

Scala.js networking made easy

Olivier Blanvillain
EPFL, Switzerland
olivier.blanvillain@gmail.com

January 16, 2015

Abstract

1 Introduction

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

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. This interface aims at *transparently* modeling the different underlying technologies, meaning that it 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: 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 not 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 1 summarizes the available *Transports* for each platform and technology.

Table 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 begins 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 widely 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. It 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 is 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 [3]. 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 configured 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 tied to a particular technology, its only requirement is to allow a back and 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 in Chrome, Firefox and Opera, and lacks 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 [4] and Autowire [2], 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 subsection 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 3.

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

Listing 3: *Transport* wrappers to handle connections with actors.

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 its constructor, to which sending messages results in sending outgoing messages through the connection, and messages received by the *handler* actor are incoming messages received from the connection. In addition, 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 4 shows an example of a simple *handler* actor which then sending back whatever it receives in uppercase.

This semantic is inspired from actor interface for WebSocket implemented in the Play Framework, with the exception that it relies on the picking mechanism developed in [1]. As a result, if *TODO: ActorRef*

```

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

```

Listing 4: C.

2.3.2 Autowire RPC

- Short intro
- The interface
- Some code
- Strong point

2.3.3 Going further

- Adding Wrappers, all Transports for free!
- Adding Transports, all Wrappers for free!
- A word on Integration Tests (Two browsers)

3 Lag Compensation Framework

- Why do we need this
- Google Docs
- Goal: Cross platform JS/JVM realtime lag compensation framework

3.1 Compensate Network Latency

- Traditional solutions (actual lag, fixed delay with animation)
- Solution: go back in time (Figure)
- Clock synked, same game simulated on both platforms

3.2 Architecture

- Requires: initialState, nextState, render, transport
- Immutability everywhere
- Scala List and Ref quality and fixed size buffer solution

4 Example: A Real Time Multi-player Game

- History: Scala.js port of a JS port of a Commodore 64 game

4.1 Functional User Interface

- React UI
- React
- Hack for the JVM version)

4.2 Functional User Interface

- React UI
- React
- Hack for the JVM version)

4.3 Result

- Everything but input handler shared (but UI shouldn't...)
- WebRTC with SockJS fallback
- Results: 60FPS on both platforms, lag free gameplay
- Results: Lag Compensation in action (Screenshots)

5 Related Work

- Js/NodeJs, relies on duck typing
- Closure
- Steam Engine/AoE/Sc2/Google docs

6 Conclusion and Future Work

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

References

- [1] S. Doeraene. scala-js-actors. <http://github.com/sjrd/scala-js-actors>.
- [2] L. Haoyi. Autowire. <http://github.com/lihaoyi/autowire>.
- [3] F. Muccio. play2-sockjs. <http://github.com/fdimuccio/play2-sockjs>.
- [4] TypeSafe. Akka. <http://akka.io/>.