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

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. Thanks to support of *futures* and *promises* in Scala.js, these interfaces cross compile 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, *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*.

An example of direct usage of the *Transport* interface is presented in Listing 2.2. This example implements a simple WebSocket client that sends a "Hello World!" message to a WebSocket echo server. After instantiating the *Transport* and declaring the *Address* of the server, *transport.connect* initiate the WebSocket connection

```
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.

and returns a *future* of *ConnectionHandle*. This is *future* will be successfully completed upon connection establishment, or result in a failure if an error occurred during the process. In the successful case, the callback is given a *ConnectionHandle* object, which is used to *write* the "Hello World!" message, handle incoming messages and *close* the connection.

```
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("Recived: " + message)
    connection.close()
  }
}
```

Listing 2.2: Example of WebSocket client implementation.

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 *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.

```
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.

In addition to the example of usage presented in Listing 2.1 and Listing 2.2, the scala-js-transport library provides two additional abstractions to express communication over the network. Section 2.3 contains examples of implementations using remote procedure calls and the actor model, which are available as wrappers around the *Transport* to prover higher level of abstraction.

2.2 Implementations

The scala-js-transport library contains several implementations of *Transports* for WebSocket, SockJS [1] and WebRTC [2]. 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	-

Platform	WebSocket	SockJS	WebRTC
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 [1] 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 [3]. Netty developers have scheduled SockJS support for the next major release.

2.2.3 WebRTC

WebRTC [2] 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 *Transports* for WebRTC, *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.

TODO: Add a sequence diagram and explain connection establishment.

2.3 Wrappers

By using *Transport* interface, it is possible write programs with an abstract communication medium. We present two *Transport* wrappers, for Akka and Autowire [4], which allow to work with different model of concurrency. Because Autowire and Akka (via [5]) 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 [5]. The communication interface implemented in [5] was revisited into the *Transport* wrapper presented in Listing 2.4.

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

Listing 2.4: 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 2.5 shows an example of a simple *handler* actor which than sending back whatever it receives in uppercase.

Thanks to the picking mechanism developed in [5], it is possible to sent messages of any type thought 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* thought a connection, making them transparently usable from the side of the connection as if they were references to local actors.

```
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 2.5: Example of a connection handling actor.

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 2.6 shows a complete remote procedure call implementation on top of WebSocket.

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.

```scala
// 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 2.6: Example of remote procedure call implementation.

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 is transfered from one machine to another. Our discussion will be focused on issues related to latency.

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, the must offer sensations and interactions similar to the one experienced by a tennis player when he caches the ball, or of a Formula One driver when he drives his car at full speed. . Techniques to compensate network latency also have uses 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 [7], 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 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 provided 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 [8]. Predictions are sometimes limited to certain type of objects and interactions, such as in the *dead reckoning* [9] 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 the variations of state) is 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 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 the an input is entered, but delaying the actual effects of the action to have it executed simultaneously on all peers. 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 [11] 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 [12]. 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. The framework cross compiles to run on both Java virtual machines and JavaScript engines, allowing to build applications targeting both platforms which can 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 [8], our framework does uses any authoritative node to hold the global state of the application, and can therefore functions 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 *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.

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 result in out-of-sync application states. [10] 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 in time* procedure introduces another source of potential mistake. Any mutation in a branch of

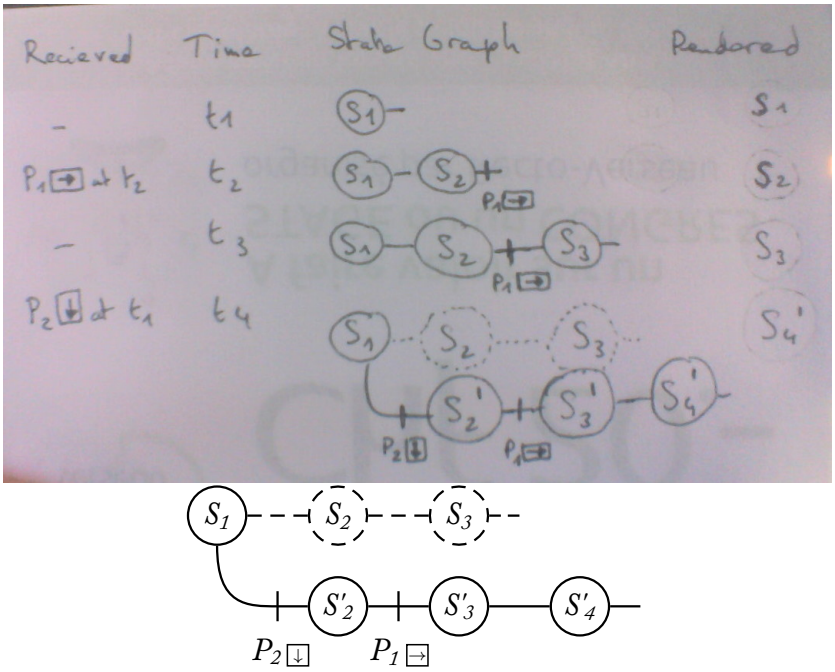

Figure 3.1: Growth of the state graph over time, from the point of view of *P1*.

the simulation that would not properly be canceled when rolling back to a previous state would induce serious bugs, of the hard to isolate 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*.

An application is entirely defined by its *initialState*, a *nextState* function that given a *State* and some *Actions* emitted during a time unit computer 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.

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

class Engine[Input, State](
    initialState: 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.

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 for the *Inputs* received so far. 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 to the user via the *render* function. Wherever an *Input* is sent to the *Engine* via *futureAct*, this *Input* is combined with the peer *identity* to form an *Action*, 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*. Ul-

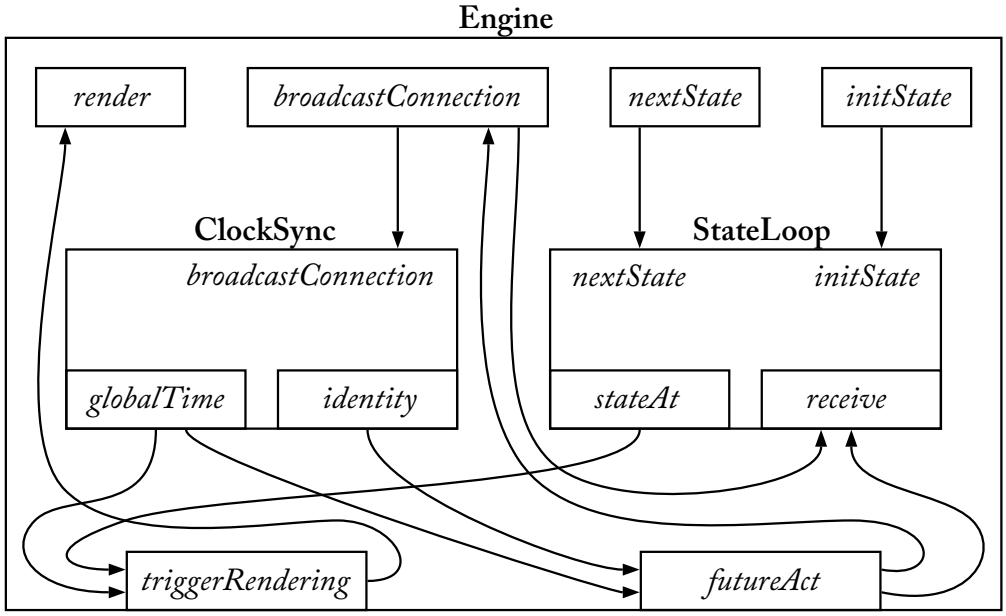

Figure 3.2: Overview the architecture of the latency compensation framework.

timately, each peer holds a difference of time Δt between it's internal clock and the globally consented clock. The global clock is defined to be the arithmetic average of all the peer's clock. In order to compute their Δt, each pair of peers needs exchange their clock values. This is accomplished in a way similar to Cristian's algorithm [13]. Firstly, 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. One all peers 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 retains the requests with the shortest round trip times.

Certainly, this approach will result in slightly shifted views of the *globalTime*. Even with more solutions elaborated, such as the Network Time Protocol, the average precision varies between 20 ms and 100 ms depending on the quality of the connection [14]. 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 generate *States* for the same *globalTime*. 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 peers P_a think 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 have already 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 potential 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 sent contains 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 *P1* pretends that his message m_1 is issued one second later that it is in reality, and *P2* sends a message m_2 that can by estimated to arrive before m_1 was issues, then *P2* can detect that *P1* 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* (pair of *Input* and time), sorted by time.

Semantically, every call to the the *stateAt* function implies a recursive computation of the current state. This recursion starts from the *initialState*, 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 expansive. 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* have 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 will then hit at that point, from where *nextState* is successively applies to obtain the *State* for the current time. This is how the *rolls back* mechanism illustrated in Figure 3.1 is implemented in a time efficient manner.

Regarding memory management, timing assumptions on the network allow 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, retraining 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

TODO: This section...

4.1 Scala Remake of a Commodore 64 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? 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 [15], Survivor [16] 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/JavaScript [16]. This latest open source remake served as a basis for the version of the game presented in this chapter. The code was rewritten from scratch in Scala to follow the functional programming style required by the scala-lag-comp framework, but still shares 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 to run on both Java Virtual Machines and JavaScript engines. Some IO 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 CPU usage by only requesting rendering when the page is visible to the user.

In order to reuse the visual assets from [16], the JVM version embeds a full WebKit browser in order to run the same implementation of the user interface than the JavaScript version. The rendering on the JVM goes as follows. It begins with *render* method being called with a *State* to display. This *State* is serialized using the uPickle serialization library [6], 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 it's argument back into a *State*, but 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 Graphical User Interface With React

In this section we discuss the implementation of the graphics user interface of our game using the React library [17]. In functional programing, the *de facto* standard to 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

[1] React supports a variety of architectures to build user interfaces, and is not limited the approach described in this section. For example, it is possible to store 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.

state of the application, and returns a complete HTML representation of the state as a result.

The key performance aspect of React is that the library 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 do the send the minimal set of mutations to the browser, thereby minimizing rendering time.

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, given a subset of the application state it returns an HTML representation for this subset of the application. *Components* can be composed into a tree that takes the complete application state at it's root, and propagates the necessary elements of state thought the tree, thus forming a complete rendering function of the application. Thanks to this subdivision into small *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 rerendered, one can use reference equality, which is a sufficient condition for equality when working with immutable data structure.

4.4 Experimental Result

- Reader are invited to try out the game! . To start a game, open the page trice and start moving around with arrow keys, and fire with space bar.
- Screenshot
- Server hosted on Amazon EC2 (via Heroku), on cross Atlantic server to test with bad network conditions.
- 60FPS on both platforms, lag free gameplay
- JVM version feels slower, probably due the WebKit embedding into JavaFx.
- Can feel some JVM “warm up” effect

- Lag Compensation in action (frame by frame Screenshots)

Chapter 5

Related Work

TODO: This Section...

5.1 Network libraries

The Node.js platform gained a lot of popularity in the recent year. 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 [18] is an implantation of WebSocket client and server for Node.js which provide *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 official WebRTC API [2], 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 for both client and server, out of which Sente [19] 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 which supports several transport mechanism. Instead of using an existing WebSocket emulation library, Sente implements its own solution to fallback on Ajax/long-polling when WebSocket is not available.

With the large number of languages that compile to JavaScript [20], 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 this variety of supported protocols and platform (summarized in Table 2.1).

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 can bring a lot of information about their implementations. For example, in *Starcraft II* every input is executed with a minimum delay of 200 ms, which in most cases is sufficient to transfer the input over the network and have it process 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 use a small pause to hold every simulation until the slowest client catch up, and then globally increases the input delay. The delay is then dynamically adjusted during the remaining of the game, aiming to bring it back at 200 ms. The absence of any prediction mechanism forces this delay to identical for every peer, thus being as slow as the slowest player.

The peer to peer model also raises several challenges in therms 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 hide information between peers. Many strategy games implement a mechanism of a *fog of war*, which hides the areas of the map that are out of sight of a players 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 [21] 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's completely undetectable.

In addition to their tool, [21] propose 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, on the category of oblivious intersection protocols, that allow 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 [7] suggest the use of a new programming 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 is used to formulate several latency compensation algorithms, which are implemented as part of the *Janus* toolkit to allow game developers to quickly try out with the different algorithms. In a sense, the approach of [7] in orthogonal to the one taken in scala-lag-comp, where we rely on functional programming principles to allows users of the framework to avoid direct managment of state.

5.3 Functional Programming In Games

- Pong Async `http://ragnard.github.io/2013/10/01/clojurecup-pong-async.html`
- [22]
- [23]

Chapter 6

Conclusion and Future Work

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

References

[1] Kahle Bryce and Majkowski Marek. Sockjs - websocket emulation. http://github.com/fdimuccio/play2-sockjs. 9, 10

[2] 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. 9, 11, 27

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

[4] Haoyi Li. Autowire. http://github.com/lihaoyi/autowire. 11

[5] Doeraene Sébastien. scala-js-actors. http://github.com/sjrd/scala-js-actors. 11, 12

[6] Haoyi Li. uPickle. http://github.com/lihaoyi/upickle. 13, 24

[7] 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. 15, 29

[8] Valve. Source multiplayer networking. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. 16, 17

[9] IEEE standard for distributed interactive simulation - application protocols. *IEEE Std 1278.1-1995*. 16

[10] 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. 16, 17

[11] 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. 16

[12] Jouni Smed and Harri Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006. 16

[13] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146--158, 1989. 20

[14] 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. 20

[15] Reilly Lee. Games on github. http://github.com/leereilly/games. 23

[16] Schiller Scott. Survivor: Remaking a commodore 64 game in html. http://www.schillmania.com/content/entries/2012/survivor-c64-html-remake/, 2012. 23, 24

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

[18] Stangvik Einar Otto. ws: a node.js websocket library. http://github.com/einaros/ws. 27

[19] Taoussanis Peter. Sente, channel sockets for clojure. http://github.com/ptaoussanis/sente. 27

[20] List of languages that compile to JS. http://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. 27

[21] 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. 28

[22] Hague James. Purely functional retrogames. http://prog21.dadgum.com/23.html, 2008. 29

[23] Chiusano Paul. The limits of procedural generation and lazy simulation in games. http://pchiusano.github.io/2014-09-16/lod-simulation.html, 2014. 29