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 master project explores the possibilities for enhancement of the Scala.js networking capabilities. JavaScript provides many different API for communication, which while being usable from Scala.js, are non-idiomatic and require substantial integration work. We built the scala-js-transport library to simplify the communication between Scala/Scala.js systems. Featuring a single uniform communication interface, the library allows to build cross platform components, agnostic of a specific transport mechanism, to later plug any of the available implementation for the targeted protocol and platform. We illustrate the possibilities of this library by implementing an online multi player game, available for both desktop and web browser. The game is built around a latency compensation framework using prediction and roll-back mechanisms to provide a responsive user experience.

Contents

Chapter 1

Introduction

Scala.js is a Scala to JavaScript compiler. As such, it allows web applications to be written entirely in Scala. Using a single language on both the server and the client side simplifies the interactions between the two programing environments, and allows data structure and algorithms to be shared transparently.

However, a significant obstacle encountered in the implementation of web applications is not directly addressed by Scala.js: the client-server communication.

Scala.js offers very good interoperability with JavaScript, meaning that existing JavaScript solutions can be used *as is* in Scala.js. One could very well do with Scala/Scala.js what is commonly done today for client-server communication, use a JSON serialization library on the non-JavaScript side to make the data understandable by JavaScript. Doing so would reduce the benefits of sharing native Scala data structure and would result in a lot of boilerplate code with two completely disjoint implementations of the communication infrastructures.

Modern web browsers expose of a multitude of network protocols and techniques such as Ajax, Server-Sent Events, WebSocket, and most recently WebRTC for peer to peer communication. Choosing the right technology is essentially making a trade off between performances, compatibility and easy of use. However, the inherent API differences makes non trivial the switch from one technology to another, and prevent quick experiments with the different technologies.

Another obstacle to the development of client-server communication interfaces comes from the disparities of network programing model. One might be willing to use an abstraction built on top of connections, such as remote procedure calls, the actor model or communication via streams of data. Previous work in this direction used macros to provide cross platform remote procedure calls [1], or implemented an actor system in Scala.js able to transparently collaborate with an Akka-based backend [2]. Because both of these efforts are agnostic of the transport mechanism, developers are left with the responsibility of doing the integration with a particular technology.

1.1 Contributions and Overview

Introducing scala-js-transport: a networking library for simple communication between Scala systems running on Java virtual machines and on JavaScript engines. This library fills the gap between the various network protocols supported by modern web browsers, and the high level, idiomatic interfaces that are convenient to build distributed applications. The remainder of the project is dedicated to an example of usage of scala-js-transport; a remake of a Commodore 64 game, augmented with online multiplayer features. Our contributions can be summarized as follows:

- We introduce the scala-js-transport library and the various technologies and network programing models it supports. The library is built around the *Transport* interface, a trait unifying the communication capabilities of clients and servers. We discuss the different available implementations of the interface, targeting both the Java virtual machine and JavaScript, and two *Transport* wrappers featuring remote procedure calls and the actor model on top of the various implementations.

- We present the scala-lag-comp framework, a general purpose, cross platform framework for latency compensation applications. Designed for peer to peer environments, the framework implement a predictive latency compensation algorithm to offer maximal responsiveness with a guaranty of eventual consistency. Thanks to a clever use of immutable data structures and an optimal caching policy, the implementation is both computational and memory efficient.

- We show how we implemented Survivor, a cross platform, multiplayer game featuring real-time interactions with latency compensation. The game runs at about 60 frames per seconds, and, thanks to the design of the scala-lag-comp framework, provides an online experience comparable to modern multiplayer games. This example demonstrates the potential of Scala.js and of the scala-js-transport library to be used in latency sensitive applications.

The source code of the software developed for this project is available online in open source (MIT) on the project repositories[1][2][3].

[1] http://github.com/OlivierBlanvillain/scala-js-transport
[2] http://github.com/OlivierBlanvillain/scala-lag-comp
[3] http://github.com/OlivierBlanvillain/survivor

Chapter 2

Transport

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 it simply delegates tasks to the actual implementation, without adding new functionalities. Thanks to support of *futures* and *promises* in Scala.js, the interface cross compiles to both Java bytecode and JavaScript.

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, *Addresses* are abstract. As we will see later, it varies greatly from one technology to another.

ConnectionHandle represents an open 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 in *ConnectionHandle* is achieved by completing a *promise* of *MessageListener*.

An example of direct usage of the *Transport* interface is presented in Listing 2.2. This example implements a simple WebSocket client that sends "Hello World!" to a WebSocket echo server. After instantiating the *Transport* and declaring the *Address* of the server, *transport.connect* initiate the WebSocket connection and returns a *future* of *ConnectionHandle*. This *future* will be successfully completed upon connection establishment, or result in a failure if an error occurs during the process. In the successful case, the callback is given a *ConnectionHandle* object, which is

```
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.

used to *write* the "Hello World!" message, listening for incoming messages and *close* the connection.

The WebSocket echo server used in Listing 2.2 has a very simple behavior: received messages are immediately sent back to their author. Listing 2.3 shows a possible implementation of an echo server with the *Transport* interface. The body of this example is wrapped in a *try-finally* block to ensure the proper shutdown of the server once the program terminate. In order to listen for incoming connections, one must use *transport.listen()* which returns a *future* connection listener *promise*. If the underlying implementation is able to listen for new WebSocket connections on the given address and port, the *future* will be successful, and the *promise* can then be completed with a connection listener.

In addition to the examples of usage presented in Listing 2.1 and Listing 2.2, Section 2.3 contains examples of implementations using remote procedure calls and the actor model.

2.2 Implementations

The scala-js-transport library contains several implementations of *Transports* for WebSocket [3], SockJS [4] and WebRTC [5]. This subsection briefly presents the different technologies. Table 2.1 summarizes the available *Transports* for each platform and technology.

```
val transport = new WebSocketClient()
val url = WebSocketUrl("ws://echo.websocket.org")
val futureConnectionHandle = transport.connect(url)

futureConnectionHandle.foreach { connection =>
  connection.write("Hello WebSocket!")
  connection.handlerPromise.success { message =>
    print("Received: " + message)
    connection.close()
  }
}
```

Listing 2.2: Example of WebSocket client implementation.

```
val transport = new WebSocketServer(8080, "/ws")
try {
  transport.listen().foreach { _.success { connection =>
    connection.handlerPromise.success { message =>
      connection.write(message)
    }
  }}
} finally transport.shutdown()
```

Listing 2.3: Implementation of a WebSocket echo server.

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

Table 2.1: Summary of the available Transports.

2.2.1 WebSocket

WebSocket [3] 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 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. On the client side, the Tyrus library offers a standalone WebSocket client which is lightweight compared to the Netty framework.

2.2.2 SockJS

SockJS [4] is a WebSocket emulation protocol which fallbacks to different protocols when WebSocket is not available. Is supports a large number of techniques to emulate the sending of messages from server to client, such as Ajax long polling, Ajax streaming, Server-Sent Events and streaming content by slowly loading an Html file in an iframe. All 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 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 [6]. Netty developers have scheduled SockJS support for the next major release.

2.2.3 WebRTC

WebRTC [5] 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 which only supports TCP, WebRTC can be configured to use either TCP, UDP or SCTP.

```
val webSocketClient = new WebSocketClient()
val webRTCClient = new WebRTCClient()

val signalingChannel: Future[ConnectionHandle] =
  webSocketClient.connect(relayURL)

val p2pConnection: Future[ConnectionHandle] =
  signalingChannel.flatMap(webRTCClient.connect(_))
```

Listing 2.4: Example of WebRTC connection establishment using a dedicated relay server.

To connect via WebRTC, peers must have a way to exchange connection establishment information. This initial communicate medium, called a *SignalingChannel*, is not tight to a particular technology: its only requirement is to allow back an forth communication between the 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.

In term of the scala-js-transport library, this translates into having the *Address* of a *WebRTCClient* being a *SignalingChannel*, that is, a *ConnectionHandle* linking two peers. Listing 2.4 shows an example of WebRTC connection establishment using a dedicated WebSocket relay server.

In some cases, the client-server connections might need to be used for something other than the WebRTC connection establishment. The example of Listing 2.4 would not be sufficient because it assumes that the client-server connections are entirely dedicated to the relay of messages. To support these scenarios the scala-js-transport library contains picklers for *ConnectionHandle* objects and a *newConnectionsPair()* function. This function returns or pair *ConnectionHandle* objects that are linked one to the other: messages written in one connection will be received by the other, and *vice versa*. By sending the *ConnectionHandle* object returned by *newConnectionsPair()* to two different clients, a server can reuse its existing connections to these clients to establish a third, virtual connection, which acts as a direct link between the two clients.

We now quickly discuss the internal implementation of WebRTC connection establishment. As opposed the WebSocket and SockJS *Transports*, which have straightforward implementations because of the similarity between the *Transport* interface and the JavaScript API, adapting the JavaScript WebRTC API into the *Transport* interface showed in Listing 2.4 require some work. The sequence diagrams in Figure 2.1 and Figure 2.2 summarize the interactions between two *WebRTCClients*, their underlying *PeerConnections* (the main interface of the We-

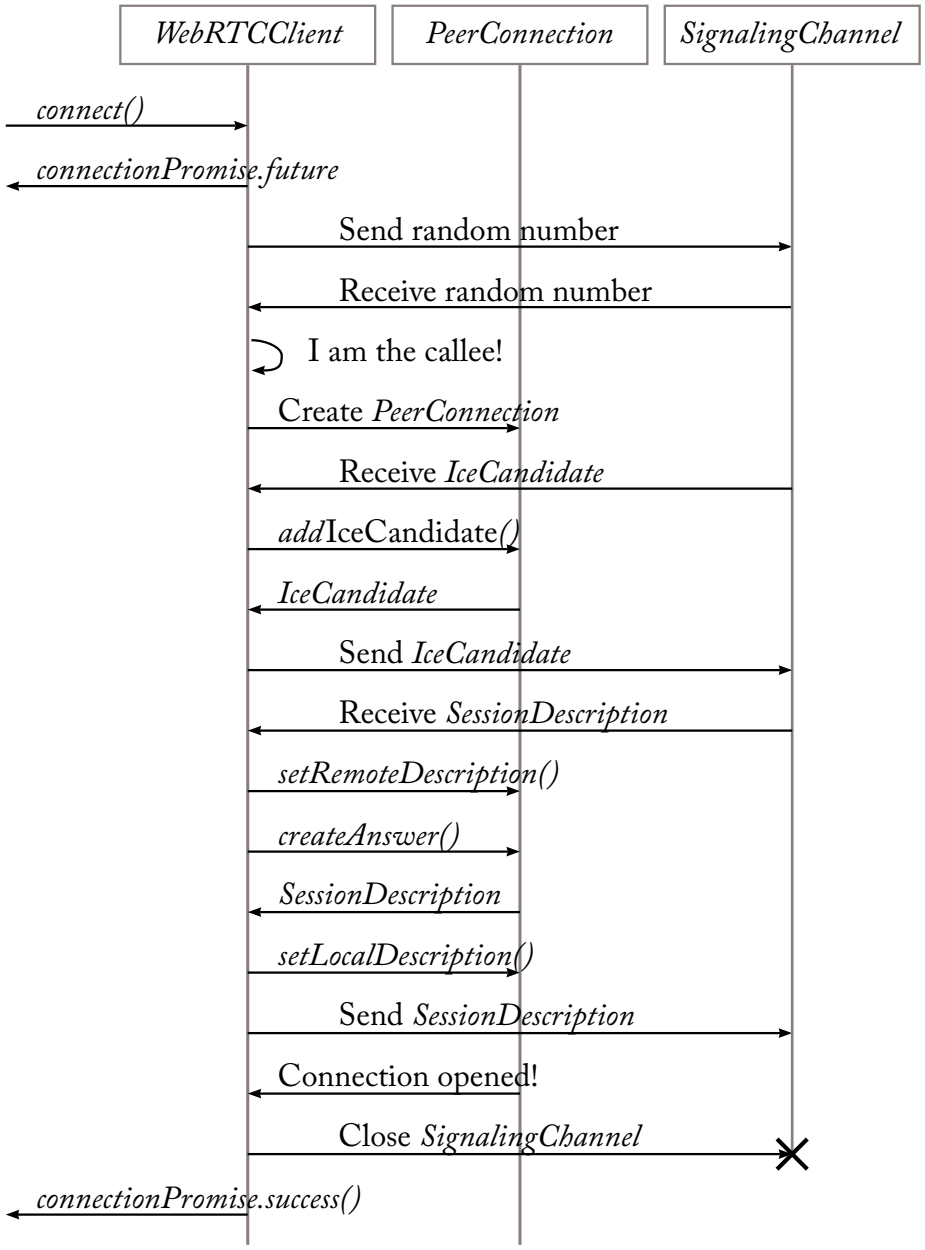

Figure 2.1: Sequence diagram of WebRTC establishment, callee point of view.

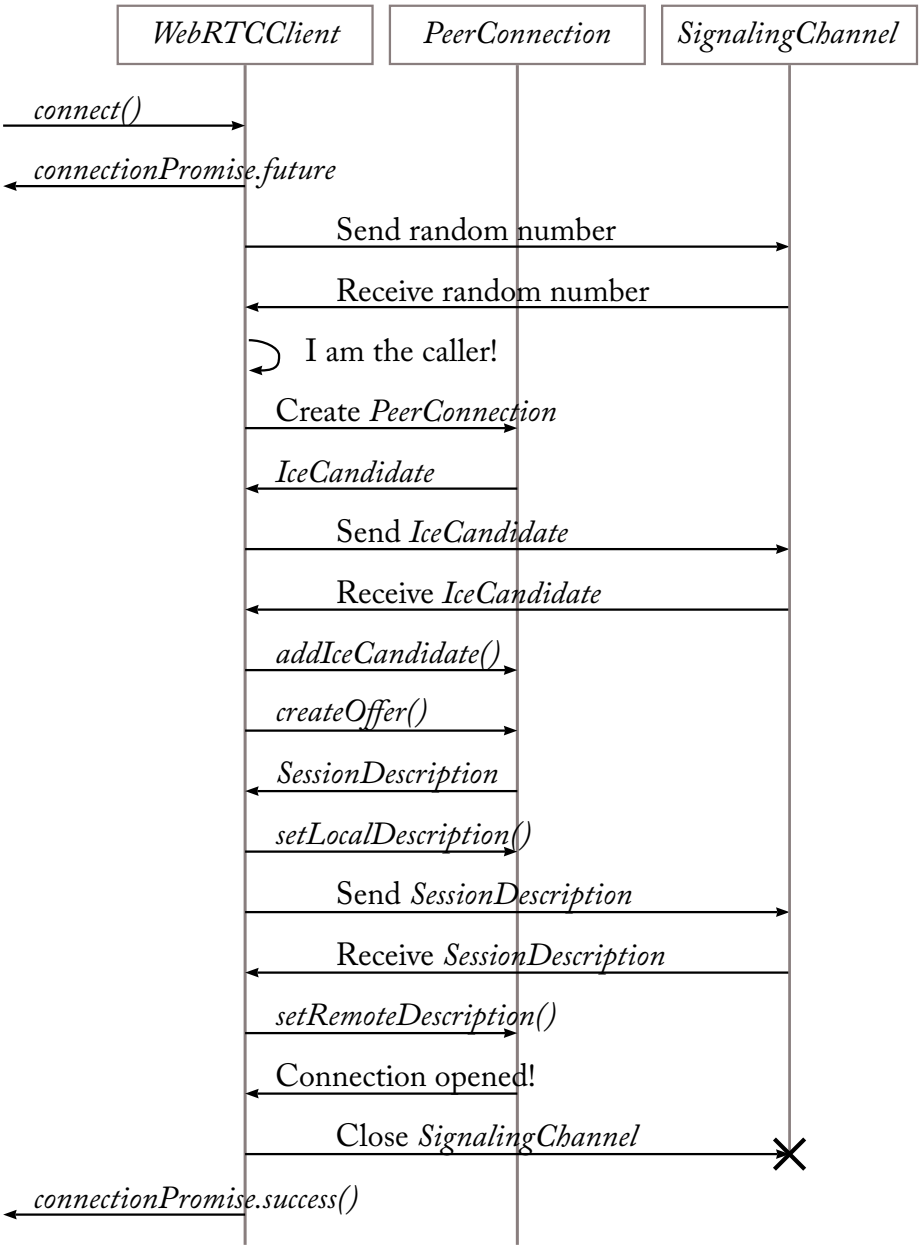

Figure 2.2: Sequence diagram of WebRTC establishment, caller point of view.

bRTC API), and the *SignalingChannel* providing the initial connection between the two peers.

The WebRTC protocol makes a distinction between the caller and callee. Because this distinction is of no interest in the scala-js-transport library, it is hidden to the library users via thanks to an additional step in the connection establishment. The first step of the establishment thus consists in an exchange of random numbers: the peer who generated smallest number is elected to be the caller (this step is repeated in case of equality). From here, peers needs to negotiate a way to achieve Network Address Translator (NAT) traversal. This negotiation is done through the Interactive Connectivity Establishment (ICE) protocol [7], which requires an exchange of *IceCandidate* messages through the *SignalingChannel*. Follows an exchange of offer and answer of *SessionDescription*, which is used to negotiate what media go through the connection [8]. Finally, when the peer to peer connection will become operational, the *WebRTCClient* closes the *SignalingChannel* connection and notifies its user by successfully completing the *connectionPromise* with the WebRTC connection.

In addition to the *WebRTCClient Transport*, the scala-js-transport library contains a *WebRTCClientFallback Transport*. This later implements some additional logic to detect WebRTC support, and automatically fall back to using the signaling channel as substitute for WebRTC when 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 implementation is available on the Node.js platform.

2.3 Wrappers

Thanks to the *Transport* interface, it is possible write programs with an abstract communication medium. We present two *Transport* wrappers, for Akka and Autowire [1], which allow to work with different network programing models. Because Autowire and Akka (via [2]) can both be used from both the JVM and JavaScript, these wrappers can be used to build cross compiling applications 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 JVM toolkit build around the actor model, was partly ported to Scala.js by

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

Listing 2.5: Transport wrappers to handle connections with actors.

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

Listing 2.6: Example of a connection handling actor.

S. Doeraene in [2]. The communication interface implemented in [2] was revisited in the *Transport* wrapper presented in Listing 2.5.

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 through the connection, and messages received by the *handler* actor are received messages incoming 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 termination of the connection, and killing the *handler* actor results in closing the connection. Listing 2.6 shows an example of a simple *handler* actor which sends back whatever it receives in uppercase.

Thanks to the picking mechanism developed in [2], it is possible to sent messages of any type through a connection, given that implicit picklers for these types of messages have been registered. 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* through a connection, making them transparently usable from the other 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 systems. 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 procedure call on top of any of the available *Transports*. Listing 2.7 shows a complete remote procedure call implementation using WebSocket, and the uPickle [9] serialization library.

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 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 induces some draw backs. Contrary to the actor model which can be used to 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, which would be 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 for another 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 in scala-js-transport 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 web browsers, wich can be configured to be run with two different browsers in order to test their compatibility.

```
// Shared
trait Api {
  def doThing(i: Int, s: String): Seq[String]
}
class MyRpcWrapper[T <: Transport](t: T)(implicit ec: ExecutionContext)
     extends RpcWrapper[T, upickle.Reader, upickle.Writer](t: T)(ec) {
  def read[Result: upickle.Reader](p: String) = upickle.read[Result](p)
  def write[Result: upickle.Writer](r: Result) = upickle.write(r)
}

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

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

Listing 2.7: Complete example of remote procedure call implementation.

Chapter 3

Dealing with latency

Working with distributed systems introduces numerous challenges compared to 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 is transfered from one machine to another. Our discussion will be focused on issues related to latency.

3.1 Latency Compensation

When talking about latency sensitive application, the first examples coming to mind might be multiplayer video games. In order to provide a fun and immersive experience, real-time games have to *feel* responsive, they must offer sensations and interactions similar to the one experienced by a tennis player when he caches the ball, or by a Formula One driver when he drives his car at full speed. Techniques to compensate network latency are also used in online communication/collaboration tools such as *Google Docs*, where remote users can work on the same document as if they where sitting next to each other. 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 [10], the different techniques 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 is then possible

to make a prediction on the client side by computing a "throwaway" state using the latest local inputs, which is later replaced by the state given by the server as soon as it is received. Predictions techniques with a centralized server managing the application state are used in most *First Person Shooter* games, including recent titles built with the Source Engine [11]. Predictions are sometimes limited to certain type of objects and interactions, such as in the *Dead Reckoning* [12] technique that estimates 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 the variations of state) is too large to be frequently sent over the network. In this case, peers can directly exchange the user inputs and simultaneously simulate application with a fixed delay. Having a centralized server is not mandatory, and peer to peer configurations might be favored because of the reduce communication latency. Very often, the perceived latency can be diminished by instantly emitting a purely visual or sonorous feedback as soon as an input is entered. 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 [13].

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* [14] are an example of such techniques where the amount of delayed applied to world entities is proportional to their distance to the 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 or arrows [15]. 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 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. The framework cross compiles to run on both Java virtual machines and

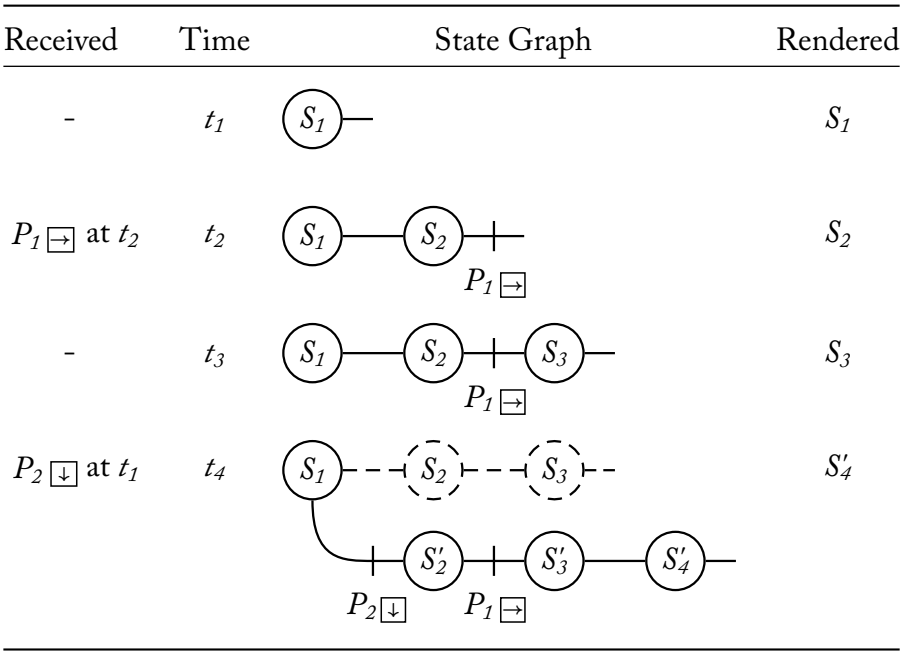

Figure 3.1: Growth of the state graph over time, from the point of view of P_1.

JavaScript engines, allowing to build applications targeting both platforms able to transparently collaborate.

By imposing a purely functional design to its users, scala-lag-comp 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 for prediction techniques, such as the one described in [11], our framework does not use any authoritative node to hold the global state of the application, and can therefore function in peer to peer, without single points of failure.

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 when it arrives at destination. 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 P_1. In this example, P_1 emits an input at time t_2. Then, at time t_3, P_1 receives an input from P_2 which was emitted at time t_1. At this point, the framework invalidates a branch of the state tree, S_2-S_3, and computes S'_2-S'_3-S'_4 to take into account both inputs.

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

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

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

Listing 3.1: Interface of the latency compensation framework.

By instantaneously applying local input, the application reactiveness is not affected by the quality of the connection; a user interacts with the application as he would if he was the only peer involved. This property comes with the price of having short periods of inconsistencies between the different peers. These inconsistencies last until all peers are aware of all inputs, at which point the simulation recovers its global unity.

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 would result in out-of-sync application states. [13] reports out-of-sync issues as one of the main difficulty they encountered during the development of multiplayer features. In our case, the *roll back* 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 would introduce serious bugs, which would be hard to reproduce and isolate.

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

An application is entirely defined by its *initState*, a *nextState* function that given a *State* and some *Actions* emitted during a time unit, computes the *State* at the next time unit, and a *render* function to display *States* to the users. *State* objects must be immutable, and *nextState* has to be a pure function. User *Inputs* are transmitted to an *Engine* via *futureAct*, and *triggerRendering* should be called whenever the platform is ready to display the current *State*, at most every 1/60th of seconds. Finally, an *Engine* excepts a *broadcastConnection* to communicate with all the participating peers.

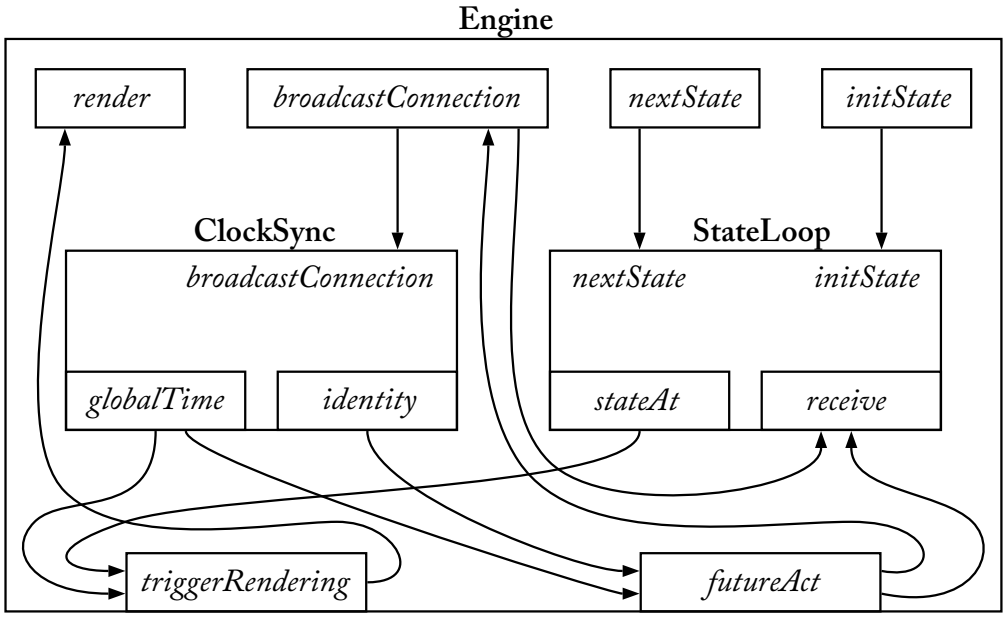

Figure 3.2: Architecture of the latency compensation framework.

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 establishment of a *globalTime*, synchronized among all peers. *StateLoop* stores all the peer *Inputs* and is able to predict the application state for the *Inputs* received so far. Figure 3.2 shows the interconnection between the components of an *Engine*. The *triggerRendering* function of the *Engine* gets the current *globalTime* from *ClockSync*, asks *StateLoop* to predict the *State* at that time, and passes the output to the user via the *render* function. Whenever an *Input* is sent to the *Engine* via *futureAct*, this *Input* is combined with the peer *identity* to form an *Action* and then couples with the *globalTime* to form an *Event*. This *Event* is directly transmitted to the local *StateLoop*, and sent via the connection to the remote *StateLoops*.

3.3.1 ClockSync

The first action undertaken by the *ClockSync* component is to send a *Greeting* message in broadcast, and listen for other *Greetings* message during a small time window. Peer membership and identity are determined from these messages. Each

Greeting message contains a randomly generated number which is used to order peers globally, and attribute them a consistent identity.

Once peers are all aware of each other, they need to agree on a *globalTime*. Ultimately, each peer holds a difference of time Δt between its internal clock and the globally consented clock. The global clock is defined to be the arithmetic average of all the peers' clocks. In order to compute their Δt, each pair of peers needs to exchange their clock values. This is accomplished in a way similar to Cristian's algorithm [16]. First, peers send request for the clock values of other peers. Upon receiving a response containing a time t, one can estimate the value of the remote clock by adding half of the request round trip time to t. Once peers all have estimations of the various clocks, they are able to locally compute the average, and use it as the estimated *globalTime*. To minimize the impact of network variations, several requests are emitted between each pair of peers, and the algorithms only retain the requests with the shortest round trip times.

Certainly, this approach will result in slightly shifted views of the *globalTime*. Even with more elaborated solutions, such as the Network Time Protocol, the average precision varies between 20 ms and 100 ms depending on the quality of the connection [17]. In the case of the scala-lag-comp framework, out-of-sync clocks can decrease the quality of the user experience but do not effect correctness. Indeed, once every user has seen every input, and once all the simulations have reached the *globalTime* at which the latest input was issued, all the simulations will generate the same *States* for the same *globalTimes*. If one user is significantly ahead of the others, this will have the effect of preventing him to react quickly to other peers actions. Suppose a peer P_a thinks the *globalTime* is t_a seconds ahead of what other peers believe. Whenever he receives an input issued at time t_1, P_a will already have simulated and displayed the application up to time $t_1 + t_a + networkdelay$, and his reaction to this input will be issued with a lag of $t_a + networkdelay$. Furthermore, being ahead of the actual *globalTime* implies that the *rolls back* mechanism will be used for significant portions of time, introducing visual glitches such as avatars teleporting from one point to another.

To prevent malicious manipulations of the clock, a form of cheating in games, one could improve the framework by adding a mechanism to verify that the issue times of inputs respects some causal consistency. Follows an example of such mechanism.

Suppose that every message m_i includes a random number r_i. Then, newly emitted inputs could include the latest random number generated locally, and the latest random number received from other peers. Adding this information would allow the detection of malicious peers. Indeed, if a malicious peer P_1 pretends that his message m_1 is issued one second later that it is in reality, and P_2 sends a message m_2 that can by estimated to arrive before m_1 was issues, then P_2 can detect that P_1 is malicious by checking that m_1 does not contain r_2. Similarly, pretending that a

message is issued in the past would break the sequence of "latest random number generated locally", and thus be detectable.

3.3.2 StateLoop

The *StateLoop* component implements the heart of the prediction algorithm: a *stateAt* function which, given a *time*, computes a prediction of the application *State*. To do so, *StateLoop* maintains a set of user *Actions* received so far, which is used to simulate the execution of the application. *Actions* are stored in an immutable list of *Events* (pairs of *Action* and *time*), sorted by *time*.

Semantically, every call to the *stateAt* function implies a recursive computation of the current state. This recursion starts from the *initState*, and successively applies the *nextState* function with the appropriate *Events* for each time unit until the process reaches the current time.

Doing the complete recursion on every frame would be too expensive. However, since this process uses a pure function, called *computeState* and returning a *State* given a time and list of *Events* happening before that time, it can be made efficient using memoization. Indeed, in the most common case when *stateAt* is called and no new *Events* has been received since the last call, the cache hits right away, after a single recursion step. Whenever a remote input is received and inserted into the sorted list of *Events*, the recursion takes place up to the time at which this newly received *Event* was issued. The cache then hit at that point, from where *nextState* can be successively applied to obtain the *State* for the current time. This is how the *rolls back* mechanism illustrated in Figure 3.1 is implemented in a computationally efficient manner.

Regarding memory management, timing assumptions on the network allows the use of a bounded cache. Indeed, if we consider a peer to be disconnected when one of his messages takes more than one second to transit over the network, it is sufficient to retain history of *States* for a period of one second. Thus, the memorization can be implemented using a fixed size array, retaining the association between a time, a list of *Events*, and a *State*. Thanks to a careful use of immutable lists to store *Events*, querying the cache can be done using reference equality for maximum efficiency.

Chapter 4

A Real-Time Multiplayer Game

There seems to be a tradition of using Scala.js to build games. Indeed, at the time of writing, half of the projects listed on the official web site of the project are video games. What could be better than a multiplayer game to showcase the scala-js-transport library?

4.1 Scala Remake of a Commodore 64 Game

To cope with time constraints and stay focused on the project topic, the decision was made to start working from an existing game. Out of the list of open source games published on GitHub [18], Survivor [19] appears to be the most suitable for the addition of real-time multiplayer features.

The original version of the game was written in 1982 for the Atari 2600. One year later, a remake with better graphics is released for the Commodore 64. Recently, S. Schiller developed an open source remake of the game using Html, CSS and JavaScript [19]. This latest open source remake served as a basis for the version of the game presented in this chapter. We rewrote the code from scratch in Scala to follow the functional programing style required by the scala-lag-comp framework, but still rely on the visual assets created by S. Schiller.

4.2 Architecture

The Scala remake of Survivor puts together the scala-js-transport library and the scala-lag-comp framework into a cross platform, real time multiplayer game. On the networking side, it uses WebRTC if available, and fallbacks to WebSocket otherwise.

Every aspect of the game logic is written in pure Scala, and is cross compiled for both JVM and JavaScript engines. Some I/O related code had to be written

specifically for each platform, such as the handling of keyboard events and of rendering requests. The JavaScript implementation is using the DOM APIs, and the JVM implementation is built on top of the JavaFX/ScalaFX platform. On the JavaScript, side rendering requests are issued with *requestAnimationFrame*, which saves battery and CPU usage by only requesting rendering when the page is visible to the user.

In order to reuse the visual assets from [19], the JVM version embeds a full WebKit browser and runs the same implementation of the user interface than the JavaScript version. The rendering on the JVM goes as follows. It begins with the *render* function being called with a *State* to display. This *State* is serialized using the uPickle serialization library [9], and passed to the embedded web browser as the argument of a *renderString* function. This function, defined in the Scala.js with a *@JSExport* annotation to be visible to the outside word, deserializes its argument back into a *State*, this time on the JavaScript engine. With this trick, a *State* can be transfered from a JVM to a JavaScript engine, allowing the implementation of the user interface to be shared between two platforms. While sufficient for a proof of concept, this approach reduces the performances of the JVM version of the game, which could be avoided with an actual rewrite of the user interface on top of JavaFX/ScalaFX.

4.3 Functional Graphic User Interface With React

In this section we discuss the implementation of the graphics user interface of our game using the React library [20]. In functional programing, the *de facto* standard for building graphics user interface seems to be functional reactive programing. React enables a different approach[1], which suits perfectly the architecture of the scala-lag-comp framework.

The interface is a single *render* function, which takes as argument the entire state of the application, and returns a complete Html representation of the state as a result.

The key performance aspect of React is that it does directly uses the DOM returned by the *render* function. Instead of replacing the content of the page whenever a new DOM is computed, React computes a diff between this new DOM and the currently rendered DOM. This diff is then used to send the minimal set of mutations to the browser, thereby minimizing rendering time.

[1] React supports a variety of architectures to build user interfaces, and is not limited the approach described in this section. For example, it supports the storage of mutable state into *Components*. React also supports server side rendering. By sharing the definition of *Components* between client and server side, rendering can take place on the server, thus limiting the client work to reception and application of diffs.

In order to lighten the diff computation, React uses *Components*, small building block to define the *render* function. A *Component* is essentially a fraction of the *render* function, that given a subset of the application state, returns an Html representation for this subset of the application. *Components* can be composed into a tree that takes the complete application state at its root, and propagates the necessary elements of states through the tree, thereby forming the complete rendering function of the application. Thanks to this subdivision into *Components*, React is able to optimize the diff operation by skipping branches of the Html DOM corresponding to *Components* that depend on subsets of the state which is unchanged since the last rendering.

There is a special optimization possible when working with immutable data structure, as it is the case with the scala-lag-comp framework. This optimization consists in overriding React's method for dirty checking *Components*. Instead of using deep equality on subsets of states to determine if a *Component* needs to be re-rendered, one can use reference equality, which is a sufficient condition for equality on immutable data structures.

4.4 Experimental Results

On modern laptop[2], the game runs at 60 Frames Per Second (FPS) on both Google Chrome and the OpenJDK 8 Java virtual machine. The game feels perfectly smooth on the JavaScript version, and shows some FPS drops on the Java virtual machine when too many objects are present on the field, probably due to overhead of communication between WebKit and JavaFx. A slight "warm up" effect is also noticeable on the Java virtual machine.

In order to experience the effects of latency compensation, we set up simple WebSocket server on Heroku, located in the United States for a higher latency. The result is quite noticeable in the other player trajectory which has small "jumps" when the remote player changes direction.

In Figure 4.1 we illustrate a situation of latency compensation, reproducible by controlling a single player. The sequence of action goes from top to bottom, the left hand side of the screenshots is the view the moving player P_m, and the right hand side is the view of the immobile player P_i. Because P_i is not emitting any input, P_m knows all the information of the game as soon as it is emitted, and does not experience any latency compensation. In this example, P_m is moving strait up towards P_i. On frame f_2 of P_m's view, we can see that P_m stops thrusting, noticeable by the dissipation of the halo surrounding his ship. On frames f_2 to f_4 of P_m's view, his ship slowly decelerating to finally stop right in front of P_i's ship.

[2]Intel Core i7-2620M @ 2.69 GHz, using the integrated graphics card.

From the point of view of P_i, we see that his game it not immediately aware that P_m is decelerating, a manifestation of the delay required for a packet to do a round trip to the United States. On frame f_4, we see that P_i detected a collision, and initiated an explosion animation. Between frame f_4 and f_5, P_i revives the message indicating that P_m actually interrupted his acceleration a few moments ago. At this point, the roll-back mechanism enters into action to take into account the last input from P_m, which results in the situation in frame f_5, where both games are in the same state.

Readers are invited to try out Survivor to get their own impressions on the smoothness and responsiveness of the game. The JavaScript executable of the game is available online[3], configures to connection the demo server located in the United States. Open the game twice to simulate a second player, and start moving around and shooting with arrow keys and the space bar.

[3] http://olivierblanvillain.github.io/survivor

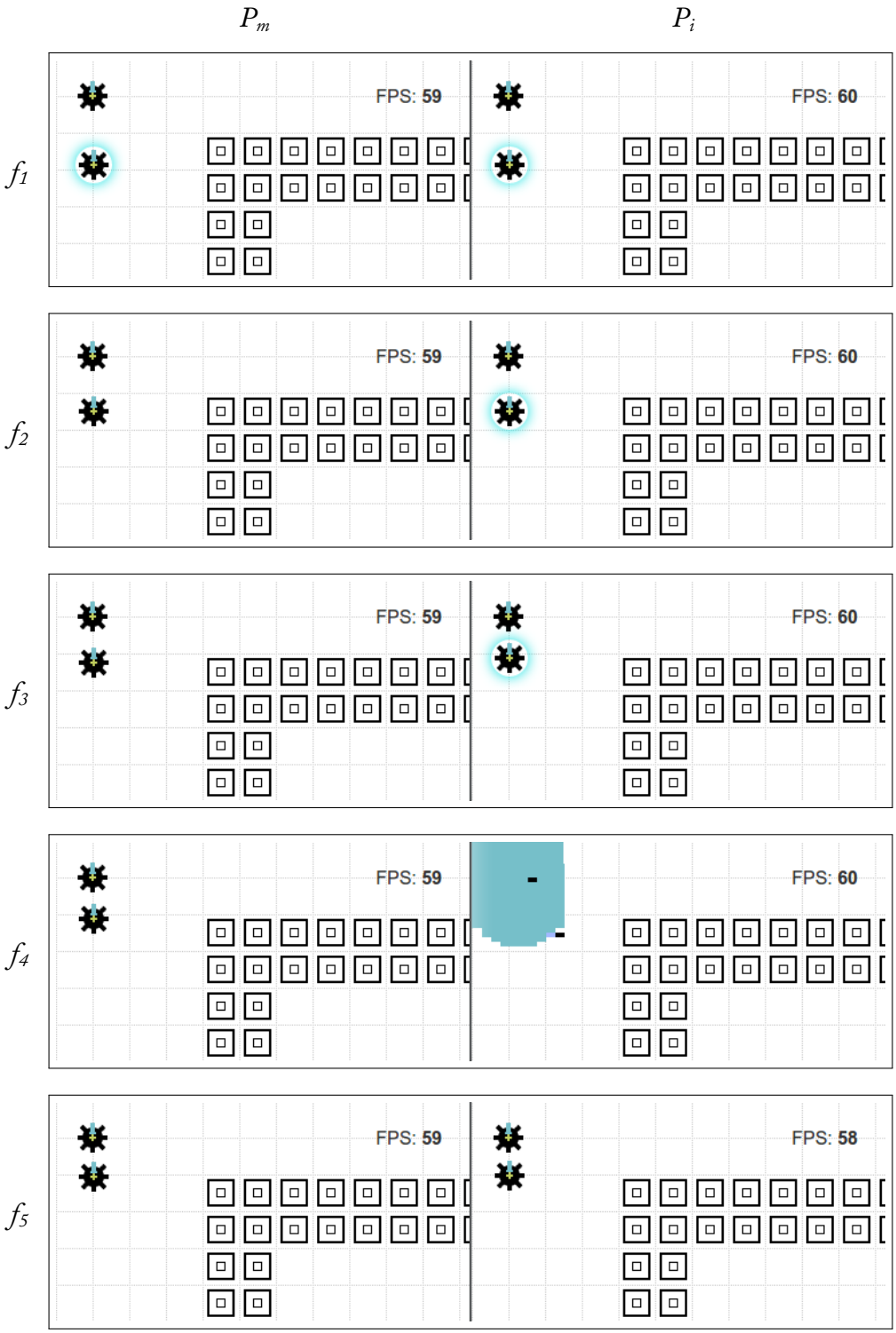

Figure 4.1: Latency compensation in action.

Chapter 5

Related Work

5.1 Network Libraries

The Node.js platform gained a lot of popularity in the recent years. By enabling server-side of applications to be written in JavaScript, it allows data structure and API can be shared between client and server. In the case of network capabilities, many JavaScript libraries imitate the WebSocket API and rely on *duck typing* to share code between client and server. For example, the *ws* library [21] is an implantation of WebSocket client and server for Node.js which provides *ws* objects behaving exactly like *WebSocket* objects do on the browser side. Similarly, SockJS clients (discussed in subsection 2.2.2) provide *SockJS* objects that are almost drop in replacement for *WebSocket* objects. Finally, we could also mention the official WebRTC API [5], which was designed such that its "API for sending and receiving data models the behavior of WebSockets".

ClojureScript, the official Clojure to JavaScript compiler, has a large ecosystem of libraries, out of which *Sente* [22] seems to be the most popular library for network communication. It goals are similar to those of scala-js-transport: offer a uniform client-server API supporting several transport mechanism. Instead of using an existing WebSocket emulation library, *Sente* implements its own solution to fallback on Ajax and long-polling when WebSocket is not available.

With the large number of languages that compile to JavaScript [23], an exhaustive coverage of the network libraries would be beyond the scope of this report. To the best of our knowledge, scala-js-transport is the first library offering such variety of supported protocols and platform.

5.2 Latency Compensation Engines

Mainstream real-time strategy game commonly rely on simultaneous simulation to implement mutliplayer over the Internet. While little to no information is made public by the game publishers, experimenting with these games brings information about their implementations. For example, in *Starcraft II* every input is executed with a minimum delay of 200 ms, which is in most cases sufficient to transfer the input over the network and have it processed simultaneously. However, because of fluctuation on the network or lack of processing power in one of the peers, this delay might not be sufficient. To recover the situation, the *Starcraft* game clients first rely on a small pause to hold every simulation until the slowest client catches up, and then globally increases the input delay. The delay is then dynamically adjusted during the remaining of the game, aiming to get back at 200 ms. The absence of any prediction mechanism forces this delay to be identical for every peer, thus being as slow as the slowest player.

The peer to peer model also raises several challenges in terms of security and cheating prevention. In addition to the potential for malicious manipulation of clocks, discussed in subsection 3.3.1, the absence of authoritative server makes it non trivial to hide information between peers. Many strategy games implement a mechanism of *fog of war*, which hides the areas of the map that are out of sight of a player units. Because every player performs a full simulation of the game, cheaters can modify their game clients to include a *map hack* in order to reveal the totality of the map. The work of [24] presents a generic and semi automatic tool to build *map hacks*, which was proven to be effective for several of the most recent strategy games. By repetitively taking snapshots of the memory allocated by a game, the tool can identify areas corresponding to the active units and building. Because this attack does not make any modification on the game but simply passively inspects the memory, it is completely undetectable. In addition to their tool, [24] proposes a new approach to implement hidden information in peer to peer configuration. In their solution peers share the minimum amount of information about the game in order to prevent any forms of *map hack*. They use a cryptographic protocol, of the category of oblivious intersection protocols, that allows peers to *negotiate* what information should be shared without relying on a third party entity.

In order to better handle the complexity of latency compensation algorithms, the authors of [10] suggest the use of a new programing model called *TimeLine*. The motivation from this model comes from the observation that dealing with both time and shared state is the main source of complexity for latency compensation algorithms. The *TimeLine* model was designed to facilitates the explicit treatment of time. This model was used to formulate several latency compensation algorithms, which are implemented as part of the *Janus* toolkit. In a sense, the approach of [10] is orthogonal to the one taken in scala-lag-comp, where we rely

on functional programing principles to avoid direct management of state by the framework user.

5.3 Functional Programing In Games

In a blog post series entitled *Purely Functional Retrogames* [25], James Hague discusses a functional implementation of the *Pac-Man* game. He explains that the amount of state to manage can be reduced by computing some parts of it as function of others. For instance, the blinking of pills does not need any storage as it can be deduced from the current time. The author advocates an architecture similar to the one we used in the Scala remake of Survivor: a top level loop executed on every frame gathers user inputs, computes the next state and displays this state to the user. In order to modularize the next state function, one could avoid returning update versions of game entities on every function by instead returning *events*. These *events*, which are essentially descriptions of state changes, can be handled in a separated step.

Torus Pong is a multiplayer remake of Pong. Implemented over the course of two days during the Clojure Cup, *Torus Pong* demonstrates how functional programing can be used to quickly build games [26]. The core of the game is a pure function to compute the state from the previous state and a set of commands. This function resides on the server which orchestrates the entirety of the game. Clients serve as simple terminal: they send local inputs to the server and render the game whenever a new state is received from the server. Every communication is asynchronous and transits over WebSocket connections.

No Man's Sky is an upcoming science fiction game set in an infinite, *procedurally* generated universe. A recent video interview [27] reveals the core mechanism of this game: every planet in *No Man's Sky* in entirely generated by a pure function, taking as input the id of the planet (a 64 bit hash) and the current time. Because everything is generated lazily, the game is able to offer a virtually infinite world without storing any information. In contrast to randomly generated worlds, players will be able to come back to a planet they liked to see how it evolved over time, and share their findings with friends.

Chapter 6

Conclusion and Future Work

In this report, we presented the scala-js-transport library which simplifies Scala.js networking. The library supports several communication technologies over different platforms, and allows its users to develop in various network programing models. In addition, it also sets the basics for the creation of cross platform networking utilities. Our work includes an example of such utility: a predictive latency compensation framework for peer to peer applications. The framework takes great advantage of the functional programing capabilities of Scala to allow the elaboration of easy to understand and easy to maintain applications. The project is concluded with the remake of the retrogame called Survivor, which puts together the scala-js-transport library and the scala-lag-comp framework to provide online, real-time mutiplayer features.

Future work could investigate the support of Web Workers [28] in the scala-js-transport library. Web Workers allow parallel computations on web browsers, and as such, are not directly related to networking. However, in their current implementations, Web Workers communicate by message passing where messages are copied from one Worker to another using *structured cloning*, thus being similar to a communication over the network.

Another possible direction would be to add communication via streams of data as another network programing model. The Reactive Stream initiative [29] defines a standard for stream processing with non-blocking back pressure. Being currently targeted at Java virtual machines, the project could benefit from an implementation for JavaScript engines, which would have its place as part of the scala-js-transport library.

References

[1] Haoyi Li. Autowire. http://github.com/lihaoyi/autowire. 5, 14

[2] Doeraene Sébastien. scala-js-actors. http://github.com/sjrd/scala-js-actors. 5, 14, 15

[3] Fette I. and Melnikov A. The websocket protocol. Technical report, Internet Engineering Task Force, 2011. http://tools.ietf.org/html/rfc6455. 8, 10

[4] Kahle Bryce and Majkowski Marek. SockJS - websocket emulation. http://github.com/fdimuccio/play2-sockjs. 8, 10

[5] Bergkvist Adam, Burnett Daniel C., Jennings Cullen, and Narayanan Anant. W3C editor's draft - WebRTC 1.0: Real-time communication between browsers. http://w3c.github.io/webrtc-pc/, 2014. 8, 10, 30

[6] Muccio Francesco. play2-sockjs. http://github.com/fdimuccio/play2-sockjs. 10

[7] Rosenberg J. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols. Technical report, Internet Engineering Task Force, 2010. http://tools.ietf.org/html/rfc5245. 14

[8] Rosenberg J. and Schulzrinne H. An offer/answer model with session description protocol (SDP). Technical report, Network Working Group, 2002. http://tools.ietf.org/html/rfc3264. 14

[9] Haoyi Li. uPickle. http://github.com/lihaoyi/upickle. 16, 26

[10] Cheryl Savery and T.C.Nicholas Graham. Timelines: simplifying the programming of lag compensation for the next generation of networked games. *Multimedia Systems*, 19(3):271--287, 2013. 18, 31

[11] Valve. Source multiplayer networking. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. 19, 20

[12] IEEE standard for distributed interactive simulation: application protocols. Technical report, New York, USA, 1996. 19

[13] Terrano Mark and Bettner Paul. 1500 archers on a 28.8: Network programming in age of empires and beyond. http://gamasutra.com/view/feature/3094, 2001. 19, 21

[14] 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. 19

[15] Jouni Smed and Harri Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006. 19

[16] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146--158, 1989. 23

[17] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2010. 23

[18] Reilly Lee. Games on github. http://github.com/leereilly/games. 25

[19] Schiller Scott. Survivor: Remaking a Commodore 64 game in Html. http://www.schillmania.com/content/entries/2012/survivor-c64-html-remake/, 2012. 25, 26

[20] Facebook. React. http://facebook.github.io/react/. 26

[21] Stangvik Einar Otto. ws: a node.js websocket library. http://github.com/einaros/ws. 30

[22] Taoussanis Peter. Sente, channel sockets for clojure. http://github.com/ptaoussanis/sente. 30

[23] List of languages that compile to JS. http://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. 30

[24] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. Openconflict: Preventing real time map hacks in online games. In *IEEE Symposium on Security and Privacy*, pages 506--520. IEEE Computer Society, 2011. 31

[25] Hague James. Purely functional retrogames. http://prog21.dadgum.com/23.html, 2008. 32

[26] Dahlén Ragnar. Clojure cup 2013, pong and core.async. http://ragnard.github.io/2013/10/01/clojurecup-pong-async, 2013. 32

[27] GameSpot. How does No Man's Sky actually work? Video interview, 10 min http://www.gamespot.com/videos/embed/6420148/, 2014. 32

[28] Hickson Ian. W3C candidate recommendation - Web Workers. http://www.w3.org/TR/workers/, 2012. 33

[29] Reactive streams. http://www.reactive-streams.org/. 33