Multi-tier Functional Reactive Programming for the Web

Bob Reynders
iMinds - Distrinet, KU Leuven
bob.reynders@student.kuleuven.be

Dominique Devriese Frank Piessens
iMinds - Distrinet, KU Leuven
{firstname.lastname}@cs.kuleuven.be

Abstract

The development of robust and efficient interactive web applications is challenging, because developers have to deal with multiple programming languages, asynchronous events, propagating data and events between clients and servers, data consistency and much more. Several approaches for (partly) addressing these challenges have been proposed. Two relevant ones are (1) multi-tier languages and (2) functional reactive programming (FRP). Multi-tier programming languages support the development of client and server in a single language, and hide much of the complexity related to distribution. FRP offers the right abstractions to make event-driven programming convenient, safe and composable. However, existing web frameworks and programming languages exploit the benefits of both approaches separately, for example by restricting the use of FRP to the client side.

We propose *multi-tier FRP for the Web*, a novel approach to writing web applications that deeply integrates FRP and multi-tier languages, and where the whole is greater than the sum of its parts. In multi-tier FRP, the developer programs server and client together as an FRP application composed of behaviors (signals) and events. He/she chooses explicitly where the boundary between server and client is crossed. To make our approach more concrete and provide evidence of its potential, this paper presents a concrete design and implementation of a multi-tier FRP API for the web in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. This allows us to present initial evidence of the benefits of the multi-tier FRP approach on example applications, and to experiment with possible answers to the remaining questions. Concretely, we show possible solutions for problems like exposing client identity on the server and efficiently preloading clients with the latest application state. Our results show that multi-tier FRP is a promising, declarative, yet practical way of writing web applications.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Data-flow languages

Keywords Functional Reactive Programming, FRP, Multi-tier Web Framework

1. Introduction

Developing interactive web applications presents a number of interesting challenges for programmers. One important challenge is the inherent distributed nature of the platform with parts of the application running on the server and other parts on the (zero or more) clients. Another challenge is dealing with the asynchronous communication that is inherent to user communication and typically used in the web's client-server communication for reasons of performance, failure tolerance and responsiveness towards the user.

The standard approaches for dealing with these challenges (the use of callbacks and separate server- and client-side codebases, typically in different programming languages) present significant downsides. In recent years, interest in web application development has not ceased to increase (both in research and industry) and several novel approaches have been proposed to improve over these approaches. In this paper, we focus on two such novel solutions specifically: Functional Reactive Programming (FRP) and multi-tier languages.

Asynchronous communication and FRP The standard approach for dealing with asynchronous user and client-server input and output is the use of *callbacks*: imperative components that are invoked in response to asynchronous events. Specifically, web applications use JavaScript event handlers on the client side and HTTP request handlers on the server side. A web application containing many such callbacks, all potentially modifying the application's mutable state, may have a very complex control flow within and across both parts of the application. Such code can be very difficult to reason about.

FRP (Elliott and Hudak 1997) (although initially proposed for modelling animations) can be used as an alternative programming model for asynchronous applications. Instead of using side-effecting callbacks, the program is con-

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2661136.2661140

structed by composing *behaviors* (also known as *signals*) and *events*: components representing time-dependent values. Programs constructed in this manner can be given elegant denotational semantics, compose nicely and are relatively easy to reason about. FRP has been applied to the client-side web setting in several practical frameworks and languages like Flapjax (Meyerovich et al. 2009), Ur/Web (Chlipala 2013) and Elm (Czaplicki 2012; Czaplicki and Chong 2013).

Recently, we are also seeing an increase in mainstream adoption of *reactive* frameworks that provide enhanced databinding: Liberty and Betts (2011)'s JavaScript implementation of Meijer's .NET Reactive Extensions (Meijer 2010) and to a lesser extent Google's AngularJS (Google 2010) and Facebook's reactive UI framework React (Facebook 2013). The precise boundary of FRP appears to depend on who you ask, but for the purposes of this text, we consider reactive frameworks as related to FRP, but not quite the same. They use abstractions that are similar to FRP behaviors (i.e., time-varying values that propagate changes), but are not really pure FRP. They step outside pure FRP with e.g. support for attaching imperative callbacks or imperative (un-)subscribing to events. While these techniques have merit, they cannot be given an equally elegant denotational semantics as FRP and are harder to reason about. In this project, we work with pure FRP and although Scala does not allow us to enforce this, we do not intend the programmer to use imperative callbacks or other impure extensions.

The web as a distributed platform To deal with the web platform's inherent distribution of an application between client and server, an application is often split into separate client-side and server-side programs. The JavaScript programming language is typically used for the client-side part, a server-side language of choice for the part on the server and often manually serialized messaging for communication between the tiers. However, this approach presents important downsides like the separation of an application over two partial codebases (often in different programming languages), the need for compatible marshalling and unmarshalling of client-server communication and the limitations of the JavaScript programming language (no static types, peculiar semantics (see e.g. Guha et al. 2010) etc.).

To deal with the distribution of web applications over client and server, the literature has recently seen the appearance of multi-tier languages (see among others Cooper et al. 2007; Serrano and Queinnec 2010; Neubauer and Thiemann 2005; Google 2006; Chlipala 2013). In such languages, both the client and server parts of a web application are written in a single codebase and a single programming language. From this single codebase, the language implementation produces client-side executable code (typically in JavaScript) and interprets or compiles it for execution on the server. Often, the language provides synchronous and/or asynchronous communication primitives without requiring the programmer to write (un)marshalling code for the messages. The advantages of multi-tier languages are that web applications are no longer separated over separate codebases in separate programming languages, the additional features that the single language may offer over JavaScript (e.g. a static type system) and native support for client-server communication. Note that in most systems it is still the programmer who delineates the client-side and server-side parts of the application by explicitly annotating where the client-server boundary is crossed.

Combining FRP and multi-tier languages FRP and multi-tier languages have been combined before in the web setting, but always by using FRP solely at the client-side (Chlipala 2013; Gugenheim 2011; Bazerman 2012). This approach combines the advantages of both ideas, but only in a limited way. It brings the FRP model only to one part of the application and asynchronous client-server communication on the server is treated differently (using callbacks) than other asynchronous input/output (using FRP).

In this paper, we propose to integrate FRP and multi-tier languages more deeply by applying what we call *multi-tier FRP*. The idea is to construct the entire program as a single FRP application. Client- and server-side behaviors and event streams are statically distinguished and the boundary can only be crossed explicitly. This is done using new primitives that replicate a client-side behavior or event stream to the server or vice versa.

To make our proposal more concrete and to present evidence of its potential, we present a concrete instantiation of our approach in the form of a detailed API design and implementation in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. The precise definition of the primitives is tightly coupled to the characteristics of the distributed web platform, like the fact that there is one server[1] but multiple clients, and the fact that the clients need to be distinguishable by the application.

A very special characteristic of the web platform that influences the primitives is the fact that web clients may be opened or closed at any time and that the client code is not pre-installed on the client but provided by the server when it is started. This allows the server to always deliver the latest version of the client-side code and even adapt it just before being sent. This characteristic of the web platform turns up in our API for replicating server-side behaviors to the client. When a user opens an application written in our framework, client-side code will be provided that is pre-loaded with the current state of server-side behaviors that were replicated to the client-side, a feature which we have found both very useful and elegant.

For modeling JavaScript parts of the application, our Scala API and implementation make use of the JS-Scala

---

[1] See Section 7 for some ideas about dealing with server replication.

library by Kossakowski et al. (2012), based on the Scala Lightweight Modular Staging (LMS) framework by Rompf and Odersky (2010). The use of JS-Scala allows us to treat JavaScript as an Embedded Domain Specific Language (EDSL) from within Scala. The types of our client-server APIs need to mention the LMS `Rep[T]` types to ensure that client-side parts of the application can in fact be compiled to JavaScript. We support transparent serialization of behaviors' content types from client to server and back by requiring a serializer object to be available as a Scala implicit object (Oliveira et al. 2010).

We emphasize again that while our proposed API and our implementation are necessarily Scala-specific, our approach is more general. While we have designed the API to support a library implementation in an unmodified widely used programming language, implementing our approach in a true multi-tier language could have advantages of its own, particularly avoiding the need for `Rep[T]` types in the API.

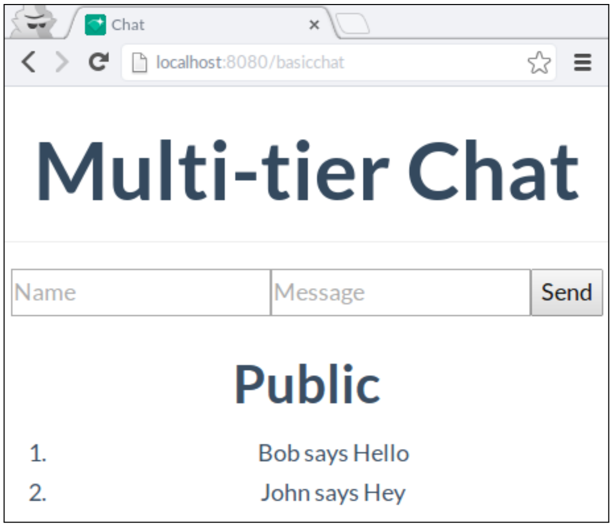
In summary, our contributions are the following:

- We propose the novel approach of multi-tier FRP for the Web.
- We instantiate our approach with the API design and implementation of a web framework in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. We explain how the API design reflects the characteristics of the web platform, specifically the need for distinguishing clients on the server and the opportunity for efficiently pre-loading client code with the latest required application state.
- We show the feasibility and potential of our approach by demonstrating simple demo applications.
- Finally, we discuss future work: challenges for developing a full-featured web framework in our approach and some of our ideas to address them.

In the rest of this paper, we start by introducing our proposal for multi-tier FRP for the web in Section 2. Next, after some more background in Section 3, we present our concrete Scala instantiation of the approach with the API design in Section 4 and the implementation in Section 5 while Section 6 contains an overall summary of our proposal. We present ideas on future work in Section 7 and give an overview of related work in Section 8.

2. Multi-tier FRP by Example

In this section we will present our approach informally based on a simple chat application example, before presenting details in the next sections. For presentation reasons, we first show an FRP implementation of only the client part of the chat application (with an unspecified server part) and then show our proposed approach with both tiers of the application as part of a single FRP implementation.



Multi-tier Chat

Public

1. Bob says Hello
2. John says Hey

Figure 1. Screenshot of our example chat application in its simplest form.

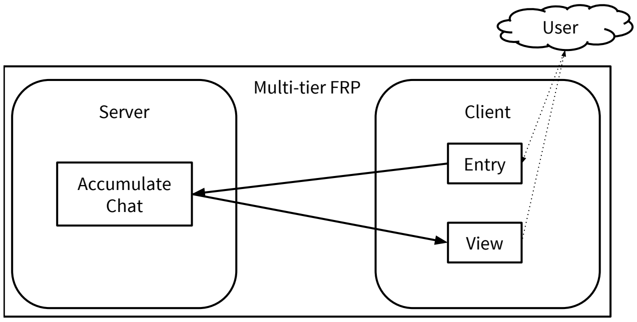


Figure 2. Mental model of data flow in chat application

Our example application is very simple, although we will make it slightly more realistic later on. For now, Figure 1 shows a screenshot: there are two input fields (*name* says *message*), a send button and a chat log. Multiple clients can connect to the single server. Figure 2 shows a very simple mental model of the implementation of the application. FRP implementations are often quite close to this kind of schematic representation, as we will see further.

2.1 FRP on the Client

Figure 3 shows a JavaScript implementation of the client-side part of our chat application. The code uses the JavaScript FRP library BaconJS (Paananen 2012) with some convenient extensions of our own that we don't go in detail for. Some functions that are not important for the presentation are left unspecified for brevity. It serves as a demonstration of how a simple chat client would be modeled if functional reactive programming is used on the client side. Let us take a closer look at the implementation.

```
function Entry(name, message) {
  return {"name": name, "message": message};
}

var entry = $("#name").values
  .combine($("#msg").values, function(n, m) {
    return Entry(n, m);
  });
var clicks = $("#send").asEventStream("click");

function mkRequest(entry) { /* create POST req */ }
var requests = entry.sampledBy(clicks).map(mkRequest);
doPOST(requests, "example.com/input");

var chatSrc = new EventSource("example.com/output");
var chat = chatSrc.asEventStream("message")
  .map(function(e) {
    return JSON.parse(e.data);
  });
function template(entries) { /* convert to html */ }
var view = chat.map(template);
render(view);
```

Figure 3. Single-tier FRP client for a chat application, written in JavaScript.

The code defines a number of *behaviors* and *event streams*. These are core FRP abstractions that we will introduce properly in Section 3.1. A behavior represents a value in the application that may change over time, such as the contents of the chat log or the contents of a user input field. An event stream represents a channel on which new values appear at certain times, such as the coordinates clicked by the user or the requests that should be sent to the server.

Concretely, the code in Figure 3 defines the behavior `entry` that `combines` the contents of `name` and `message` input fields, wrapping their values in a JavaScript object using the `Entry` method. The event stream `clicks` contains an event for every user click on the `send` button. An event stream of requests to be sent to the server is then constructed by sampling the behavior `entry` whenever an event appears on stream `clicks` and constructing a request from its contents. An unspecified function `doPOST` ensures that these events are sent to the server (when they appear) using XMLHTTPRequests.

Figure 3 also defines the event stream `chat`. It models a network connection on which the client listens to new messages from the server. These messages will contain the updated contents of the chat log and arrive whenever new messages appear on the server. From this stream of chat log updates, an event stream `view` is constructed that contains updated HTML renderings of the chat log. The `render` function ensures that these updates will be applied in the displayed web page.

Note that functions like `asEventStream`, `render` and `doPOST` link the FRP application to the outside world. In a pure FRP setting, they (or the primitives they make use of) should be considered as APIs that are part of the framework, not the application.

To make this client operational it needs an accompanying server. For the sake of brevity we will not include the server code, but merely sketch it. Besides serving the client web page and the code in Figure 3 to the client, the server needs to listen to client requests on `example.com/input`. Received messages from clients are added to the chat log which is kept around as the server's state. When the chat log is updated, the server will push new messages to all clients listening on `example.com/output`. The book-keeping of the client connections listening on `example.com/output` can typically be taken care of by a library or framework.

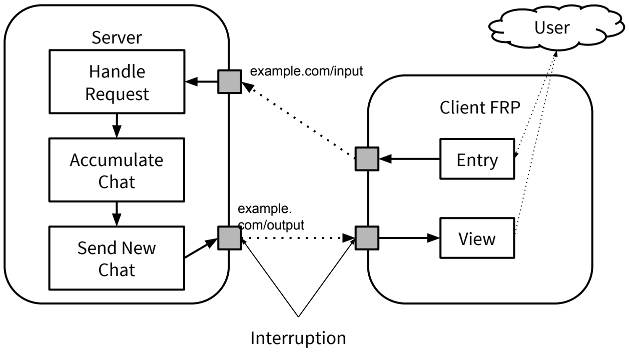


Figure 4. Mental model of data flow in first version of our chat application (using client-side FRP).

In a typical implementation, the server would be implemented using imperative callbacks. Concretely, there would be a request handler responding to incoming requests on `example.com/input`, imperatively updating the server state and sending out messages to clients on `example.com/output`. A less common alternative is to write the server-side part as an FRP application as well, using server-side FRP frameworks like the Reactive Extensions by Meijer (2010).

Typically, the server-side part of the application would be written in a server-side programming language of choice, resulting in a codebase split between JavaScript and that other language. Alternatives are to use JavaScript on the server as well or using a multi-tier language that allows the client-side code to be written in the same language as the server-side code. Multi-tier languages have the advantage that they often offer features that are not available in JavaScript, such as a type system, class-based object-orientation and more standard semantics. We think the example so far already demonstrates some essential aspects of the FRP approach. If we ignore the interfaces to the outside world, then the code is constructed by composing event streams and behaviors, not by defining imperative callbacks and attaching them to events. Figure 4 schematically represents our implementation and at least on the client side, it approaches the mental model in Figure 2 that we started from.

Nevertheless, there are still quite a few opportunities for improving this implementation. We see the following remaining problems:

Interrupted Reasoning The schema in Figure 4 shows a much more strict separation between the client-side and server-side tier than was present in the mental model in Figure 2. Very often, the codebase splits up the application in separate codebases, but even when this is avoided in a multi-tier language, there typically remains a semantics gap between the tiers caused by using FRP on the client side and imperative callbacks on the server side. Even if FRP should be used on both the server and client side (which we have not found examples of), current approaches would still treat both parts as separate applications with separate semantics.

Duplicate Code In non-multi-tier languages, our application would typically suffer from significant code duplication. For example, data definitions would need to appear in both the server- and client-side codebase.

Glue-code Although this is sometimes taken care of by languages and frameworks, connecting both tiers of the application can involve quite a bit of glue code. In our chat application, the `example.com/[in/out]put` handlers are good examples. They are boilerplate sections of code that are there for (admittedly important) technical reasons; they are not present in the mental model in Figure 2 and we prefer a framework or library to take care of this part of the work completely.

Bootstrap A problem that we have underemphasized so far is how the client's view of the chat log is initially populated. In our implementation, this could be handled by making the client issue a request to the server after startup to load the data. However, it is more efficient to distribute the initial data as part of the JavaScript code. However, both approaches require special programmer effort. The first approach requires an additional server-side request handler that can send the chat log contents to clients and additional client-side initialization code and the second approach requires server-side code that injects the current chat log state into the code to be sent to a new client.

We think these aspects of the implementation can be improved by writing the application using our multi-tier FRP approach. We demonstrate and explain this in the next section.

2.2 Alternative: Multi-tier FRP

In Figure 5, we demonstrate multi-tier FRP for the web with an alternative implementation of our example chat application. The code implements both the client and server part of the application. It is implemented in Scala, using our framework that we discuss in Section 5. Let us take a closer look.

A first thing that the reader may notice is that the code sometimes mentions types of the form `Rep[T]`, where `T` is a normal Scala type. These types come from the use of the JS-Scala library (Kossakowski et al. 2012). We will introduce JS-Scala in Section 3.2 and our use of it in Section 4.2. For now, it suffices to know that a value of type `Rep[T]` represents a JavaScript term of type `T`.

```
case class Entry(name: String, msg: String)
  extends Adt
val EntryRep: (Rep[String], Rep[String]) => Rep[Entry] =
  adt[Entry]

lazy val name: Rep[Input]  = text("Name")
lazy val msg: Rep[Input]   = text("Message")
lazy val send: Rep[Button] = button("Send")

lazy val submit: Event_c[Entry] = {
  val nameV: Beh_c[String] = name.values
  val msgV: Beh_c[String]  = msg.values
  val entry = nameV.combine(msgV) { EntryRep(_, _) }

  val clicks: Event_c[MEvent] = send.toStream(Click)
  entry.sampledBy(clicks)
}
lazy val serverSubmit: Event_s[Entry] = submit.toServerAnon

lazy val chat: Beh_s[List[Entry]] =
  serverSubmit.fold(List.empty[Entry]) { (acc, entry) =>
    entry :: acc
  }

def template(view: Rep[List[Entry]]): Rep[Element]
lazy val main: Beh_c[Element] =
  chat.toAllClients.map(template)

implicit val itemFormat = jsonFormat2(Entry)
```

Figure 5. Example chat application implemented using multi-tier FRP for the web.

Other types in the example that deserve some explanation are those of the form `ClientBehavior[T]`, `ClientEvent[T]`, `ServerBehavior[T]` and `ServerEvent[T]`.[2] These types correspond to the FRP behaviors and event streams that we encountered in the previous example, except that we now make the distinction between those on the server and client.

The code constructs HTML text input fields `name` and `msg` and submit button `send`. The event stream `submit` models the user's submitted messages; it is constructed by combining the behaviors `nameV` and `msgV` (representing the contents of the corresponding text inputs) by wrapping their values in the `Entry` case class and sampling the resulting behavior whenever the `clicks` event stream (representing button clicks) fires.

In the next step, the event stream `submit` is replicated to the server side, using the `toServerAnon` method. Note how this method transforms a `ClientEvent[Entry]` to a `ServerEvent[Entry]`, as expected. The `toServerAnon` is one of the simplest communication primitives in our API. It is *anonymous* in the sense that the server cannot determine from what client an event on the resulting stream originated (see Section 4.4 for alternative APIs when this is not sufficient).

From the `serverSubmit` event stream, the `chat` behavior is constructed, containing the accumulated server state as a list of all entries in the chat log. It is constructed using the `fold` FRP primitive which takes an event stream to accumu-

---

[2] We shorten `ClientBehavior` to Beh$_c$, `ClientEvent` to Event$_c$, `ServerBehavior` to Beh$_s$ and `ServerEvent` to Event$_s$ for improving code presentation.

late over, an initial value and a method that adds a new event to the accumulated state, similarly to well-known `fold` or `reduce` methods for accumulating over collections.

The accumulated server state is replicated back to the client using the `toAllClients` method. The result is used to fill a `template` of the web page. We have omitted the template implementation for brevity, but note that it must include the text inputs and submit button constructed previously, for everything to work.

Finally, the code in Figure 5 contains some technical definitions that are needed to make everything work. We use a standard Scala case class `Entry`, but we need some technical tools to work with it. Specifically, we use the JS-Scala `adt` macro (that we do not go into details for) to build `EntryRep`, a function to construct values of type `Entry` in JS-Scala code. Secondly, we use spray's `jsonFormat2` to automatically construct `itemFormat`: a JSON serializer and deserializer for our `Entry` type. The value is required implicitly for the calls to `toServerAnon` and `toAllClients`.

In summary, the implementation in Figure 5 shows the essence of our multi-tier FRP for the web approach. Both server and client parts of the chat application are implemented as a single FRP application, with special primitives `toAllClients` and `toServerAnon` used for explicitly crossing the tier boundary. If we take another look at the problems discussed in the previous section, we can see that our approach provides several improvements.

Duplication and Glue-code Our use of a multi-tier framework, as well as the use of a single FRP network covering both tiers of the application, solves the problems related to interrupted reasoning, code duplication and glue code that we discussed before. Like in other multi-tier frameworks, there is no more duplication of data type definitions and the application is not split into several independent parts.

Interrupted Reasoning Unlike existing multi-tier languages and frameworks, our use of a single FRP network that covers both tiers, ensures that we benefit from the advantages of Functional Reactive Programming on both tiers and that there is no semantic gap in the treatment of asynchronicity on both tiers. Glue code between client and server is removed by our use of `toAllClients` and `toServerAnon` communication primitives.

Bootstrap Interestingly, our use of the `toAllClients` communication primitive, which replicates a server behavior to the client allows us to efficiently solve the *bootstrapping* problem as well. Let us explain how this works. Consider the `chat` server-side behavior (modeling the server-side state of the chat log) that is replicated to the client using the `toAllClients` method. When a client connects to the server, it needs to know the initial values of the replicated behavior, in order to calculate the initial value of its other behaviors, specifically the ones needed for constructing the initial version of the web page like our `chat.toAllClients`. Our framework automatically takes care of this, by including the most recent value of the `chat` behavior in the code that is sent to the server. This is very efficient since no further network requests to the server are needed before the initial display of the web page. However, it is completely taken care of by the framework without requiring special programmer effort.

Note that we are not saying that our approach is the only one solving some of the above problems. However, we do believe that our approach is unique in solving *all* of them. Specifically, we use FRP on both tiers for modeling asynchronous code and there is no gap (in terms of programming languages or semantics) between the implementations on both tiers. Furthermore, we benefit from standard advantages of multi-tier languages or frameworks, specifically the fact that we avoid duplicating code, we use a single programming language and we can exploit the static type system and other features of a powerful programming language like Scala.

With this informal introduction, we hope the general idea of multi-tier FRP for the web is clear. In the next sections, we make the idea more concrete and prove feasibility and usefulness of the approach by showing a possible API design and implementation for the approach in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language.

3. Background

Before we can explain our proposed API and implementation in the next sections, we need to briefly present some background regarding FRP in general and about the JS-Scala library.

3.1 Functional Reactive Programming

FRP is a functional approach to programming with time-dependent values and asynchronous events. We adhere to the primary two concepts that were part of the original development in Fran (Elliott and Hudak 1997):

Event Stream An event stream is a channel on which events arrive as values of a given type. Mouse click events, for example, can be represented by values containing metadata regarding the click (e.g. position, button pressed). An event stream `Event[T]` can be thought of as a continuously expanding list of type `List[(Time, T)]`.

Behavior A `Behavior[T]` is an abstraction representing time varying values of type `T`. It can semantically be thought of as a function of type $\texttt{Time} \to \texttt{T}$; a function that always returns a value which may or may not depend on time. The user cursor position, for example, is a behavior since it is always defined and may vary over time.

FRP is a way of reifying event streams and time varying values into first class citizens so interactive applications can be modeled in a declarative manner by combining behaviors and events into an actual application. For code examples, we refer to the code shown in Section 2.

It is not our intention to give a thorough overview here, but we mention that FRP comes in many variations. For the purposes of this paper, we use a *discrete time model* (i.e., the values of behaviors change discretely, not continuously).

3.2 JS-Scala: writing JavaScript in Scala

For representing JavaScript code in the client side of our API, we use JS-Scala: an Embedded Domain Specific Language (EDSL) for writing JavaScript programs within Scala in a natural syntax.[3] The library builds on LMS by Rompf and Odersky (2010): a framework for embedding DSLs inside Scala, supporting modular DSLs and DSL interpretations.

In JS-Scala, a JavaScript block that produces a value of type T is represented by a Scala value of type `Rep[T]`. The following example is a JavaScript program built with JS-Scala (note its type):

```
def main(): Rep[Unit] = {
  val name: Rep[String] = prompt("What's your name?")
  println("Hello, " + name + "!")
}
```

The JS-Scala code generator can be used convert this code to JavaScript, resulting in the following output:

```
function main() {
var x0 = prompt("What's your name?");
var x1 = "Hello, "+x0;
var x2 = x1+"!";
var x3 = console.log(x2);
}
```

For technical reasons related to JS-Scala's reification of side-effecting statements in the JavaScript DSL, code that contains JS-Scala scripts will sometimes be constructed as *lazy* values in our code samples. This ensures that their effects are properly reified as part of the correct JavaScript block.

4. A Scala API for multi-tier FRP

The Scala API that we propose consists of different parts that we will explain in the following sections. We start with the server-side FRP APIs, which are quite standard. Next, we present the client-side API which is similar except that it needs to be adapted to the use of JS-Scala to support generating JavaScript code. Finally, we present the communication primitives that make the link between both tiers. The entire API is written in Scala's object-oriented style.

4.1 FRP API on the server

The two main classes for the Server API are `Event`$_{\mathtt{s}}$`[T]` and `Beh`$_{\mathtt{s}}$`[T]`. Figure 6 lists the available methods using the notation

`instance`$_{\mathtt{tier}}$`.method[type param](args) : return type`

```
Event_s[T].map[A](f: T => A): Event_s[A]
Event_s[T].merge[A >: T](e: Event_s[A]): Event_s[A]
Event_s[T].filter(p: T => Boolean): Event_s[T]
Event_s[T].hold[U >: T](i: U): Beh_s[U]
Event_s[T].fold[A](i: A)(f: (A, T) => A): Beh_s[A]

Beh_s[T].map[A](f: T => A): Beh_s[A]
Beh_s[T].sampledBy(e: Event_s[_]): Event_s[T]
Beh_s[T].fold[A](i: A)(f: (A, T) => A): Beh_s[A]
Beh_s[T].combine[A, B](b: Beh_s[A])
                        (f: (T, A) => B): Beh_s[B]
```

Figure 6. Server FRP API

```
Event_c[T].map[A](f: Rep[T] => Rep[A]): Event_c[A]
Event_c[T].merge[A >: T](e: Event_c[A]): Event_c[A]
Event_c[T].filter(p: Rep[T] => Rep[Boolean]): Event_c[T]
Event_c[T].hold[U >: T](i: Rep[U]): Beh_c[U]
Event_c[T].fold[A](i: Rep[A])
            (f: (Rep[A], Rep[T]) => Rep[A]): Beh_c[A]

Beh_c[T].map[A](f: Rep[T] => Rep[A]): Beh_c[A]
Beh_c[T].sampledBy(e: Event_c[_]): Event_c[T]
Beh_c[T].fold[A](i: Rep[A])
            (f: (Rep[A], Rep[T]) => Rep[A]): Beh_c[A]
Beh_c[T].combine[A, B](b: Beh_c[A])
            (f: (Rep[T], Rep[A]) => Rep[B]): Beh_c[B]
```

Figure 7. Client FRP API

Let us briefly introduce the individual methods.

map The map transformation applies a given function to every occurrence or update of an event stream or behavior to create a new result.

merge Interleaves the occurrences of two event streams. For example, two button event streams can be merged into a new one that contains occurrences for both buttons. If both event streams fire at the same time, the first event will precede the second.

filter Filters events on an event stream according to a given predicate.

hold Transforms an event into a behavior by holding the latest event value that fired. When no event has fired yet, the resulting behavior has the given initial value.

fold From an event, an initial value `i` and a stepper function `f`, `fold` produces the behavior of accumulated values.

sampledBy Takes a snapshot of a behavior whenever a given event fires.

combine Composes the values of two behaviors using a provided combination function. For brevity we only list one combine method, our actual API contains combine methods for an arbitrary amount of behaviors.

4.2 FRP API for the Client

The client part of our proposed Scala API is shown in Figure 7. With `Event`$_{\mathtt{c}}$ and `Beh`$_{\mathtt{c}}$ replacing `Event`$_{\mathtt{s}}$ and `Beh`$_{\mathtt{s}}$, it is a mirror of the server API except for one aspect. The difference is that most of the operations require parameters of

---

[3] JS-Scala should not be confused with Scala.js (Doeraene 2013) which is a Scala to JavaScript compiler

the staged form Rep[T] instead of T. Compare, for example, the method signatures for Events' *map* method on both tiers:

```
Event_s[T].map[A](f: T => A): Event_s[A]
Event_c[T].map[A](f: Rep[T] => Rep[A]): Event_c[A]
```

The map method for an $\texttt{Event}_\texttt{c}$[T] takes a function of type Rep[T] ⇒ Rep[A] instead of T ⇒ A. This ensures that the function passed to the client-side map is a JS-Scala function that represents and can be translated to JavaScript code. Other client-side APIs use Rep[_] types for the same reason.

4.3 Client-Server Interaction

In order to cross the boundary between the two tiers, we provide special communication primitives. Our primitives are methods for events and behaviors that allow replicating them from client to server or vice versa. In the background, the framework will implement this by sending requests between the tiers when events arrive as appropriate. The simplest of the methods that we offer are the following:

```
Event_c[T: JSJsonW: JsonR].toServerAnon: Event_s[T]
Event_s[T: JsonW: JSJsonR].toAllClients: Event_c[T]
```

The notation T : JSJsonW : JsonR indicates a context bound and can be used as a Scala analogue to type classes (Oliveira et al. 2010). We use four type classes JsonR, JsonW, JSJsonR and JSJsonW to require the availability of a function for reading or writing a JSON encoding for a type T in either server or client code. The client-server interaction methods can only be used for types T for which the appropriate type class instances are available.

We also offer the following API extension for behaviors:

```
Beh_s[T: JsonW: JSJsonR].toAllClients: Beh_c[T]
```

We believe replicating a behavior from a server to clients can be given a precise semantics. For practical purposes, it is in fact essential in our multi-tier setting for allowing clients access to the latest server-side state when they start. The replication of the chat behavior in Figure 5 shows that the primitive is quite natural and practical in a multi-tier FRP web application. In Section 5.2, we explain how the primitive can be implemented efficiently, by pre-loading client code with the latest values of the behaviors that are replicated to the client.

Finally, we point out that we do *not* offer an API for replicating a client-side behavior to the server. More about this in Section 4.4.

4.4 Distinguishing Clients in the API

The reader may have noticed that until now, our API makes an important simplification: clients cannot be distinguished on the server. The previous example replicates events from the client anonymously (i.e. no way to learn what client an event came from) and every replication from the server as a broadcast (i.e. no way to limit the clients that receive an event or send different values to different clients). With such a limitation it would be impossible to write genuine web applications, which send different data to different clients and treat data coming from them differently. For a concrete example, consider how a chat application might send each client only his own private messages or only allow them to see a chat log after they are authenticated).

We therefore propose the following additional communication primitives (type requirements omitted for brevity):

```
Event_c[T].toServer: Event_s[(Client, T)]
Event_s[Client => Option[T]].toClient: Event_c[T]
Beh_s[Client => T].toClient: Beh_c[T]
```

We chose to add client information explicitly to the results, effectively providing access to information that was lost using the APIs presented earlier.

As we will explain, these primitives are more general than the simpler ones in the previous section, i.e. the old ones can be implemented in terms of the corresponding new one.

$\texttt{Event}_\texttt{c}$.toServer The first method should be read with the intuition that a client event contains more information than just the event occurrence and value. Since it is bound to one client it also contains the client's identity. To allow full access to the information of a client event on the server we propose the API above. The Client value in the returned server event is an opaque but comparable value, which uniquely identifies a client.

$\texttt{Event}_\texttt{s}$.toClient The second method on the other hand, involves transforming event occurrences on the server into occurrences on the client. This primitive allows the programmer to define per client whether it should receive an event and if so, what value should be sent. More concretely, when the server event stream fires with an event value of type Client ⇒ Option[T], then only the clients for which this function returns Some x will receive the event and they will receive x as the event value.

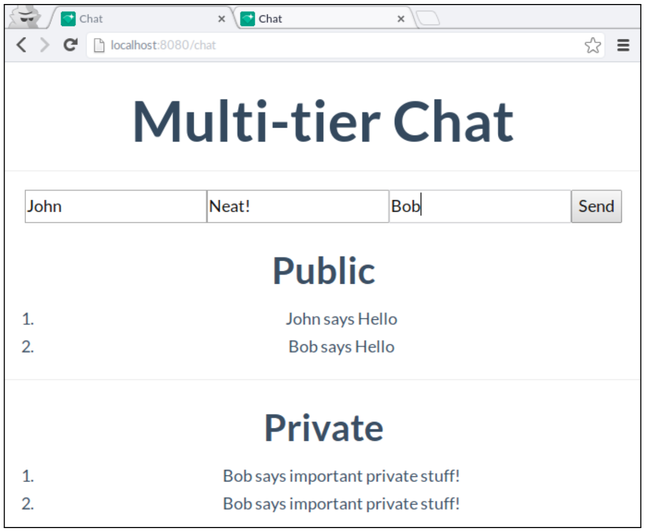
$\texttt{Beh}_\texttt{s}$.toClient Finally, when replicating a behavior from the server to the client, we also want to allow showing different values to different clients. Concretely, when the server behavior contains a value of type Client ⇒ T, the replicated behavior will contain, on every client, the function's result for *that* client.

$\texttt{Beh}_\texttt{c}$.toServer We previously mentioned that we did not implement replication for client behaviors to the server. Given that clients' lifetimes in a web setting are subintervals of the server's, a plausible design would be to provide an API primitive of type:

```
Beh_c[T] => Beh_s[Map[Client, T]]
```

This primitive would produce a server behavior that represents the value of the behavior in each individual client. However, we found the type signature rather clunky and the added value limited, so we decided not to include it.

To demonstrate the power and flexibility of the extended API, Figure 9 shows an example application that extends our previous chat application. This time we create a chat



Multi-tier Chat

John | Neat! | Bob

Public

1. John says Hello
2. Bob says Hello

Private

1. Bob says important private stuff!
2. Bob says important private stuff!

Figure 8. Multi-tier FRP chat application with private messaging UI

```
case class Entry( name: String,
                  target: Option[String],
                  content: String) extends Adt
val EntryRep = adt[Entry]
implicit val itemFormat = jsonFormat3(Entry)

case class Chat( pub: List[Entry] = Nil,
                 priv: Map[Client, List[Entry]] =
                         Map() withDefaultValue Nil)
case class View(pub: List[Entry], priv: List[Entry])
  extends Adt
implicit val viewFormat = jsonFormat2(View)

lazy val name: Rep[Input] = text("Name")
lazy val msg: Rep[Input] = text("Message")
lazy val target: Rep[Input] = text("Target")
lazy val send: Rep[Button] = button("Send")

lazy val submit: Event_c[Entry] = {
  val combined: Beh_c[Entry] =
    name.values.combine(target.values, msg.values) {
      (n, t, m) =>
        EntryRep(n, if (t == "") none else some(t), m) }
  combined.sampledBy(send.toStream(Click))
}

lazy val onServer: Event_s[(Client, Entry)] =
  submit.toServer

lazy val chat: Beh_s[Chat] =
  onServer.fold((Map[String, Client](), Chat())) {
    case ((ppl, c@Chat(pub, priv)), (sender, entry)) =>
      val newPpl = ppl + (entry.name -> sender)
      val newChat = entry.target match {
        case Some(t) =>
          def cons(c: Client) = c -> (entry :: priv(c))
          c.copy(priv =
            priv + cons(ppl(t)) + cons(sender))
        case None => c.copy(pub = entry :: pub)
      }
      (newPpl, newChat)
  }.map { case (map,chatLog) => chatLog }

lazy val chatVw: Beh_s[Client => View] =
  chat.map {
    case Chat(pub, priv) =>
      client: Client => View(pub, priv(client))
  }

def template(view: Rep[View]): Rep[Element]
def main: Beh_c[Element] = chatVw.toClient.map(template)
```

Figure 9. Multi-tier FRP chat application with private messaging

application with support for private messaging with a UI that looks something like Figure 8. The listing shows the full code with just the UI template omitted for brevity.

5. Implementation

We have implemented the proposed APIs in a proof-of-concept multi-tier FRP web framework in Scala. The implementation can be downloaded from Github[4], together with instructions for building the code and experimenting with it.

Our implementation makes use of a composable web server abstraction called a *route*, that (to our knowledge) was made popular by Sinatra (Mizerany 2007). For our purposes you may think of a route as a combination of an URL and its corresponding functionality. The implementation of routes that we are using is the Spray library (Rudolph and Doenitz 2012).

5.1 Client & Server FRP API

To implement our API with minimal effort we rely on an existing JavaScript FRP framework called BaconJS (Paananen 2012). Our $\mathtt{Event}_\mathtt{c}$ and $\mathtt{Beh}_\mathtt{c}$ are implemented using BaconJS representatives underneath. The implementation consists essentially of a BaconJS event or behavior (defined in JS-Scala) and an optional route. The optional route comes into play for implementing the communication primitives (see below).

$$\mathtt{Event}_\mathtt{c}[\mathtt{T}] \approx \mathtt{Option}[\mathtt{Route}] \times \mathtt{Rep}[\mathtt{BaconEvent}[\mathtt{T}]]$$

The methods from the standard FRP API on $\mathtt{Event}_\mathtt{c}$ and $\mathtt{Beh}_\mathtt{c}$ are simply delegated to the BaconJS representative to create a new $\mathtt{Event}_\mathtt{c}$ containing the untouched previous route and the new BaconJS result.

The implementation of $\mathtt{Event}_\mathtt{s}$ and $\mathtt{Beh}_\mathtt{s}$ is very similar. On the server, we rely on a Scala FRP implementation named Scala-reactive (Gugenheim 2011). As for client-side event streams and behaviors, the implementation is a pair between an optional route and its Scala-reactive counterpart:

$$\mathtt{Event}_\mathtt{s}[\mathtt{T}] \approx \mathtt{Option}[\mathtt{Route}] \times \mathtt{ReactiveEvent}[\mathtt{T}]$$

Operations on server-side events and behaviors are delegated to their counterparts as well.

5.2 Basic Client & Server Interaction

Implementing the communication primitives from Section 4.3 requires generating appropriate glue code. Although our APIs keep the client-server boundary visible to the programmer and he/she still decides when it is crossed, the technical machinery for actually replicating events and be-

---

[4] https://github.com/Tzbob/s-mt-frp/releases/tag/onward14

haviors is abstracted away. In this section, we briefly discuss how the different primitives are implemented, using sections of pseudo-code to convey the core idea.

`Event`$_c$`.toServerAnon` For replicating a client-side event stream to the server, the first thing that we create is a fresh URL on which the server and client will communicate:

```
val url = URLEncoder
  .encode(UUID.randomUUID.toString, "UTF-8")
```

On the client, we generate an imperative callback that pushes events towards the server when they arrive. This is done by issuing POST requests to the generated URL.

```
def init_c[T: JSJsonW](e: Event_c[T], url: String) =
  e.baconRep.onValue(fun { value =>
                  doPost(url, value.toJSONString) })
```

The callback is attached to the BaconJS primitive underneath the target `Event`$_c$ so that it is executed when a new event value arrives. Note that this client initialization code is not explicitly kept track of as part of the returned server event stream. Instead, by issuing the commands during the initialization of the application, we make use of JS-Scala's reification of effects to include the code in the generated JavaScript client initialization code.

The server-side glue code that is needed for implementing the `Event`$_c$`.toServer` primitive is a handler for the aforementioned POST requests. We create a Scala-reactive `EventBus`, essentially an `Event` on which we can imperatively push data. Every time a new request comes in, the following route implementation will decode the contained JSON data using the appropriate `JsonR` method and push the decoded value onto the corresponding `EventBus`.

```
def mkRoute[T: JsonR](url: String, tgt: Event_s[T]) =
  path(url) { post {
    entity(as[String]) { data => complete {
      tgt.reactiveRep.push(data.asJson.convertTo[T])
  }}}}
```

The `Event`$_s$ that we return will contain the above route, composed with the routes of the original `Event`$_c$. It will have the created `EventBus` as the underlying event stream.

`Event`$_s$`.toAllClients` Replicating a server-side event stream to the client follows a similar process. Our implementation uses Server-Sent Events (SSE, part of HTML5) but Websockets could be used instead. SSE allow clients to connect to listening URLs and keep the connection open for the server to push chunks of data.

Again, we start by generating a fresh URL for communicating over. A route is created for this URL and whenever a client connects to it, we connect a callback to the Scala-reactive event stream underlying the `Event`$_s$. This callback will push events to the listening client.

```
def mkRoute[T: JsonW](url: String, evt: Event_s[T]) =
  path(url) { get {
    respondWithMediaType(‘text/event-stream‘) {
      startHTTPChunk(ctx)
      evt.reactiveRep.foreach { data =>
        sendChunk(ctx,
          Chunk(s"data:" + data.toJson + "\n\n"))
  }}}}
```

This route for the listening URL will be included in the `Event`$_c$ returned from the `Event`$_s$`.toAllClients` call.

On the client side we create a Bacon `EventBus`, which is similar to Scala-reactive's. The `Event`$_c$ that we will return will have this `EventBus` as its underlying BaconJS event stream. We use a client-side `EventSource` (the standard client-side interface to listen to Server-Sent Events) to connect to the generated URL. The received messages are converted from JSON and injected into the `EventBus`.

```
def init_c[T: JSJsonR](evt: Event_c[T]], url: String) =
  EventSource(url).onmessage =
    fun { ev: Rep[Dataliteral] =>
      evt.baconRep.push(ev.data.toJson.convertTo[T]) }
```

`Beh`$_s$`.toAllClients` Replicating server behaviors to client behaviors requires additional work. We can consider a behavior as an initial value together with an event stream of updates. Replicating the changes is easily implemented in terms of the previously presented `Event`$_s$`.toAllClients` method. We will now explain how we transfer the initial value to the client, so that it can be combined with the replicated event stream of changes to the full replicated behavior.

As mentioned before, our idea is to include this initial value in the JavaScript code that is sent to the client when it first connects. The behavior that we return from `Beh`$_s$`.toAllClients`, is constructed using the following code:

```
def json(): Rep[String] =
  unit(serverBehavior.reactiveRep.now.toJson)
val changes = serverBehavior.changes.toAllClients
changes.hold(delay(json).asJson.convertTo[T])
```

We use $\texttt{unit}(\texttt{t} : \texttt{String}) : \texttt{Rep}[\texttt{String}]$ to lift a constant Scala value into the JS-Scala program. The initial value is constructed as the `json` thunk above (a function of type $() \Rightarrow \texttt{Rep}[\texttt{String}]$). It is wrapped in a special `delay` call that will cause it to be evaluated only when the client code is being prepared for sending to a new client. At that moment, the 'current' value of the server behavior will be converted to JSON and included in the code.

5.3 Client Aware Interaction

Implementing the client aware APIs from Section 4.4 requires some extensions to the basic implementation.

`Event`$_c$`.toServer` We modify the preceding implementation of `toServerAnon` to replicate an $\texttt{Event}_c[\texttt{T}]$ to an $\texttt{Event}_s[(\texttt{Client}, \texttt{T})]$ by generating a unique identifier for a client when it first connects and injecting this identifier into the code generated for that client.

The client pseudo-code for `Event`$_c$`.toServer` is then extended by passing the client identifier as a parameter in the target URL.

```
def initClient[T: JSJsonW](e: Event_c[T], url: String) =
  e.baconRep.onValue(fun { value =>
    doPost(delayForClientId { id =>
          url + "/?id=" + id }), value.toJSONString)
  })
```

`delayForClientId` is an extension of the previously mentioned `delay` method. Instead of thunks of type $() \Rightarrow$ `Rep[T]`, it allows including thunks of type `Client` $\Rightarrow$ `Rep[T]` in client-side code, so that the correct client identifier is filled in when generating JavaScript code for that client.

The server-side code for `Event`$_c$`.toServer` now extracts the client identifier from the request, uses it to construct a correct client object and includes it in the produced event value.

```
def mkRoute[T: JsonR](url: String,
tgt: Event_s[(Client, T)]) =
  path(url) { parameter('id) { id =>
    post {
      entity(as[String]) { data => complete {
        tgt.reactiveRep
          .push((Client(id), data.asJson.convertTo[T]))
  }}}}}
```

`Event`$_s$`.toClient` The new `Event`$_s$`.toClient` will be invoked on event streams of type `Client` $\Rightarrow$ `Option[T]`. The server-side glue code is changed to evaluate the event values (which are now functions!) for the identifier of the client that connects to the listening URL and (optionally) push the result information to the client.

```
def mkRoute[T: JsonW](url: String,
evt: Event_s[Client => Option[T]]) =
  path(url) { get { parameter('id) { id =>
    val client = Client(id)
    respondWithMediaType(`text/event-stream`) {
      startHTTPChunk(ctx)
      evt.reactiveRep.foreach { fun =>
        fun(client).foreach { data
          sendChunk(ctx,
            Chunk(s"data:" + data.toJson + "\n\n"))
  }}}}}}
```

The changes required for the discussed implementation of `Beh`$_s$`.toAllClients` are similar to the ones just discussed, so we omit them for brevity.

6. Conclusion

In this paper, we propose multi-tier FRP for the web, an approach for developing reactive web applications in an elegant way by combining both FRP and multi-tier languages. The approach simultaneously solves existing problems such as (1) interrupted reasoning in multi-tier web development, (2) the complex control flow when using imperative callbacks to handle asynchronous user and network I/O and (3) the problem of bootstrapping clients.

FRP reifies time dependent data such as event streams and time-varying values, allowing a programmer to *model* his time dependent problems rather than *handle* them. Our proposal of spanning the FRP paradigm across client- and server-side tiers, allows writing a single FRP program whose semantics spans both tiers rather than two programs whose separate semantics combine to a full web application, thus removing a semantic gap within an application. Practically, our approach has the potential to reduce the amount of technically important yet difficult and repetitive boilerplate code in applications. Additionally, the client bootstrapping problem is handled elegantly and implicitly through the primitive for replicating server behaviors to the client.

We have made our multi-tier FRP for the web more concrete with a proposal for a Scala API, as well as an implementation built on existing technology like JS-Scala, BaconJS, Spray and Scala-reactive. The API and implementation allow us to demonstrate the potential of the general approach. They also show some possible solutions for some of the problems that arise, such as exposing client identity on the server and producing separate client and server code from the single code base of a multi-tier FRP application.

7. Future Work

We believe the results in this paper show the feasibility and the potential of multi-tier FRP for the web as a new approach to the declarative development of interactive web applications. However, quite some interesting questions remain to be solved before this can become fully practical. We see the following interesting directions for future work. Most of the tracks discussed here should be tackled at the level of the general approach as well as that of our Scala API design and implementation.

Continuous Time Model We spent little time in this paper explaining the differences between continuous or discrete time FRP and the impact of that choice on our design. First, note that some of our primitives (e.g. `Beh`$_s$`.fold`) are only meaningful in the discrete time model that we use. Secondly, it is important to understand that the choice between a continuous and discrete time model has implications for the choice between a pull- or push-based evaluation strategy. We use a discrete time model and thus a push based strategy to allow efficient tiered updates. If we had used a continuous time model, we would have needed to use a pull-based evaluation in our FRP network, including the client-server primitives (i.e. using a primitive like websockets instead of XML-HTTPRequests, for example). For future work we would like to further research the implications of continuous time models and pull-based evaluation on multi-tier FRP in general. Also Elliott (2009)'s work on combining push and pull evaluation semantics seems a promising addition to multi-tier FRP, as it might allow more fine-tuned access to resources like the DOM or databases.

Atomicity In FRP, when a primitive behavior changes value or a primitive event stream fires, the semantics dictate that all behaviors and events that depend on it should correctly reflect the changes before their new values are made visible to the outside world (e.g. by updating the web page visible to the user). This requires calculating a correct dependency order for the FRP network. The term *glitches* is used to describe situations where an FRP implementation makes new values visible that do not yet correctly reflect the changed values. The term glitch is used because it is often a temporary problem and the correct value is propagated quickly afterwards. Such glitches may result in inconsistencies in the application state and many FRP frameworks work hard to prevent them.

Within both the client- and server-side tier, we do not expect any problems related to glitches. Our implementation relies on the underlying frameworks (BaconJS and Scala-reactive) to properly ensure correct updates to the network.

When it comes to our communication primitives, we intend these to have a semantics similar to Elm's `async` (Czaplicki and Chong 2013). This means that every event on the original stream will be propagated to the FRP network on the other end and will be processed separately from events that are replicated through other replicated event streams.

Although the semantic correspondence to Elm's `async` strengthens our trust in the described semantics, we still plan to experiment with an alternative semantics for the communication primitives in future work. Under this alternative semantics, all changes or events that should be propagated to the other tier would be collected during a propagation cycle. They would then be shipped to the other tier as part of a single network message and be processed in a single time instant on the other tier. We have the intuition that this would make it more practical for programmers to avoid glitches. It might also fit better into a denotational semantics for the FRP framework, but this remains to be confirmed.

Error Handling We have neglected proper error handling in our proposal, the API shown in this paper has no way of handling communication problems between server and client. An approach we have been thinking about is another extension of the interaction methods to incorporate a behavior that represents the current status(Connected, Pending, Disconnected) of the accompanying connection.

$$\texttt{Event}_c\texttt{[T].toServer: (Event}_s\texttt{[(Client, T)], Beh}_c\texttt{[Status}_c\texttt{])}$$
$$\texttt{Event}_s\texttt{[Client => T].toClient: (Event}_c\texttt{[T], Beh}_s\texttt{[Status}_s\texttt{])}$$

Another alternative may be to provide just two primitives $\texttt{networkProblem}_\texttt{c}$ and $\texttt{networkProblem}_\texttt{s}$, of types $\texttt{Event}_\texttt{c}[\texttt{ProbDesc}]$ and $\texttt{Event}_\texttt{s}[\texttt{ProbDesc}]$ that can be used for signaling problems of any connection.

Performance In our chat application example, we model the entire application state as a behavior on the server that we make available on the client side by replicating it into a client behavior. This has an impact on performance due to the nature of behavior updates. They are updated by replacing the current value with a newer value: $\texttt{Beh}[\texttt{T}] \approx \texttt{T} \times \texttt{Event}[\texttt{T}]$. In the case of our example this means that updates containing the entire chat log would be sent over the network to the client. It would be more efficient to send just the new chat messages to the client and reconstruct the full chat log on the client side from its previous value and the new messages received.

We want to approach this problem in the future using the concept of incremental behaviors. An incremental behavior is a behavior that can be considered as an initial value together with an event stream of increments: $\texttt{IncBeh}[\texttt{T}] \approx \texttt{T} \times \texttt{Event}[\texttt{T} \Rightarrow \texttt{T}]$. This approach for FRP has been researched before in Scala by Maier and Odersky (2013) and we think it could allow us to encode increments and sending them over the network instead of the entire state.

Persistence To support non-trivial web-applications, an FRP persistence API seems essential. We plan to investigate the design of an API and implement it by making use of the notifications of changes that some databases can send. As far as we know, this has not yet been attempted in an FRP API.

Scalability (Replication) Many modern web-frameworks support replicating the server part of a web application over multiple servers for scalability reasons. Because of the `fold`-related primitives in our API, our framework is stateful on the server, which makes it impossible to scale out using traditional replication. After defining a persistence API, we intend to investigate a solution for replication based on removing these primitives in the server API and instead only allow folding in the persistence API.

Security We make no claims for security of our implementation yet. The use of UUIDs and their possible predictability could be a security vulnerability. We only provide a means to distinguish between client connections and provide no methods yet for authentication and authorization.

8. Related Work

It is not feasible here to give a comprehensive overview of web development approaches. We therefore focus on work that is related to our proposal of multi-tier functional reactive programming for the web.

In order to deal declaratively with the asynchronous programming model of the web, we believe that we should avoid the use of imperative callbacks, i.e. event or request handlers that update some form of shared mutable state. As such, we do not go in details about much other work, even approaches that strive for declarative web programming like SUNNY by Milicevic et al. (2013) or Hilda by Yang et al. (2006) but in the end do not manage to avoid the use of imperative event handlers in some form. The main alternative to imperative callbacks is FRP, which we have introduced before.

Functional Reactive Programming What we have described during this paper is *our* view on functional reactive programming and corresponds closely to the original definition and those found in newer frameworks such as Flapjax and Elm. There has however been a lot of research done around this paradigm in the past. We do not have enough space to do the work justice and decided not to dedicate the larger part of this paper towards it. For descriptive work that reviews the status quo of FRP, providing highlights in some of the FRP specific problems such as glitch prevention, we recommend a survey by Bainomugisha et al. (2012).

As a side note, an anonymous reviewer pointed us to a paper by Jeffrey (2013) - on FRP with liveness guarantees - which ends with the words "It would be interesting to see if HTTP could be used as the communication protocol between FRP instances, and so build a distributed FRP implementation." This idea seems close to the topic of this text.

FRP for Client-side Web Development FRP in the web setting is not new. Flapjax first provided an FRP implementation for client-side web development in the form of a library and as a standalone language (Meyerovich et al. 2009). Like our proposal it targets Ajax-rich web development. A more recent web language based around FRP is Elm (Czaplicki 2012), it is mainly focused on GUI development and has rich APIs for canvas drawing. Unlike Flapjax, which treats behaviors and events as separate concepts, Elm merges the two and provides only one FRP primitive: *Signal*.

Multi-tier Languages Single-language web development is an idea that has been implemented several times; Ur/Web (Chlipala 2013), Hop (Serrano et al. 2006), Links (Serrano et al. 2006), Opa (MLstate 2007), MeteorJS (Schmidt et al. 2011) and Google Web Toolkit (Google 2006). Tools like Sunroof (Bracker and Gill 2014) or the already mentioned JS-Scala (Kossakowski et al. 2012) embed JavaScript as a DSL within other languages, arguably making them usable as multi-tier languages for the web as well. These tools provide one language for both server and client development, often including convenient extras such as data transfer without conversions.

Most of these projects follow the traditional JavaScript model of callback based interactivity. They often simplify client-server communication by providing RPC calls so that server functions can be bound to client actions without requiring manual boilerplate.

The projects most related to our work are the ones which also provide extra concepts to model reactivity: Ur/Web, MeteorJS and Opa.

Ur/Web provides a *source* that can be compared to the *EventBus* we discussed a few times before. You can create, push and get sources, only creating and pushing on sources is allowed on the server side, so that we get an RPC-style interface to a client-side FRP network. Subscribing to a *source* creates a *signal*, signals are composable in ways similar to traditional FRP.

MeteorJS provides several data sources on which you can subscribe; these include but are not limited to Session variables and Database queries. It is not possible to combine several data sources though it is possible to subscribe on the server as well.

Opa's reactivity implementation is similar to MeteorJS', it provides a client-subscribable and server-pushable communication channel that cannot be composed however in flexible ways needed for FRP.

Multi-tier Reactivity Several approaches to multi-tier reactivity have been made both in academia and industry.

Reactive Web is an FRP extension of the Lift framework in Scala (Gugenheim 2011). It uses a limited JavaScript DSL and provides a core library for events and behaviors. In the framework itself it is possible to create client events and have them available on the server. For example handling an event stream corresponding to a button on the client directly on the server by calling *.toServer* on the event stream. It however does not provide a means to go from server to client with behaviors or events.

JMacro-RPC (Bazerman 2012) is a Haskell library that also seems to provide a form of client-server FRP. It is built on top of JMacro (Bazerman 2009), a Template Haskell quasi-quoting function that allows embedding JavaScript code in Haskell. We have not been able to find sufficient documentation to assess the relation to our work, but from what we can tell, there seem to be important differences such as the fact that JMacro-RPC only seems to support client-side state, while we offer server-side stateful FRP primitives and hope to offer a persistence API in the future.

Other than specifically targeting web development, academia has also focused on the more general *distributed reactive programming* (DRP) with the aim of providing alternatives to the Observer pattern in a distributed environment. An overview of requirements and challenges of DRP is provided in *Towards Distributed Reactive Programming* by Salvaneschi et al. (2013).

An extension of AmbientTalk/R to combine the advantages of loosely-coupled publish/subscriber systems with the elegance of reactive programming constructs is explained in *Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks* by Carreton et al. (2010). They provide *ambient behaviors* which is a construct that allows the propagation of events to reactive values hosted on other FRP networks by means of publish/subscribe. An *ambient behavior* is a behavior that is subscribed to previously exported behaviors. Our approach can be compared to theirs by looking at `.to(Client|Server)` as a combination of export/subscribe. Since we assume a 'single server with multiple clients' architecture we greatly simplify our API, as a result we do not provide the flexibility that AmbientTalk/R provides.

Margara and Salvaneschi (2014) define a DRP approach that focuses strongly on consistency guarantees. They deliver three levels of consistency guarantees: causal, glitch free and atomic. Causal consistency refers to propagation that maintains causality properties within one process, e.g. e_1 happens before e_2 in the origin process and will only be able to be observed in that order by other processes. Glitch freedom means that a *partially* propagated FRP network is never observable. Finally, atomic is a consistency guarantee that delivers *total* FIFO ordering and glitch freedom and thus is the most expensive of them all.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, by the Research Foundation - Flanders under grant number G004321N, and by the IWT project SPION. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO).

References

E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012.

G. Bazerman. Hackage: jmacro: Quasiquotation library for programmatic generation of Javascript code. `http://hackage.haskell.org/package/jmacro-0.1`, 2009. (Visited on 03/19/2014).

G. Bazerman. Hackage: jmacro-rpc: JSON-RPC clients and servers using JMacro, and evented client-server FRP. `http://hackage.haskell.org/package/jmacro-rpc-0.2`, 2012. (Visited on 03/19/2014).

J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In *PADL*, volume 8324 of *LNCS*, pages 65–80. Springer International Publishing, 2014.

A. L. Carreton, S. Mostinckx, T. Van Cutsem, and W. De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *ICT*, pages 41–60. Springer, 2010.

A. Chlipala. The Ur/Web programming language family. online, 2013. URL `http://impredicative.com/ur/`.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296. Springer Berlin Heidelberg, 2007.

E. Czaplicki. Elm: Concurrent FRP for functional GUIs. Master's thesis, Harvard, 2012.

E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422. ACM, 2013.

S. Doeraene. Scala.js: Type-directed interoperability with dynamically typed languages. Technical Report EPFL-REPORT-190834, EPFL, 2013.

C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, pages 25–36. ACM, 2009.

C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

Facebook. React — a JavaScript library for building user interfaces. `http://facebook.github.io/react/`, 2013. (Visited on 03/19/2014).

Google. Google web toolkit. online, 2006. URL `http://www.gwtproject.org/`.

Google. AngularJS superheroic JavaScript MVW framework. `http://angularjs.org/`, 2010. (Visited on 03/19/2014).

N. Gugenheim. reactive-web. `http://scalareactive.org/`, 2011. (Visited on 03/19/2014).

A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP*, pages 126–150. Springer, 2010.

A. Jeffrey. Functional reactive programming with liveness guarantees. In *ICFP*, pages 233–244. ACM, 2013. .

G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript as an embedded DSL. In *ECOOP*, pages 409–434. Springer-Verlag, 2012.

J. Liberty and P. Betts. Reactive extensions for JavaScript. In *Programming Reactive Extensions and LINQ*, pages 111–124. Springer, 2011.

I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731. Springer, 2013.

A. Margara and G. Salvaneschi. We have a DREAM: distributed reactive programming with consistency guarantees. In *ICDES*, pages 142–153. ACM, 2014.

E. Meijer. Reactive extensions (Rx): curing your asynchronous programming blues. In *CUFP*, page 11. ACM, 2010.

L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA*, pages 1–20. ACM, 2009.

A. Milicevic, D. Jackson, M. Gligoric, and D. Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *Onward!*, pages 17–36. ACM, 2013.

B. Mizerany. Sinatra. `http://www.sinatrarb.com/`, 2007. (Visited on 03/19/2014).

MLstate. The Opa framework for JavaScript. `http://opalang.org/`, 2007. (Visited on 03/19/2014).

M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL*, pages 221–232. ACM, 2005.

B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360. ACM, 2010.

J. Paananen. baconjs/bacon.js. `https://github.com/baconjs/bacon.js`, 2012. (Visited on 03/19/2014).

T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.

J. Rudolph and M. Doenitz. spray — REST/HTTP for your Akka/Scala actors. `http://spray.io/`, 2012. (Visited on 03/19/2014).

G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In *ICC*, pages 226–235. Springer, 2013.

G. Schmidt, M. DeBergalis, and N. Martin. Meteor. `https://www.meteor.com/`, 2011. (Visited on 03/19/2014).

M. Serrano and C. Queinnec. A multi-tier semantics for Hop. *Higher-Order & Symbolic Computation*, 23(4):409–431, 2010.

M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.

F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, pages 32–32, 2006.