École polytechnique fédérale
de Lausanne

Master Project in Computer Science

Programming Methods Laboratory

Scala.js networking made easy

Student:
Olivier Blanvillain

Assistant:
Sébastien Doeraene

Responsible professor:
Martin Odersky

External expert:
Aleksandar Prokopec

Handed in: January 16, 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Abstract

This thesis: - Transport library - Functional lag compensation framework - Nice showcase of Scala.js capability with a cross-platform real-time mutliplayer game.

Contents

List of Figures

List of Tables

List of Source Code Listings

Chapter 1

Introduction

- Context: What is Scala.js
- Relevance: importance of networking for Scala.js
- Motivation: Many JS APIs
  - Websocket
  - Comet
  - WebRTC
- Motivation: Many network programing models
  - Akka
  - RPC (type safe)
  - Steams (scalaz, akka-stream)
- Plan/Contributions

Chapter 2

Transport

- This section, scala-js-transport library, main contribution

2.1 A Uniform Interface

We begin our discussion by the definition of an interface for asynchronous transports, presented in Listing 2.1. This interface aims at *transparently* modeling the different underlying technologies, meaning that is simply delegates tasks to the actual implementation, without adding new functionalities.

```
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 closedFuture: Future[Unit]
  def write(message: String): Unit
  def close(): Unit
}
type ConnectionListener = ConnectionHandle => Unit
type MessageListener = String => Unit
```

Listing 2.1: Definition of the core networking interfaces.

A *Transport* can both *listen* for incoming connections and *connect* to remote *Transports*. Platforms limited to act either as client or server will return a failed future for either of these methods. In order to listen for incoming connections, the user of a *Transport* has to complete the promise returned by the listen method with a *ConnectionListener*. To keep the definition generic, *Address* is an abstract type. As we will see later, it varies greatly from one technology to another.

ConnectionHandle represents an opened connection. Thereby, it supports four type of interactions: writing a message, listening for incoming messages, closing the connection and listening for connection closure. Similarly to *Transport*, listening for incoming messages is achieved by completing a promise of *MessageListener*.

The presented *Transport* and *ConnectionHandle* interfaces have several advantages compared to their alternative in other languages, such the WebSocket interface in JavaScript. For example, errors are not transmitted by throwing exceptions, but simply returned as a failed future. Also, some incorrect behaviors such as writing to a no yet opened connection, or receiving duplicate notifications for a closed connection, are made impossible by construction. Thanks to support of futures and promises in Scala.js, these interfaces cross compile to both Java bytecode and JavaScript.

2.2 Implementations

The scala-js-transport library contains several implementations of *Transports* for WebSocket, SockJS and WebRTC. This subsection briefly presents the different technologies and their respective advantages. Table 2.1 summarizes the available *Transports* for each platform and technology.

Table 2.1: Summary of the available Transports.

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

2.2.1 WebSocket

WebSocket provides full-duplex communication over a single TCP connection. Connection establishment begin with an HTTP request from client to server. After the handshake is completed, the TCP connection used for the initial HTTP request is *upgraded* to change protocol, and kept open to become the actual WebSocket connection. This mechanism allows WebSocket to be wildly supported over different network configurations.

WebSocket is also well supported across different platforms. Our library provides four WebSocket *Transports*, a native JavaScript client, a Play Framework server, a Netty client/server and a Tyrus client. While having all three Play, Netty and Tyrus might seem redundant, each of them comes with its own advantages. Play is a complete web framework, suitable to build every component of a web application. Play is based on Netty, which means that for a standalone WebSocket server, using Netty directly leads to better performances and less dependencies. Regarding client side, the Tyrus library offers a standalone WebSocket client which is lightweight compared to the Netty framework.

2.2.2 SockJS

SockJS is a WebSocket emulation protocol which fallbacks to different protocols when WebSocket is not supported. Is supports a large number of techniques to emulate the sending of messages from server to client, such as AJAX long polling, AJAX streaming, EventSource and streaming content by slowly loading an HTML file in an iframe. These techniques are based on the following idea: by issuing a regular HTTP request from client to server, and voluntarily delaying the response from the server, the server side can decide when to release information. This allows to emulate the sending of messages from server to client which not supported in the traditional request-response communication model.

The scala-js-transport library provides a *Transport* build on the official SockJS JavaScript client, and a server on the Play Framework via a community plugin [5]. Netty developers have scheduled SockJS support for the next major release.

2.2.3 WebRTC

WebRTC is an experimental API for peer to peer communication between web browsers. Initially targeted at audio and video communication, WebRTC also provides *Data Channels* to communicate arbitrary data. Contrary to WebSocket only supports TCP, WebRTC can be configures to use either TCP, UDP or SCTP.

As opposed to WebSocket and SockJS which only need a URL to establish a connection, WebRTC requires a *signaling channel* in order to open the peer to peer connection. The *signaling channel* is not tight to a particular technology, its only requirement is to allow a back an forth communication between peers. This is commonly achieved by connecting both peers via WebSocket to a server, which then serves as a relay for the WebRTC connection establishment.

To simplify the process of relaying messages from one peer to another, our library uses picklers for *ConnectionHandle*. Concretely, when a *ConnectionHandle* object connecting node A and B is sent by B over an already established connection with C, the *ConnectionHandle* received by C will act as a connection between A and C, hiding the fact that B relays messages between the two nodes.

At the time of writing, WebRTC is implemented is Chrome, Firefox and Opera, and lakes support in Safari and Internet Explorer. The only non browser implementations are available on the node.js platform.

2.3 Wrappers

By using *Transport* interface, it is possible write programs with an abstract communication medium. We present two *Transport* wrappers, for Akka [9] and Autowire [3], which allow to work with different model of concurrency. Because Autowire and Akka (via [1]) can both be used on the JVM and on JavaScript, these wrappers can be used to build cross compiling programs compatible with all the *Transport* implementations presented in section 2.2.

2.3.1 Akka

The actor model is based on asynchronous message passing between primitive entities called actors. Featuring both location transparency and fault tolerance via supervision, the actor model is particularly adapted to distributed environment. Akka, a toolkit build around the actor model for the JVM, was partly ported to Scala.js by S. Doeraene in [1]. The communication interface implemented in [1] was revisited into the *Transport* wrapper presented in Listing 2.2.

The two methods *acceptWithActor* and *connectWithActor* use the underlying *listen* and *connect* methods of the wrapped *Transport*, and create an *handler* actor to handle the connection. The semantic is as follows: the *handler* actor is given an *ActorRef* in it is constructor, to which sending messages results in sending outgoing messages

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

Listing 2.2: Transport wrappers to handle connections with actors.

thought the connection, and messages received by the *handler* actor are incoming messages received from the connection. Furthermore, the life span of an *handler* actor is tight to life span of its connection, meaning that the *preStart* and *postStop* hooks can be used to detect the creation and the termination of the connection, and killing the *handler* actor results in closing the connection. Listing 2.3 shows an example of a simple *handler* actor which than sending back whatever it receives in uppercase.

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

Listing 2.3: Example of a connection handling actor.

Thanks to the picking mechanism developed in [1], it is possible to sent messages of any type thought a connection, given that implicit picklers are available for these types of messages. Out of the box, picklers for case classes and case objects can be macros-generated by the pickling library. In addition, an *ActorRef* pickler allows the transmission of *ActorRefs* thought a connection, making them transparently usable from the side of the connection as if they were references to local actors.

2.3.2 Autowire

Remote procedure call allow remote systems to communicate through an interface similar to method calls. The Autowire library allows to perform type-safe, reflection-free remote procedure calls between Scala system. It uses macros and is agnostic of both the transport-mechanism and the serialization library.

The scala-js-transport library offers a *RpcWrapper*, which makes internal use of Autowire to provide remote provide call on top of any of the available *Transports*. Because the *Transport* interface communicates with *Strings*, the *RpcWrapper* is able to set all the type parameters of Autowire, as well embedding the uPickle serialization library [4], thus trading flexibility to reduce boilerplate. Listing 2.4 shows a complete remote procedure call implementation on top of WebSocket.

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

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

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

Listing 2.4: Example of remote procedure call implementation.

The main strength of remote procedure calls are their simplicity and type-safety. Indeed, because of how similar remote procedure calls are to actual method calls, they can be used without any learning curve, and consistency between client and server side is can be verified by the compiler. However, this simplicity also comes with some draw backs. Contrary to the actor model which can be used to explicitly model the life span of connections, and different types of failures, there are no such things build in when using remote procedure calls. In order to implement a fine grain error detection and recovery mechanism on top of recovery procedure calls, one would have to work at a lower lever than the one offered by the model it self, which is with the *Transport* interface in our case.

2.4 Going further

The different *Transport* implementations and wrappers presented is this section allows for several interesting combinations. Because the scala-js-transport library is built with a central communication interface (presented in section 2.1), it is easily expendable in both directions. Indeed, any new implementation of the *Transport* interface with a different underlaying technologies would immediately be usable with the different wrappers. Analogously, any new wrapper on top of the *Transport* interface would automatically be compatible with the variety of available implementations.

All the implementations and wrappers are accompanied by integration tests. In most cases, these tests are built using the *Selenium WebDriver* included in the Play Framework test tools, which allow to access proper behavior of the library using real web browsers. Our tests for WebRTC include an extension of the default Play test tools to use two browsers on a single test. The tests can then be configures to run with two different browsers, such as Chrome and Firefox, to test their compatibility.

Chapter 3

Dealing with latency

- This section, the framework, the game

3.1 Latency Compensation

Working with distributed systems introduces numerous challenges compared to the use of a single machine. Much of the complexity comes from the communication links; limited throughput, risk of failure, and latency all have to taken into consideration. Our discussion will be focused on issues related to latency.

When thinking about latency sensitive application, the things that comes to mind might be multiplayer video games. In order to provide a fun and immersive experience, real-time games have to *feel* reactive. Techniques to compensate network latency also have uses in online communication/collaboration tools. Essentially, any application where a shared state can be simultaneously mutated by different peers is confronted with this problem. According to [6], the different latency compensation mechanisms can be divided into three categories: predictive techniques, delayed input techniques and time-offsettings techniques.

Predictive techniques estimate the current value of the global state using information available locally. These techniques are traditionally implemented using a central authoritative server which gathers inputs from all clients, computes the value of global state, and broadcasts this state back to all clients. It then possible to do prediction on the client side by computing a "throwaway" state using the latest local inputs, which is later replaced by the state the server as soon as it is received. This architecture with a central authoritative server is used in most *First person shooter* games, such as the ones built using the Source Engine [10].

Delayed input techniques defer the execution of all actions to allow simultaneous execution by all client. Very often, the perceived latency can be reduced by instantly starting a purely visual animation as soon as the an input is entered but still delaying the any actual effect of the action. This solution is typically used in games where the game state is too large to be frequently sent over the network. In this case, peers would directly exchange the user inputs and simultaneously simulate the game with a fixed delay. Having a centralized server is then not mandatory, and peer to peer configurations might be used to reduce communication latency. The classical *Age of Empires* uses the input delayed techniques with a fixed delay of 500 ms [8].

*TODO: *Time-offsettings techniques**... [7].

Each technique comes with its own advantages and disadvantages, and are essentially making different tradeoffs between consistency and responsiveness. Without going into further details on the different latency compensation techniques, this introduction should give the reader an idea of the variety of possible solutions and their respective sophistication.

3.2 A Functional Framework

We now present a Scala framework for latency compensation. By imposing a purely functional design to its users, the framework is focused on correctness, leaving very little room for runtime errors.

It implements predictive latency compensation in a distributed fashion. As opposed to traditional architecture for prediction such as the one described in [10], our framework does uses any authoritative node to hold the global state of the application, but functions in a purely peer to peer fashion.

To do so, each peer runs a local simulation of the application up to the current time, using all the inputs available. Whenever an input is transmitted at a peer via the network, this remote input is necessarily slightly out of date because of the communication latency. In order to incorporate out of date input into the local simulation, the framework rolls back the state of the simulation as it was just before the time of emission of this input, and replays the simulation until the current time, thus taking into account the remote input. Figure 3.1 shows this process in action from the point of view of peer *P1*. In this example, *P1* emits an input at time *t2*, and then, at time *t3*, receives an input from *P2* which was emitted at time *t1*. The framework then invalidates a branch of the state tree, *S2-S3*, and computes *S2'-S3'-S4'* which take into account both inputs.

eventually consistent instant feed back

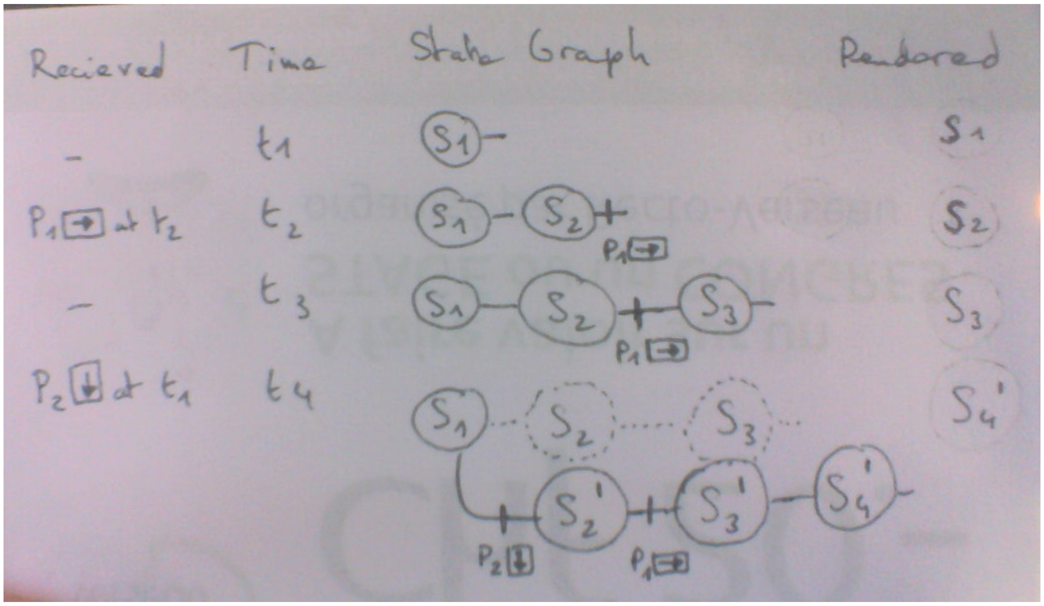

Figure 3.1: The figure...

- The solution: simulate with local inputs, go back in time when getting remote inputs (Figure)
- Functional interface (Listing)
- Requires: initialState, nextState, render, transport

3.3 Architecture and Implementation

- [8] issues about determinism and game states getting out-of-sync
- Identical simulation on all peers
- Immutability everywhere
- Pure function
- Goal: Cross platform JS/JVM realtime lag compensation framework

3.3.1 clock sync

- Where do we cheat: clock sync

3.3.2 back in time

- Scala List and Ref quality and fixed size buffer solution

3.4 Putting It All Together: A Real-Time Multiplayer Game

- History: Scala.js port of a JS port of a Commodore 64 game
- Functional GUI with React (Hack for the JVM version)
- Everything but input handler shared (but UI shouldn't…)
- Functional design of gun fire (-> function of time!)
- WebRTC with SockJS fallback
- Live reload heaven
- Results: 60FPS on both platforms, lag free gameplay
- Results: Lag Compensation in action (Screenshots)

Chapter 4

Related Work

- Js/NodeJs, relies on duck typing
- Closure
- Steam Engine/AoE/Sc2
- [6]
- Cheating concerns

Chapter 5

Conclusion and Future Work

- Web workers
- scalaz-stream/akka-stream wrappers
- More utilities on top of Transport

Appendix A

React

TODO: React [2] is a JavaScript library for building user interfaces.

- Just the UI: Lots of people use React as the V in MVC. Since React makes no assumptions about the rest of your technology stack, it's easy to try it out on a small feature in an existing project.
- Virtual DOM: React uses a virtual DOM diff implementation for ultra-high performance. It can also render on the server using Node.js, no heavy browser DOM required.
- Data flow: React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional data binding.

References

[1] S. Doeraene. scala-js-actors. http://github.com/sjrd/scala-js-actors. 7, 8

[2] Facebook. React. http://facebook.github.io/react/. 17

[3] L. Haoyi. Autowire. http://github.com/lihaoyi/autowire. 7

[4] L. Haoyi. upickle. http://github.com/lihaoyi/upickle. 9

[5] F. Muccio. play2-sockjs. http://github.com/fdimuccio/play2-sockjs. 6

[6] C. Savery and T. Graham. Timelines: simplifying the programming of lag compensation for the next generation of networked games. *Multimedia Systems*, 19(3):271–287, 2013. 11, 15

[7] P. M. Sharkey, M. D. Ryan, and D. J. Roberts. A local perception filter for distributed virtual environments. In *Proceedings of the Virtual Reality Annual International Symposium*, VRAIS '98, pages 242–, Washington, DC, USA, 1998. IEEE Computer Society. 12

[8] M. Terrano and P. Bettner. 1500 archers on a 28.8: Network programming in age of empires and beyond. http://gamasutra.com/view/feature/3094, 2001. 12, 13

[9] TypeSafe. Akka. http://akka.io/. 7

[10] Valve. Source multiplayer networking. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. 11, 12