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 programing 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

In order to interchangeably use different means of communication, we begin by the definition of an interface for asynchronous transports. This interface aims at *transparently* modeling the different underlying technologies, meaning it does not add functionality but simply serves as a delegator.

```
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
```

Figure 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 being notified of 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 alternative found 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 can be cross compiles to both Java bytecode and JavaScript.

2.2 Implementations

The scala-js-transport library provides several implementations of Transport across three communication technologies: WebSocket, SockJS and WebRtc. We briefly present these implementations grouped by technology.

WebSocket provides full-duplex communication over a single TCP connection. Connections are established by emitting an HTTP request to initiate a handshake, the HTTP connection is then upgraded to become the 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*, the native JavaScript client, the Play Framework server, the Netty client/server and the Tyrus client. While having all three Play, Netty and Tyrus might seem redundant, each of them has its advantages. Play is a complete web framework is more suitable to build every component of a web application. On the other hand, Netty might be more suitable to build a standalone WebSocket server. Since Play is itself based on Netty, it adds some overhead in term of performances and dependencies. Regarding client side, the Tyrus library offers a standalone WebSocket client which is lightweight compared to the equivalent Netty implementation. It would therefore be more suitable for a mobile JVM client.

SockJS is a WebSocket emulation protocol which fallbacks to different protocol when WebSocket is not supported.

- js (WebSocket client, SockJS client, WebRtc client)
- netty (WebSocket server, SockJS server (next netty))
- tyrus (WebSocket client)
- play (WebSocket client, SockJS client (plugin))

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

Table 1: Here's the caption. It, too, may span

2.3 Wrappers

- Works fine with the raw api
- Akka
- Autowire (RPC)

2.4 Going further

- Testing infrastructure
- Two configurable browsers

3 Example: A Cross-platform Multiplayer Game

- Goal: Cross platform JS/JVM realtime mutiplayer game
- History: Scala.js port of a JS port of a Commodore 64 game

3.1 Architecture

- Purely functional multiplayer game engine
- Clock synked, same game simulated on both platforms
- Requires: initialState, nextState, render, transport
- Result: Immutability everywhere
- Result: everything but input handler & UI is shared

3.2 Compensate Network Latency

- Traditional solutions (actual lag, fixed delay with animation)

- Solution: go back in time (Figure)
- Scala List and Ref quality and fixed size buffer solution

3.3 Implementation

- React UI (& hack for the JVM version)
- Simple Server for matchmaking
- WebRtc with SockJS fallback
- Results: 60FPS on both platforms, lag free gameplay
- Results: Lag Compensation in action (Screenshots)

4 Related Work

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

5 Conclusion and Future Work

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

[1]

References

[1] J. Y. Gil. LaTeX 2_ε for graduate students. manuscript, Haifa, Israel, 2002.