École polytechnique fédérale
de Lausanne

Master Project in Computer Science
Programming Methods Laboratory

Scala.js networking made easy

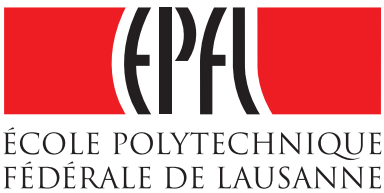
Student:
Olivier Blanvillain

Assistant:
Sébastien Doeraene

Responsible professor:
Martin Odersky

External expert:
Aleksandar Prokopec

Handed in: January 16, 2015



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

TODO: 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 [7]. 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 acts 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.

The scala-js-transport library provides two WebRTC *Transports*, *WebRTCClient* and *WebRTCClientFallback*. The later implements some additional logic to detect WebRTC support, and automatically fall back to using the signaling channel as substitute for WebRTC if either peer does not support it.

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 [11] and Autowire [5], which allow to work with different model of concurrency. Because Autowire and Akka (via [3]) 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 [3]. The communication interface implemented in [3] was revisited into the *Transport* wrapper presented in Listing 2.2.

```
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.

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 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 [3], 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 [6], 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 require little learning for the programmer. In addition, consistency between

client and server side can be verified at compile time, and integrated development environment functionalities such as *refactoring* and *go to definition* work out of the box. However, this simplicity also comes with some draw backs. Contrary to the actor model which explicitly models the life span of connections, and different failure scenarios, this is not build in when using remote procedure calls. In order to implement fine grain error handling and recovery mechanism on top of remote procedure calls, one would have to work at a lower lever than the one offered by the model itself, that 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 around a central communication interface, it is easily expendable in both directions. Any new implementation of the *Transport* interface with for a different platform or technology would immediately be usable with all the wrappers. Analogously, any new *Transport* wrapper would automatically be compatible with the variety of available implementations.

All the implementations and wrappers are accompanied by integration tests. These tests are built using the *Selenium WebDriver* to check proper behavior of the library using real web browsers. Our tests for WebRTC use two browsers, wich can be configured to be run with two different browsers to test their compatibility.

Chapter 3

Dealing with latency

TODO: This section, the framework, the game

3.1 Latency Compensation

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

When thinking about latency sensitive application, the example coming 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 such as *Google Docs*. Essentially, any application where a shared state can be simultaneously mutated by different peers is confronted to issues related to latency.

While little information is available about the most recent games and collaborative applications, the literature contains some insightful material about the theoretical aspects of latency compensation. According to [8], the different 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. The described architecture with a central authoritative server is used in most *First person shooter* games, including the recent titles built using the Source Engine [12]. Predictions are sometimes limited to certain type of objects and interactions, such as with the *dead reckoning* [1] technique that estimate the current positions of moving objects based on their earlier position, velocity and acceleration information.

Delayed input techniques defer the execution of all actions to allow simultaneous execution by all peers. This solution is typically used in application where the state, or variations of state, are too large to be frequently sent over the network. In this case, peers would directly exchange the user inputs and simultaneously simulate application with a fixed delay. Having a centralized server is then not mandatory, and peer to peer configurations might be used to reduce communication latency. Very often, the perceived latency is reduced by instantly emitting a purely aesthetic feedback (visual and/or sonorous) as soon as the an input is entered, but delaying the actual effects of the action. The classical *Age of Empires* series uses this techniques with a fixed delay of 500 ms, and supports up to 8 players and 1600 independently controllable entities [10].

Time-offsettings techniques add a delay in the application of remote inputs. Different peers will then see different versions of the application state over time. Local perception filters [9] are an example of time-offsettings technique where the amount of delayed applied to world entities is proportional to their distance to peer avatar. As a result, a user can interact in real time with entities spatially close to him, and see the interaction at a distance as if they where appending in real time. The most important limitation of local perception filters is that peers avatar have to be kept at a minimum distance from each other, and can only interact by exchanging passive entities (such as bullets). Indeed, passed a certain proximity threshold the time distortion becomes smaller than the network latency which invalidates the model.

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 scala-lag-comp, a Scala framework for predictive latency compensation.

By imposing a purely functional design to its users, the framework focuses on correctness and leaves very little room for runtime errors. It implements predictive latency compensation in a fully distributed fashion. As opposed to the traditional architectures implementing prediction techniques, such as the one described in [12], our framework does uses any authoritative node to hold the global state of the application, and can therefore 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 information available locally. Whenever an input is transmitted to a peer via the network, this remote input will necessarily be slightly out of date because of communication latency. In order to incorporate this 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 remote input, and then replays the simulation up to the current time. 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*. Then, at time *t3*, *P1* receives an input from *P2* which was emitted at time *t1*. At this point, the framework invalidates a branch of the state tree, *S2-S3*, and computes *S2'-S3'-S4'* to take into account both inputs.

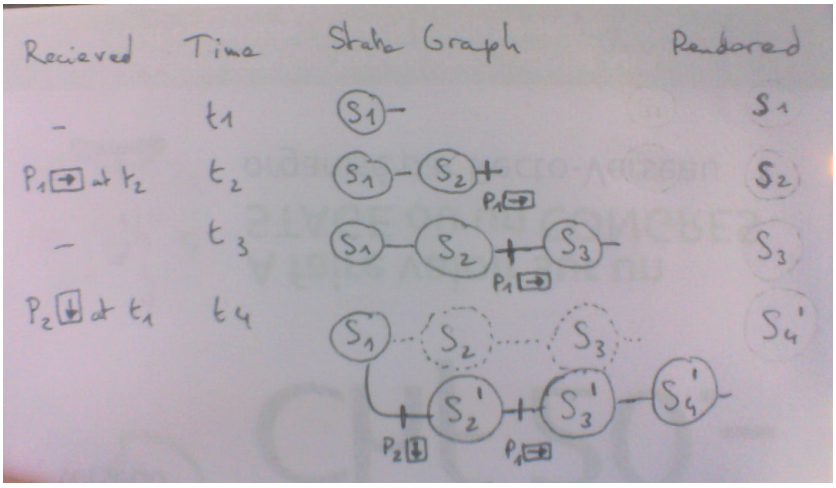


Figure 3.1: Growth of the state graph over time, from the point of view of *P1*.

By instantaneously applying local input, the application feels highly reactive

to the end user, and this reactiveness is not affected by variations on the quality of the connection. This property comes with the price of having short periods inconsistency between the application state on the different peers. This inconsistency lasts until all peers are aware of all inputs, at which point the simulation recovers its global consistency.

By nature, this design requires a careful management of the application state and it evolutions over time. Indeed, even a small variation between two remote simulations can cause a divergence, and result in out-of-sync application states. [10] reports out-of-sync issues as having been one of their main difficulty during the development of multiplayer features. In our case, the *roll back in time* procedure introduces another source of potential mistake. Any mutation in a branch of the simulation that would not properly be canceled when rolling back to a previous state could induce serious bugs, of the hard to find and hard to reproduce kind.

To cope with these issues, the scala-lag-comp framework takes entirely care of state management and imposes a functional programming style to its users. Listing 3.1 defines the unique interface exposed by the framework: *Engine*.

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

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

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

Listing 3.1: Interface of the latency compensation framework.

An application is entirely defined by its *initialState*, a *nextState* function that given a *State* and some *Inputs* emitted during a time unit computer the *State* at the next time unit, and a *render* function to display *States* to the users. All *States* and *Inputs* objects must be immutable, and *nextState* must be a pure function. User inputs are transmitted to an *Engine* via the function returned by *futureAct*, and *triggerRendering* should be called whenever the hardware is able to display the current *State*, at most every 1/60th of seconds. Finally, an *Engine* excepts a *connection* to communicate in broadcast with all the concerned peers.

3.3 Architecture and Implementation

We now give a quick overview of the architecture and implementation of the scala-lag-comp framework. The *Engine* interface presented in section 3.2 is composed of two stateful components: *ClockSync* and *StateLoop*. *ClockSync* is responsible for the initial attribution of peer *identity*, and the agreement on a *globalTime*, synchronized among all peers. *StateLoop* stores all the peer *Inputs* and is able to predict the application state at a certain time with the current inputs. Figure 3.2 shows the interconnection between the components of an *Engine*. The *triggerRendering* function of *Engine* gets the current *globalTime* from the *ClockSync*, ask the *StateLoop* to predict the *State* at that time, and passes the output for actual rendering to the *render* function. Wherever an *Input* is sent to the *Engine* via *futureAct*, this *Input* is combined with the *globalTime* and peer identity to form an *Event*. This *Event* is then directly transmitted to the local *StateLoop*, and via the broadcast connection to all the remote *StateLoops*.

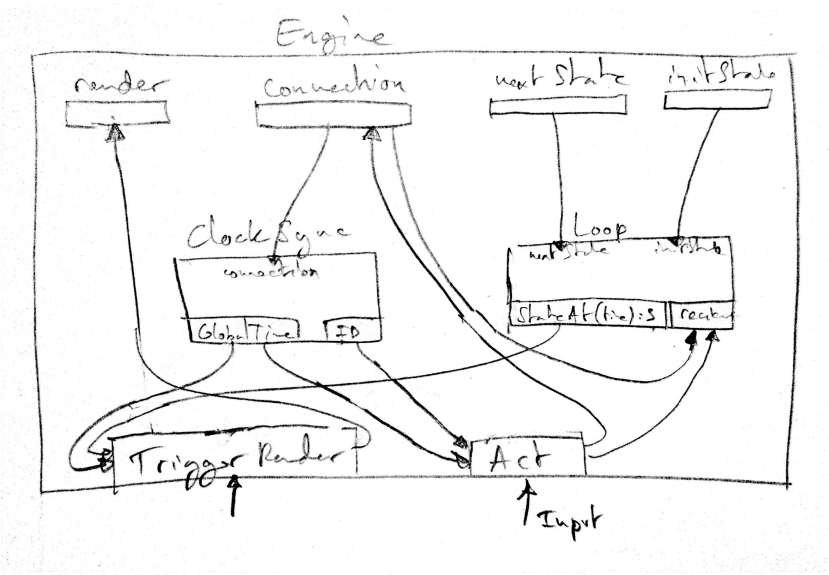


Figure 3.2: Overview the architecture of the latency compensation framework.

It should be noted that the scala-lag-comp framework is written in pure Scala, and that thus be cross compile to both Java bytecode and JavaScript.

3.3.1 ClockSync

The first action undertaken by *ClockSync* is to send *Greeting* message in broadcast, and listen for other *Greetings* during a small time window. Peer membership and identity are determined from these messages. Each of contains a large randomly generated number which is used to sort peers globally to attribute them a consistent identity.

Once all peers are aware of each other, they need to agree of a *globalTime*. Ultimately, each peer holds a difference of time between it's internal clock and the globally consented clock. The global clock is defined to be the arithmetic average between all the peer's clock. In order to determine the proper time difference, each pair of peers needs exchange their clock values. This is accomplished in a way similar to Cristian's algorithm [2]. Firstly, peers send request for the clock values of other peers. Upon receiving a response containing a time t, the current value of the remote clock can be estimated by adding half of the request round trip time to t. To minimize the impact of network variations, several requests are emitted between each pair of peers, and the algorithms only uses the requests with the shortest round trip times.

3.3.2 StateLoop

The *StateLoop* component is responsible for the maintenance of *Events* received by the framework, and for making predictions of the application *State* for the *Events* received so far.

- sorted, immutable list of *Events* (event = input, time, peer).
- The *stateAt* uses a recursive *computeState* function to computes the current state by applying the *nextState* function with the appropriate *Events* for each time unit, starting from the *initialState*. In essence, *StateLoop* does not need to store anything beside a list *Events*, from which the the current state can be derived.
- expensive
- Uses a cache to memorization on the *computeState* function. The *computeState* takes are argument a *time* and an immutable list of *Events* (sorted, and appending before *time*), and returns a state. The memorization is implemented using a bounded cache, retrains the association between a pair of time, list of *Events*, and a *State*. In the most common case when *stateAt* and not new inputs have been received since the last call, the cache will hit right away,

after a single recursive call. Whenever a remote input is received and inserted into the list of inputs, the recursion will take place up to the time at which this newly received input was issued.
- Timing assumptions allows to use implement the cache using fixed size array. Indeed Because of the implementation of immutable linked lists in Scala, the checking that a

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
- 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
- [8]
- Cheating concerns

Chapter 5

Conclusion and Future Work

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

Appendix A

Scala Futures and Promises

TODO: Futures provide a nice way to reason about performing many operations in parallel– in an efficient and non-blocking way. The idea is simple, a Future is a sort of a placeholder object that you can create for a result that does not yet exist. Generally, the result of the Future is computed concurrently and can be later collected. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.

By default, futures and promises are non-blocking, making use of callbacks instead of typical blocking operations. To simplify the use of callbacks both syntactically and conceptually, Scala provides combinators such as flatMap, foreach, and filter used to compose futures in a non-blocking way. Blocking is still possible - for cases where it is absolutely necessary, futures can be blocked on (although this is discouraged).

Appendix B

React

TODO: React [4] 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] IEEE standard for distributed interactive simulation - application protocols. *IEEE Std 1278.1-1995*. 15

[2] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989. 19

[3] S. Doeraene. scala-js-actors. http://github.com/sjrd/scala-js-actors. 10, 11

[4] Facebook. React. http://facebook.github.io/react/. 24

[5] L. Haoyi. Autowire. http://github.com/lihaoyi/autowire. 10

[6] L. Haoyi. upickle. http://github.com/lihaoyi/upickle. 12

[7] F. Muccio. play2-sockjs. http://github.com/fdimuccio/play2-sockjs. 9

[8] 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. 14, 21

[9] 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. 15

[10] 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. 15, 17

[11] TypeSafe. Akka. http://akka.io/. 10

[12] Valve. Source multiplayer networking. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. 15, 16