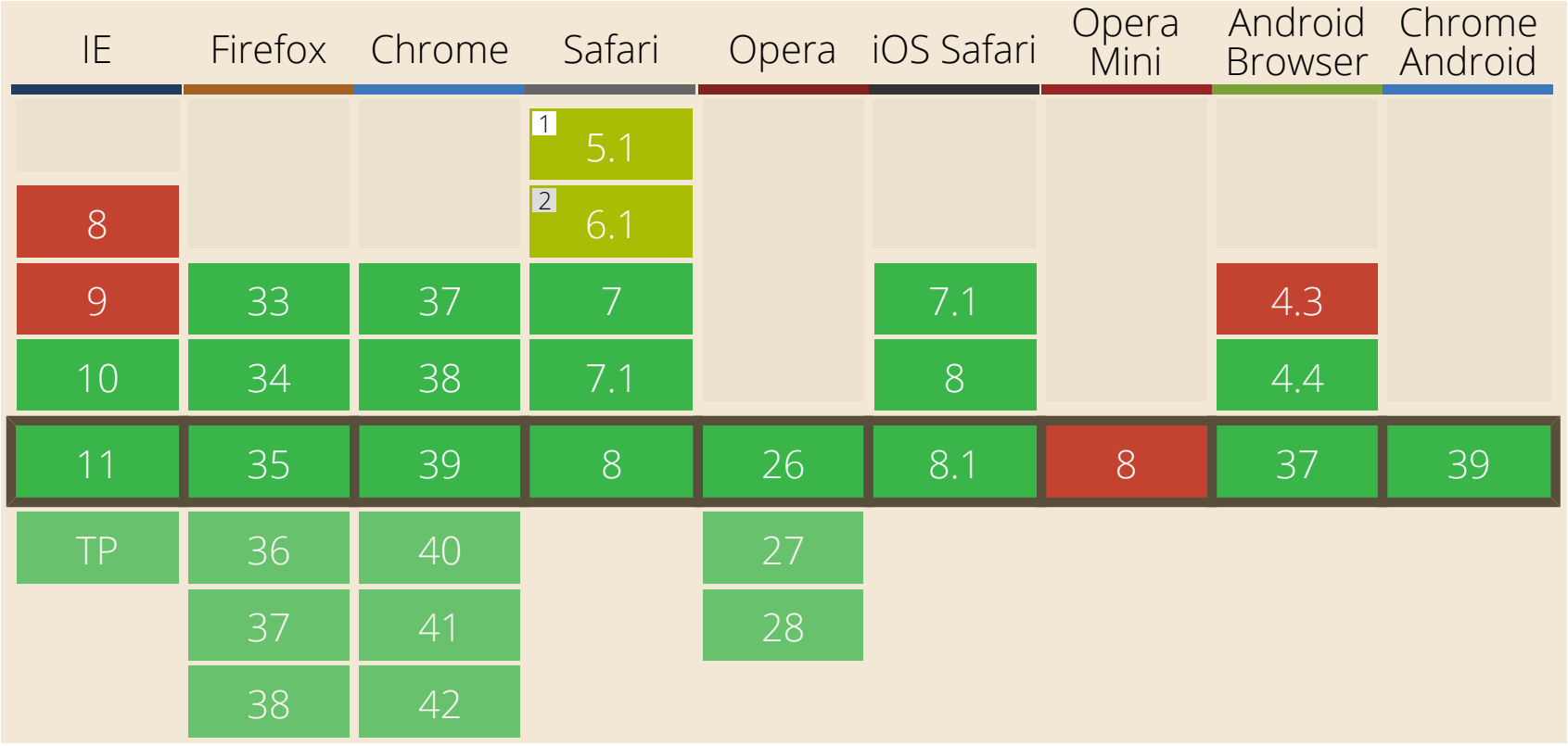
Targeted Technologies

- WebSocket
- SockJS
- WebRTC

WebSocket

- Bidirectional client-server communication
- Handshake = HTTP upgrade request
- Long lived TCP

WebSocket Support, caniuse.com



Availability: ~84%

SockJS

- WebSocket emulation, same API
- Fallbacks to HTTP requests + long polling
- Supports sticky sessions
- Well defined protocol, standard test suite

SockJS, Supported Transports

Transport	*References*
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).
eventsource	EventSource.
iframe-eventsource	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.

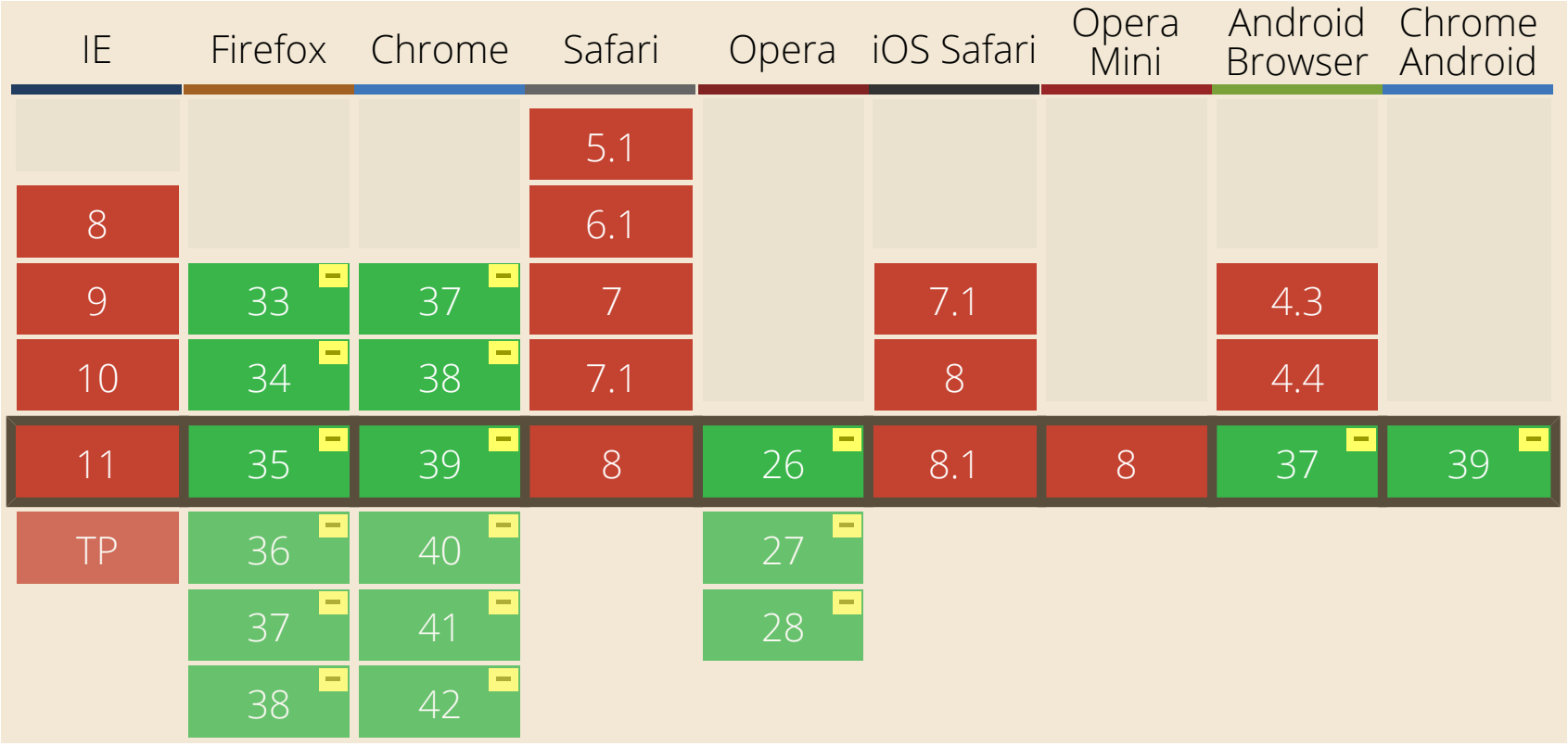
WebRTC

- 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 thought a relay server
- ```
  class WebRTCClient extends Transport {
    type Address = ConnectionHandle
    ...
  }
  ```
  ```

WebRTC Support, caniuse.com



Availability: ~54%

Transport Implementations

Platform	WebSocket	SockJS	WebRTC
JavaScript	client	client	client
Play Framework	server	server	-
Netty	both	-	-
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
  }
}
```

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

```scala
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()
```

```scala
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()
```

```scala
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()
```

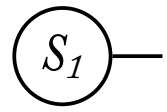
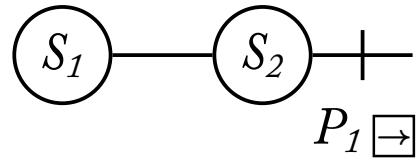
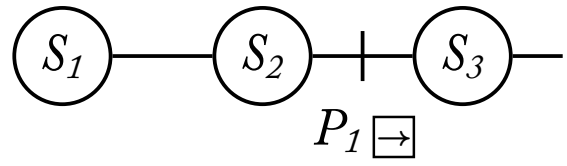
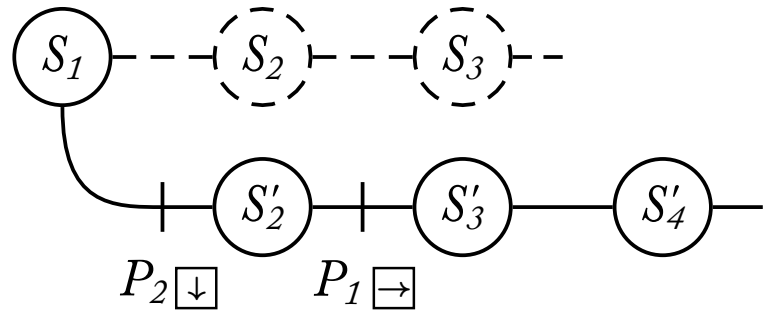
```
trait Api {
  def doThing(i: Int, s: String): Seq[String]
}

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
TIGA Privat Roller
Privat Roller TIGA

Let's see how Google Docs does it!

```
sudo tc qdisc add dev eth0 root netem delay 3000ms

sudo tc qdisc del dev eth0 root netem
```

Received	Time	State Graph	Rendered
-	t_1		S_1
P_1 ⊡→ at t_2	t_2		S_2
-	t_3		S_3
P_2 ⊡↓ at t_1	t_4		S'_4

Latency Compensation Framework

- Predictive algorithm
- Peer to peer
- Zero input latency
- Eventual consistency

Functional interface

```
case class Action[Input](input: Input, peer: Peer)

class Engine[Input, State](
    initState: State,
    nextState: (State, Set[Action[Input]]) => State,
    render: State => Unit,
    broadcastConnection: ConnectionHandle) {

  def triggerRendering(): Unit
  def futureAct: Future[Input => Unit]
}
```

Functional interface

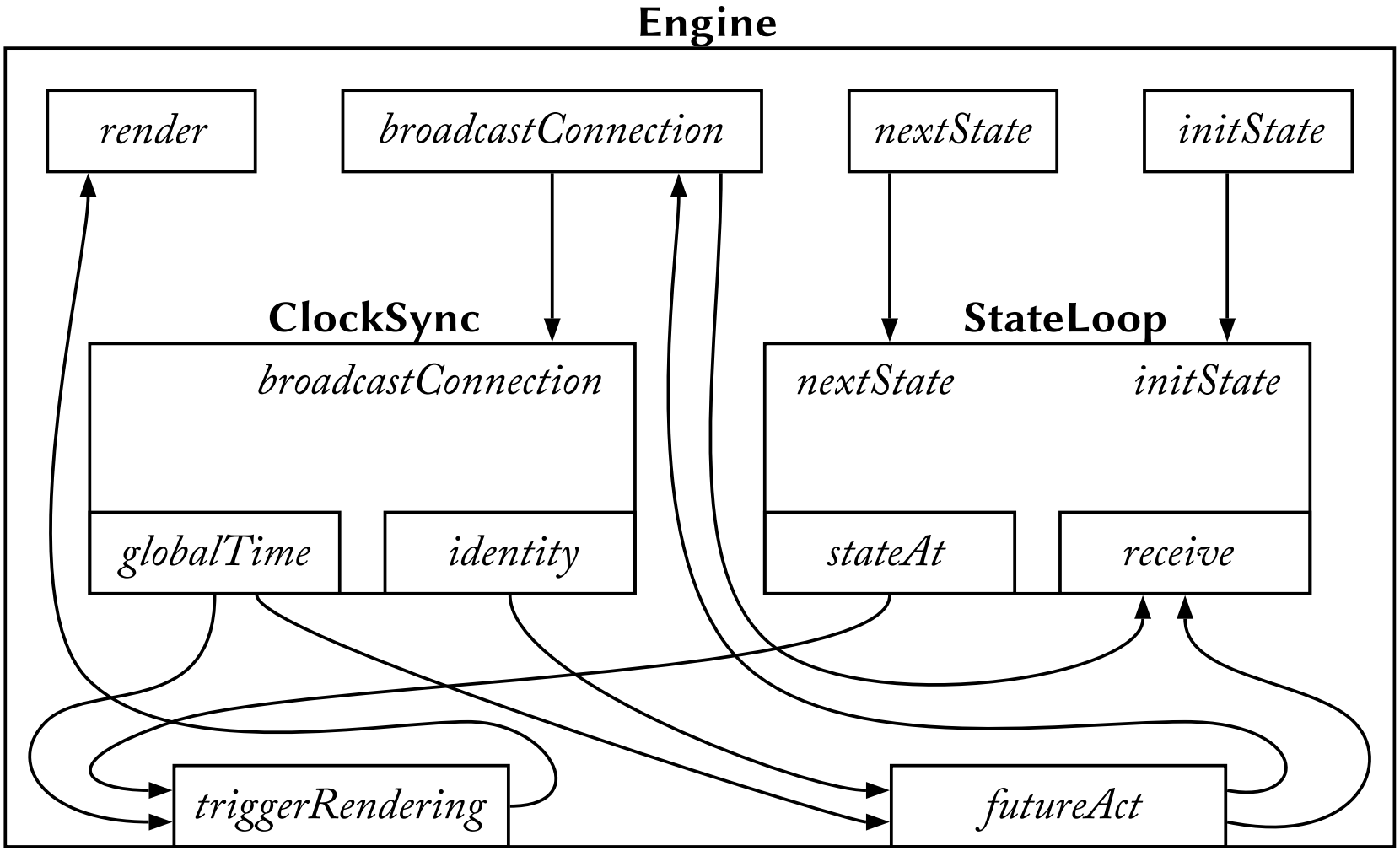
```
case class Action[Input](input: Input, peer: Peer)

class Engine[Input, State](
    initState: State,
    nextState: (State, Set[Action[Input]]) => State,
    render: State => Unit,
    broadcastConnection: ConnectionHandle) {

  def triggerRendering(): Unit
  def futureAct: Future[Input => Unit]
}
```

Functional interface

```
case class Action[Input](input: Input, peer: Peer)

class Engine[Input, State](
    initState: State,
    nextState: (State, Set[Action[Input]]) => State,
    render: State => Unit,
    broadcastConnection: ConnectionHandle) {

  def triggerRendering(): Unit
  def futureAct: Future[Input => Unit]
}
```

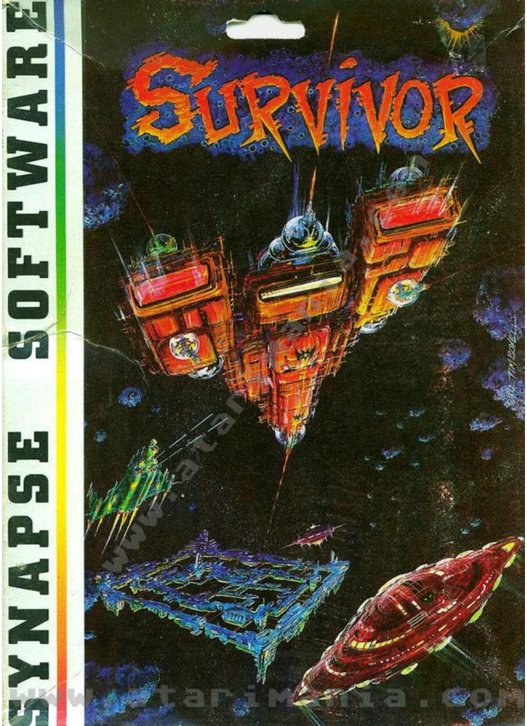
Functional interface

```
case class Action[Input](input: Input, peer: Peer)

class Engine[Input, State](
    initState: State,
    nextState: (State, Set[Action[Input]]) => State,
    render: State => Unit,
    broadcastConnection: ConnectionHandle) {

  def triggerRendering(): Unit
  def futureAct: Future[Input => Unit]
}
```


Engine
render
broadcastConnection
nextState
initState
ClockSync
broadcastConnection
globalTime
identity
StateLoop
nextState
initState
stateAt
receive
triggerRendering
futureAct

StateLoop Implementation

todo...



Demo

- Cross platform
- WebRTC
- Online

React

- Re-render the whole application every frame
- `def render(state: State): Html`
- Divided into Components
- Virtual DOM diff algorithm

Thanks!