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

- Scope: A unified, transparent interface to build upon.
- No magic.

2.1 The interface:

```
trait Transport {}
trait ConnectionHandle {}
```

2.2 Implementations

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

2.3 Wrappers

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

2.4 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
- 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.