An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications

Kennedy Kambona
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
kkambona@vub.ac.be

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.ac.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.ac.be

ABSTRACT

JavaScript programs are highly event-driven, resulting in 'asynchronous spaghetti' code that is difficult to maintain as the magnitude programs written in the language grows. To reduce the effects of this *callback hell*, various concepts have been employed by a number of JavaScript libraries and frameworks. In this paper we investigate the expressiveness of two such techniques, namely *reactive extensions* and *promises*. We have done this by means of a case study consisting of an online collaborative drawing editor. The editor supports advanced drawing features which we try to model using the aforementioned techniques. We then present a discussion on our overall experience in implementing the application using the two concepts. From this, we propose a roadmap of how to improve support of programming event-driven web applications in JavaScript.

Categories and Subject Descriptors

D.1.5 [**Software Programming Techniques**]: Object oriented programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Javascript, Callbacks, Reactive programming, Promises, Event Streams, Behaviours, Futures

1. INTRODUCTION

JavaScript has been identified as the most dominant language in the web [5]. From its conception, the design of the JavaScript language was primarily targeted for the web browser [17]. As such, the main focus at the time was to provide ways to improve user interaction. The language therefore provides support for user interaction employing an event-driven style of programming.

Though the asynchronous nature of the language lends itself well to building responsive web applications, it also gives birth to a variety of challenges. The most notable one that has recently gained high recognition among JavaScript enthusiasts is the *asynchronous spaghetti*[16]: deeply-nested *callbacks* that have dependencies on data that have been returned from previous asynchronous invocations.

Since callbacks are common in JavaScript, programmers end up losing the ability to think in the familiar sequential algorithms and end up dealing with an unfamiliar program structure. A program might be forced to shelve program execution aside and continue when a return value becomes available in the unforeseen future while in the meantime execution continues performing some other computation. This structure forces programmers to write their applications in a nested hierarchy of callbacks, inevitably leading to what is termed as *callback hell* or the *pyramid of doom* [21].

In the next sections we present two increasingly popular approaches that have been proposed as a solution to these problems in the JavaScript context: reactive programming [9] and the concept of promises [10]. We describe these two approaches and apply them on a non-trivial online drawing application using the standard version of JavaScript, ECMAScript5[1]. At the same time we also compare the implementations when necessary and observe the lessons learnt from these experiences.

2. THE PROBLEM WITH CALLBACKS

Callbacks are used in event-driven programming to obtain results of asynchronous computations. Rather than an application blocking on a potentially indeterministic event, the execution semantics of the application shifts from the programmer, relinquishing control of the event when the invocation happens and registering a method with the intent of reacting when the event is performed.

Utilizing these callbacks in large-scale applications leads to a mix of collocated code fragments that are not easily composable and creates complex flows that force the programmer to pass the callbacks around in order to utilize their delayed values as illustrated in Listing 1. In this scenario of a chat application, there are three asynchronous calls: the first one registers a user to a chat server (line 1), the second asks the server for an available room to join (line 2) and the third broadcasts an initial chat message (line 3) to users in the room.

As a result, callbacks become hard to understand, particularly when the control flow of an application require handling

[1]As of this writing

Dyla '13, Montpellier, France

of multiple events together, or of events that depend on one another. The programmer is forced to write boilerplate code in order to orchestrate the asynchronous operations.

Listing 1: Nested Callbacks

```
1 registerToChatOnServer(username,function(rooms){
2   joinAvailableRoom(rooms, function(roomname){
3     sendChatToAll(roomname, msg, function(reply){
4       showChatReply(reply);
5     })
6   })
7 });
```

3. JAVASCRIPT PROMISES

One approach to deal with the problem imposed by callbacks has been in the use of *promises*. In 1976, Daniel Friedman and D. Wise [10] originally proposed the term *promise* as a proxy object that represents an unknown result that is yet to be computed.

Most implementations regard promises as *immutable* once-resolved promises as per the CommonJS Promises/A specification [12], which states that a promise can be rejected or resolved once *and only once*, to prevent unanticipated changes in behaviour. They furthermore encapsulate asynchronous actions, acting much like a value returned from the result of a computation only that the value may not be available at the time. Promises therefore represent the eventual completion and result of a *single* asynchronous operation.

The most common term used for JavaScript promises are the *thenables*, as a developer uses `then` to attach callbacks to a promise when it is either fulfilled or an exception is realised[2]. In line 1 of Listing 2 we use `Q.fcall` of the Q [21] library which creates a promise from the returned value of asynchronous invocations. It is important to note that `then` also *returns a promise*, implying that promises can be chained together. Listing 2 shows chaining of promises (lines 1-4) using the Q library. The listing performs similar functionality as the previous scenario of Listing 1, where predefined functions in lines 1-3 perform asynchronous requests to the server.

Listing 2: .then() in Q

```
1 Q.fcall(registerToChatOnServer)
2   .then(joinAvailableRoom)
3   .then(sendChat)
4   .then(function (reply){
5       showChatReply(reply)
6       },function (error){
7       // catch error from all async operations
8   })
9   .done();
```

3.1 Composing Promises

In most libraries, promises are higher-level abstractions. Therefore in addition to chaining, they can also be passed around and their operations composed easily – unlike asynchronous callbacks. A programmer can perform a number of high-level operations on promises such as composing a group of independent asynchronous calls as shown in Listing 3 in the Q library using `Q.all`. This listing computes the Fibonacci of two numbers through promises.

Listing 3: Composing Promises

```
1  fibpromise = Q.all([
2    computeFibonacci(n-1),
3    computeFibonacci(n-2)
4  ]);
5  fibpromise.spread(function (result1, result2) {
6    //resolver for both operations..
7    console.log(result1 + result2);
8  }),function(err){
9   //error occurred
10 });
```

To react to `fibpromise` we attach a resolver in line 5. The Q library provides `Q.spread`, which allows the arguments of the individual elements in a composition to be sent to the resolver. In this case, `fibpromise` awaits for both fibonacci sequences to be computed before it passes the results to the resolving function (line 5) which then sums the results (line 7).

3.2 Alternative Approaches

Within the domain of web development, there exist other frameworks which provide the abstractions for implementation of promises apart from the Q library. These include AngularJs[3], a JavaScript MVC-based framework for web applications, and Dart[4], a class-based web programming language.

The AngularJs promises were inspired by Q and are similar to Q's promises. However, the functionality provided by AngularJs' promises are a subset to those provided in Q in order to make them accustomed to the framework's execution style. It therefore lacks useful features such as promise chaining.

Dart promises (recently re-implemented as Dart futures), are also similar to Q's promises, with `Completers` used to manage the futures. Dart also contains a `Futures` interface for easily creating multiple future objects. We however did not use Dart since the aim of this experiment was to focus on mainstream technologies, i.e. JavaScript. Dart is a separate specialized language and currently lacks full support for integration with normal JavaScript applications.

Outside the web, promises are widely employed as abstractions in the domain of concurrent and distributed languages. The distributed, actor-based language AmbientTalk [6] uses similar concepts known as futures for non-blocking remote invocations, which can composed by grouping. Argus [13] provides *promise pipelining*, similar to the aforementioned promise chaining. Fignale [19] provides future *combinators* to compose delayed actions in a scatter/gather pattern.

3.3 Summary

Promises provide an elegant fall-back from the nested callback problem. JavaScript code that is hard to follow can potentially be replaced with a more structured design. In comparison to callbacks, promises provide a portable encapsulation mechanism capable of being passed around As a result, promises afford programmers the convenience of

[2]A third callback, a progress notifier for the promise can also be attached, as per the Promises/A specification

[3]http://angularjs.org

[4]http://dartlang.org

	Discrete Events	Continuous Events
Flapjax	Event Streams	Behaviors
Baconjs	Event Streams	Properties
ReactJs	Data Flows	Data Flows
RxJs	Observables	Observables
Dart	Streams	Streams
AngularJS	Bindings	-

Figure 1: Reactive JS Libraries: Terminology

	Explicit Lifting	Implicit Lifting
Flapjax	✔	✔
Baconjs	✘	✔
ReactJs	✔	✘
RxJs	✘	✔
Dart	✔	✘
AngularJs	✘	✔

Figure 2: Reactive JS Libraries: Lifting

handling them as first class objects, a great advantage given asynchronous mechanisms they represent.

4. REACTIVE JAVASCRIPT

The reactive programming paradigm has recently gained attention in the web development domain. This is because contemporary web applications communicate with servers and provide enhanced user interactions. As such, programmers need to deal with time-varying events involving various web-related concepts such as long-polling XHR requests [8], and persistent HTTP connections in Comet [4].

Reactive programming abstracts time-varying events for their consumption by a programmer. The programmer can then define elements that will react upon each of these incoming time-varying events. Furthermore, abstractions are first-class values, and can be passed around or even composed within the program.

Most reactive programming solutions support two kinds of reactive abstractions which model continuous and discrete (or sparse) time-varying values. The discrete time-varying values are most commonly referred to as *event streams*. These can be described are asynchronous abstractions for the progressive flow of intermittent, sequential data coming from a recurring event. For example, mouse events can be represented as event streams. On the other hand, continuous values over time are usually referred to as *behaviours* or *signals*. These can be thought of abstractions that represent uninterrupted, fluid flow of incoming data from a steady event. For example, a timer can be represented as a behaviour.

Support in JavaScript reactive programming is categorized by dedicated languages and extensions. Dedicated reactive languages require the programmer to develop their applications using the language that eventually compile to JavaScript for use in the web. Examples are Flapjax (as a language) and Elm [7]. Libraries and extensions provide additional constructs that extend the JavaScript language with reactive constructs when applied to various elements in a program – e.g. Flapjax (as a library) and Bacon.js [11]. Figure 1 shows the variations of terminologies in time-varying reactive abstractions in the web development context.

With these kinds of abstractions, programmers need not explicitly trigger a recomputation of time-varying data – reactive libraries extend JavaScript with automatic recomputation of reactive values in the program.

Consider as an example of the use of reactive programming a time-ticker in Flapjax [15], shown in Listing 4. Line 1 creates a timer behaviour that produces a value after every 100 milliseconds. Since the behaviour returns a value to microsecond precision, we divide the value by 1000 to show the time in seconds in line 3. We then insert the behaviour within the DOM element with the ID `timer-div` on a webpage, line 5. The value of the timer will therefore be continuously updated and displayed on the page.

Listing 4: A timer in Flapjax

```
1 var timer = timerB(100);
2 var seconds = liftB(
3   function (time){ return Math.floor(time / 1000);}
4   ,timer);
5 insertDomB(seconds, 'timer-div');
```

An obvious implication of this is that the values in expressions that depend on the reactive values need to be reactive themselves. If a variable gets assigned the `timerB` behaviour then the variable should be converted to a reactive variable itself, since updates from the behaviour will affect the variable. This process is known as *lifting*.

A number of JavaScript libraries implicitly perform lifting for the programmer – e.g. in Bacon.js. For some, the programmer has to perform lifting of reactive expressions manually – such as in React.js. A third category of libraries offer both implicit and explicit lifting – for instance Flapjax, which if used as a library the programmer perform lifting explicitly but if used as a compiler the code is transformed implicitly. This is illustrated in Figure 2.

In order to correctly recompute all the reactive expressions once a event stream or behaviour is triggered, most libraries construct a *dependency graph* [2] behind the scenes. Whenever an expression changes, the dependent expressions are recalculated and their values updated. In the timer example of Listing 4, a time change that happens from line 1 triggers the re-evaluation of the function in line 3 which subsequently updates the value inserted in line 5.

4.1 Composing Event Streams & Behaviours

For avoiding the callback nightmare described in Section 2 and maintaining such interactions in their programs, one useful abstraction for programmers is composing reactive abstractions that replace the asynchronous callbacks. For instance, instead of having three separate callbacks to separately handle mouse clicks, mouse moves or mouse releases, we can compose them as a single event stream which responds to all the three events. Most JavaScript libraries provide this kind of fundamental support. For example, in Bacon.js, properties (or behaviours) can be composed using the `combine` operator [11].

A different approach to composing event streams is through Flapjax's `mergeE`, where instead of combining two event streams, it always throttles the latest stream that comes in

from either event. We show an example in Listing 5. Here we create a timer event stream that is triggered after every 10 seconds (line 1), and another stream that is triggered whenever a user clicks on the `save-button` (line 2-3). In line 4 we create a `save` event stream that is triggered whenever either the timer elapses or the save button is clicked, calling the `doSave` function (line 5).

Listing 5: Flapjax: Merging Event Streams

```
1 var saveTimer = timerE(10000); //10 seconds
2 var saveClicked =
3   extractEventE('save-button','click');
4 var save = mergeE(saveTimer,saveClicked);
5 save.mapE(doSave); //save received data
```

4.2 Alternative Approaches

Within the web domain, there are other frameworks providing some reactive abstractions such as Dart and AngularJs, introduced in Section 3.3. AngularJs exposes some level of reactive programming semantics in its *data/variable bindings*. The framework's MVC architecture allows updates in models to trigger updates in client-side views, and vice-versa. This results in automatic synchronization of data between the model and view components. AngularJs therefore contains implicit lifting with bindings behaving similar to event streams. For the Dart language, event streams and behaviours are both known as `Streams`, to which a `Subscriber` can attach a `Listener`. Also, a programmer can attach several listeners to one event stream, which broadcasts its changes. Dart exposes constructs for programmers to explicitly lift their applications to use reactive programming in a separate Stream API.

Outside the web domain, functional reactive programming is common in fields related to event-driven and data-flow programming. FrTime [3], an extension to Scheme, consists of time-varying *signals* and *event sources* representing continuous and discrete values respectively. Native scheme functions called on these values are implicitly lifted, and the language supports functional operators such as `map-e` and `filter-e` akin to Flajax's `mapE` and `filterE`. Similar operators are present in Scala.React [14]. Scala.React provides `next` and `delay` dataflow methods that suspend calls until an event stream or a signal (i.e. behaviour) emits a value, after which the object continues execution. These can be used to orchestrate delayed calls and reduce complexity in programs with asynchronous semantics.

4.3 Summary

To summarize, support of reactive programming concepts in JavaScript can be distinguished as reactive languages that compile to JavaScript e.g. the Dart language, and libraries or extensions to the language that expose some reactive abstractions for programmers e.g. Bacon.js. These libraries also use slightly different terminologies of their abstractions, and some expose the important distinction between event streams and behaviours.

Existing web applications wishing to take advantage of reactive abstractions either need to rewrite their applications using dedicated languages and compiling them for the web, or manually lifting of various parts of their applications to use reactive semantics.

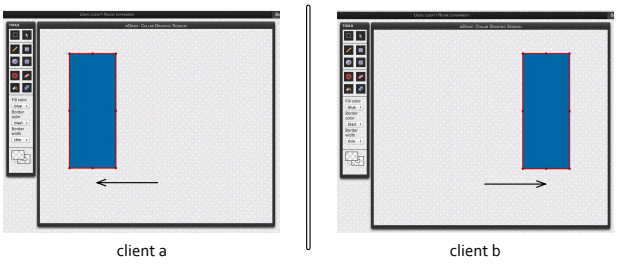

Figure 3: Drag operation by two clients a and b

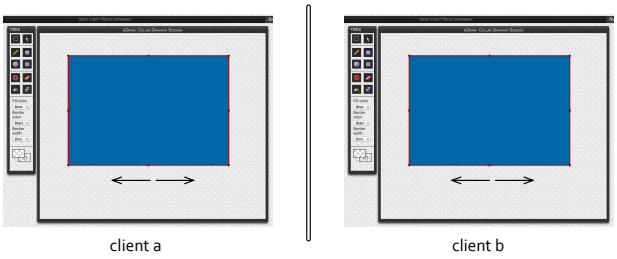

Figure 4: Result a distributed drag operation of Figure3

In the next sections we put these abstractions into practice for the implementation of a multi-user drawing editor. We then discuss the support provided and identify its shortcomings.

5. CASE STUDY

In this section we describe our implementation of a collaborative drawing application in the JavaScript language[5]. The application follows a multi-user, session-based approach where several users are connected via the web, sharing the same canvas.

In addition, the application incorporates some advanced distributed user interactions which employ composition of events such as a multi-user interaction. We present an example of such an interaction, consider two users participating in the same drawing session of the application. One user starts to drag a drawn shape to the right, while at the same time a different user in the same session drags the same shape to the left (Figure 3). Instead of the application recognizing two separate sequential interactions (e.g. drag-right and drag-left respectively), the application recognizes this as a single distributed resize interaction. As a result, instead of moving the shape, the application reacts by causing the shape to resize. The result is depicted in Figure 4 where the shape is stretched in both directions.

5.1 Architecture

We illustrate the architecture of our drawing application in Figure 5. To support a multi-user interaction mechanism through the web browser we employed a Node.js server since

[5]The source code can be cloned from GitHub at https://github.com/mtafiti/promise-and-react

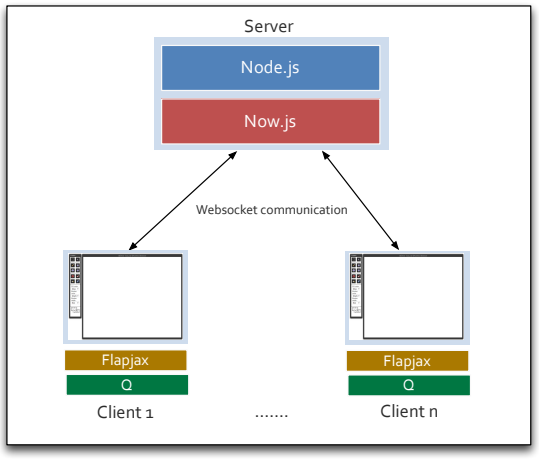

Figure 5: Application architecture

it provides a server-side environment for JavaScript.

On the client side, we implemented the drawing editor using the HTML5 canvas which supports primitives that provide rendering of elements through JavaScript operations. The effect of the choice of technology for the canvas is discussed later in Section 6. To sum up, both the server and the client (web browser) are implemented in the JavaScript language.

We use websockets to provide the interconnection between our application on the client side and the server side. However, in order to avoid manually maintaining the websocket connections between them we employed the NowJs package[6]. NowJs allows us to call methods on the server without having to explicitly manipulate data packets on a websocket connection. It also manages the inconsistencies in implementations of websocket communication across different browsers.

For providing abstractions for promises, we use the Q [21] library in the clients, whereas for the reactive abstractions we used the Flapjax library [15] in both the Node.js server and the client side.

5.2 Implementation

In this section we discuss the implementation of the drawing application with support for multi-user interactions. We highlight the most relevant parts in the use of futures and reactive programming, namely, supporting drawing sessions and implementing a multi-user interaction.

5.2.1 *Plain drawing editor*

The first step was to implement a plain drawing editor where users can draw shapes. The canvas listens to local mouse events from the user to capture and model interactions. It renders shapes in a canvas by implementing a refresh operation after a number of milliseconds. If there is any change in a shape's properties it is updated using the canvas' rendering engine.

[6]https://github.com/Flotype/now

5.2.2 *Composing local user interactions*

Next we modelled local user interactions as Flapjax event streams. For example, we have implemented a moving shape interaction by composing the action as a single event stream. More concretely, we model one complete drag gesture as a single, composed, merged event of a `mouse down` followed by a number of `mouse move` events, ending with a `mouse up` – as shown in Listing 6[7]. The `dragE` composed event stream now contains the merged streams of the three actions representing a complete drag event.

Listing 6: Moving a shape

```
1  function mouseDownAndMoveE (canvas) {
2   return extractEventE(canvas,"mousedown")
3     .mapE(function(md) {
4             return extractEventE(canvas,"mousemove")
5               .mapE(function(mm) {
6                 //...
7         });
8     });
9  }
10 function mouseDownMoveAndUp(element){
11    var downMoveAndUpE =
12     mouseDownAndMoveE(element)
13       .mergeE(mouseUpE(element));
14     return downMoveAndUpE;
15 }
16 var dragE = mouseDownMoveAndUp(canvas);
17 dragE.mapE(function(e) {
18     //update shape...
19 });
```

Given the combined `mergeE` event stream, line 18 attaches a handler that is triggered whenever the complete drag event is detected.

5.2.3 *Supporting drawing sessions.*

For joining a drawing session a client has to perform a sequence of events, similar to the example shown in Listing 1. First the client needs to initiate a registration request on the server, for some authentication. If acknowledged, the client subsequently requests for an available room to join. If successful, the client has joined a room, and sends an introductory message to the rest of the clients in the room.

Listing 7 shows the implementation of the above sequence of events. A client remotely signals to the server for registration through the function `registerOnServer` using the `now` construct of NowJs in line 1. The last argument to this invocation is a the callback to be executed once the asynchronous call returns. The server responds with the room list `rooms`, and the user proceeds to choose one room to join its drawing session (line 4). The client then sends a 'hi' message to all clients in the room (line 6). On the server we append the user's id to the list of users.

[7]We follow the conventions set by Flapjax of a suffix E to represent an event stream.

Listing 7: Joining a session - without promises

```
1  now.registerOnServer(name,function(res1,rooms){
2    if (res1 == true){
3    // ...
4     now.joinAvailableRoom(rmname,function(res2){
5           if (res2 == true){
6            var msg =“Hi everyone, I am ” + name;
7            now.sendChatToAll(roomname, msg,
8                function(response){
9                      console.log(response);
10               });
11          }
12    }
13   }
14 });
```

Notice already that there is an emergence of nesting and that orchestrating these nested asynchronous calls starts to be a burden to the programmer. This is a problem because most libraries supporting remote asynchronous invocations in JavaScript such as NowJs do not support promises. As a result, a programmer would need to interface remote asynchronous invocations with promises by manually creating and resolving promises once the asynchronous invocation returns.

To solve this issue, we added a layer of abstraction on top of NowJs by implementing a helper function `fnpromise` on the client side[8]. `fnpromise` (shown in Listing 8) assists in transforming callback-based functions which make invocations to our server through NowJs into promise-based functions. The helper takes the `now`-based function `fn` and creates and returns a promise (line 3 and line 14) that is resolved once the function completes (line 8) or is rejected if it returns an error (line 12).

Listing 8: Creating a promise-based function

```
1  function fnpromise(fn){
2   return function(){
3    var deferred = Q.defer();
4    fn.apply(this,arguments.slice().concat(cb);
5    //..
6    function cb (err, value){
7     if (error == null) {
8      deferred.resolve((arguments.length > 2) ?
9        Array.prototype.slice.call(arguments, 1) :
10       value);
11    } else {
12          deferred.reject(new Error(error)); }
13   }
14   return deferred.promise;
15  }
16 }
```

With the helper function, we can now chain our asynchronous callbacks with less effort. For our initialization operation, we pass our functions for registering a user, joining a room and sending an introductory message to be chained as promise-returning functions, illustrated in Listing 9.

[8]Inspired by http://bit.ly/trypromise

Listing 9: Joining a session - with promises

```
1  function err(msg){ console.log(”Error: ” + msg)}
2  p1 = fnpromise(now.registerOnServer);
3  p2 = fnpromise(now.joinAvailableRoom);
4  p3 = fnpromise(now.sendChatToAll);
5  p1(name)
6    .then(function(res){
7      return p2(res);
8    },err)
9    .then(function(rmname){
10     return p3(rmname, msg);
11   },err)
12   .then(function(response){
13     console.log(response)
14   },err)
15   .done();
```

`p1`, `p2` and `p3` in lines 2 and 3 respectively are promise-returning functions that can be chained together (or even composed if they were independent). If any errors are encountered then the chaining is aborted and the err function is invoked – line 7 for `p1` and line 10 for `p2`, displaying an error message to the console (in function `err`, line 1).

5.2.4 *Advanced distributed user interactions*

Distributed drag operation. For composing our multi-user operation described in the introduction of Section 5, the server orchestrates the drag operations performed by the users in order to identify whether two users are participating in a distributed action.

In order to realize this, we create a dictionary in the server `dictInteractions` that stores the interactions that clients are performing. On the client side, we create an event stream that sends a request to the server whenever the user performs a drag operation, such as the one already mentioned in Listing 6. Only this time in Listing 10 line 7 we register the drag on the server and proceed with the drag operation once this is completed, providing the logic to update the shape from line 8.

Listing 10: Multi-user drag interaction

```
1  function mouseDownAndMoveE (canvas) {
2   return extractEventE(canvas,”mousedown”)
3    .mapE(function(md) {
4
5     return extractEventE(canvas,”mousemove”)
6      .mapE(function(mm) {
7       now.registerDrag(shapeid, shape, function(){
8       //update shape on canvas
9       });
10     });
11  })
12 }
```

When the server receives notification that a client has started dragging by a call on `registerDrag` represented in line 1 of Listing 11, it proceeds to create an event stream `reqE` (line 6), which will sporadically trigger events on the client to proceed with the drag operation. The client then updates the shape information when the function `cb` is called (line 8).

Listing 11: Distributed interaction event stream

```
1  now.registerDrag = function (shpid, info, cb){
2    var reqE;
3    if  (! distInteractions .contains(this.user.id)){
4     reqE = distInteractions.getStream(this.user.id);
5    } else {
6      reqE = receiverE();
7      distInteractions .add(this.user.id, reqE, info);
8      reqE.mapE(function(val){
9        cb(val);
10     });
11   }
12   reqE.sendEvent(info);
13 }
```

The conditional in line 3 checks if the same client initiated a previous drag gesture. If so the server retrieves the event stream (line 4), in order to notify the client of the new update. If not, the server creates another event stream for this client and stores it in a dictionary containing all users and the interactions they are currently (in line 7). The server completes by triggering the event stream using the Flapjax primitive `sendEvent` as shown in line 11.

In the case where one client stops the drag interaction, the server receives this event and deletes the client's entry in the dictionary, completing the interaction.

Distributed resize interation. To detect a distributed resize operation, we need to modify the code in Listing 11. Consider that there is already a `client12345` that is participating in the drag operation. In the `dictInteractions` dictionary, we already have a client with the associated event stream and the last invoked arguments.

Listing 12: Distributed interaction invocation

```
1  if  (! distInteractions .isEmpty() &&
2        ! distInteractions .contains(this.user.id)){
3    var streams = distInteractions.getAllStreams();
4    // ...
5    streams.forEach{function(stream){
6      stream.sendEvent((isComposed:true, args:allinfo});
7    });
```

When another client contacts the server to perform a separate drag operation, the server looks up in the dictionary to see whether there is already a client performing a drag[9], as in line 1 of Listing 12. If one is present, then all clients should be notified that a distributed resize has been realized, and thus we retrieve all streams and trigger their events to all clients. The server triggers the events with a status value which indicates to the client that the event should be handled as a composed interaction, as shown in line 2 of Listing 13.

Listing 13: Detecting distributed interaction - client

```
1  now.registerDrag(shapeid, shape, function(res){
2     if  (res.isComposed){
3       //distributed  resize  realized
4     } else {
5       //normal drag with res.arg
6     }
7  });
```

[9]For clarity, we omit an additional check *of the same shape* in the code listing

5.3 Summary

We have been able to represent an advanced user interaction in our distributed drawing application with the help of reactive programming and promises. Although they provide good abstractions for reacting to events in an elegant way, with current approaches the programmer still has to manually encode advanced interactions based on the incoming events that these abstractions represent and to manually interface remote asynchronous invocations with promise abstractions.

6. LESSONS LEARNT

Using reactive and promise-based abstractions allows us to de-clutter our code from nested asynchronous callbacks. It also allows us to represent events, which are common in JavaScript programs, as first-class citizens. This is particularly important in structuring and maintaining programs as opposed to implementing them using native asynchronous callbacks.

Nevertheless, in the process of developing our drawing application we have made observations regarding challenges with the programming support in Q and Flapjax, as well as other observations pertinent to application development in JavaScript. We summarize the challenges in this section.

Dependent promises. Although promises offer a convenient way of representing and composing callbacks, we observed a problem when chaining a number of promises. As pointed out by the creators of the Q library[21], we found that nested promise resolve and reject callbacks were still present whenever we had chained promises dependent on results of previous promises.

This dependence exposes a different manifestation of the asynchronous spaghetti code problem, only this time it is applied to promise resolver and rejecter functions. As a concrete example, consider Listing 14. Here, we create three promises where each successive function depends on data from the preceeding functions. After registering to a random drawing room in line 5 and adding a client to a room in line 7, we now want to load the drawing canvas data for the room. For this we need the data from the first two invocations, leading nest the `thens` as in line 8-11.

Listing 14: Nested promises

```
1  function authenticate() {
2    p1 = fnpromise(now.registerRandomRoom);
3    p2 = fnpromise(now.addClientToRoom);
4    p3 = fnpromise(now.loadCanvasData);
5    p1(nickname)
6      .then(function(roomname){
7          return p2(clientId, roomname);
8          //nested since we need client id and roomname
9          .then(function(noofusers){
10             return p3(clientId, roomname);
11         })
12     })
13     .done();
14 }
```

Lifting non-reactive components. The methodology used to implement the drawing application was through an incremental development process. We started from building a vanilla drawing editor and added distributed functionality such as the distributed drag and resize operations in an incremental way. As such, we performed manual lifting of

various components of our drawing application in order to support the reactive programming semantics.

While Flapjax and other similar libraries have support for lifting non-reactive components, we observed that this process was unnecessarily complex as opposed to starting out with a reactive language or library that performs implicit lifting of various reactive elements in the program.

Event streams vs. Behaviours. During our implementation of the reactive components of the drawing application it became apparent to us that event streams were better suited for the reactive implementation of our scenario. This is because an event stream can update its dependent values whenever the event itself is triggered, unlike behaviours which can trigger recomputations continuously. This distinction is particularly important when abstracting remote events and user interaction as reactive components, improving on the responsiveness of the application.

Distributed event streams. In our implementation we modelled event streams as reactive components that listen to events coming from a websocket and expose them to listeners. We create separate event streams on the client and the server and trigger the event stream when data from the websocket is detected. This manual management of events coming from distributed sources with separate event streams within one application shows that there lacks distributed abstractions for event streams.

Inconsistencies in promise implementations. During the implementation of the drawing application, we tested several libraries providing support for the concept of promises, such as jQuery [18] and FuturesJS [1]. Although they provide the same concept, the different implementation strategies with respect to the Promises/A specification [12] may have an impact on the effects of an application. For instance, jQuery's deffereds have a slight difference in propagating errors in chained promises whenever a rejected non-promise value is returned. This may lead to undetected rejections or exceptions when chaining promises in an application.

The choice of the drawing mechanism. When considering a drawing GUI mechanism that is supported in HTML an initial solution is to employ DOM-tree elements, such as SVG (Scalable Vector Graphics). Most JavaScript libraries supporting reactive programming abstractions have support for DOM-tree elements. However, SVG models drawing elements as DOM elements with special constructs for rendering shapes on a web-page. Most reactive programming libraries in JavaScript do not support these advanced types of DOM elements.

For our case study we therefore used a HTML5 canvas as the drawing component of choice. This in effect means that reactive support is only for the top-level canvas DOM element since a HTML5 canvas does not not represent its items in the web-page's DOM tree[10]. We therefore could not model the shapes drawn by the user as reactive objects natively. This choice resulted in having to model operations performed by the end-user as manual event streams, as in Listing 10.

7. RESEARCH ROADMAP

In this section we present the research possibilities that this experience has availed, while reflecting on some of the research challenges that were encountered.

Chained Promises Hell. From the previous section we identified that promises that depend on values from previous promises in the chain are still prone to nesting. Using the Q library, we noted that dependent, chained promises can lead to the pyramid of doom and other complexities that are attributed to orchestrating asynchronous callbacks. Techniques alleviating this specific type of problem evident when manipulating promises are a possible research path to follow.

Improving Explicit Lifting. It is apparent that existing JavaScript applications incorporate reactivity in their development to add reactive elements to it. This forces programmers to perform manual lifting, a process which this experiment has identified as having potential to be convoluted. A possible research area could be to investigate further support for lifting, such as providing a number of lifting operators.

Distributed Reactive Programming. We have mentioned how we modelled event streams listening to changes from our websocket abstractions on both the server side and the client side. To avoid this manual implementation, an alternative approach can be to provide a reactive abstraction that traverses both the server and the client. This approach could even go further and implement a reactive abstraction that traverses from client to client (e.g. browser to browser), abstracted over a server.

Event streams and behaviours in practice. We have mentioned our experience of modelling most of our reactive elements as event streams. As a matter of fact, our only implementation of a behaviour was a timer at the client which was calmed down to trigger after every second, essentially becoming an event stream. In the end, we are left questioning the practicality of behaviours in large-scale reactive implementations of JavaScript applications.

8. CONCLUSION

Callbacks have been accused of being the modern 'goto' statements in asynchronous programming. The use of promises and reactive programming abstractions has been proposed as a solution to dealing with these kinds of complexities introduced by callbacks.

While reactive programming is more natural in programming web applications due to its asynchronous nature, it lacks support for distribution of reactive abstractions it exposes. On the other hand, although promises are a positive step in eliminating the problems encountered when orchestrating a number of asynchronous callbacks, they introduce problems when used with dependent values leading to nested resolving functions.

In this paper, we have reported on our experiences in developing a collaborative drawing application. The experience has highlighted some of the less obvious issues that a programmer may encounter. In addition, we have described some of our observations of the development process, complete with possible problems and some research areas that could uncover solutions in the future. Investigating some of these problems may open up research areas that may prove to be vital in the JavaScript world, given the prominence of programming with asynchronous semantics due to its distributed architecture.

[10]It can be viewed as a rendering component with no support for a *scene graph* [20]

9. ACKNOWLEDGMENTS

The authors would like to thank the reviewers and all contributors to this work in the Software Languages Lab, Vrije Universiteit Brussel, with special thanks to Prof. Dr. Tom Van Cutsem.

10. REFERENCES

[1] AJ O'Neal. Futuresjs. `https://github.com/coolaj86/futures`, 2010.

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

[3] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.

[4] D. Crane and P. McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, Berkely, CA, USA, 2008.

[5] D. Crockford. The world's most misunderstood programming language has become the world's most popular programming language. `http://goo.gl/gu2Sd`, 2009. [Online; accessed 13-April-2013].

[6] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[7] E. Czaplicki. The Elm Programming Language. `http://elm-lang.org`, 2011. [Online; accessed 26-March-2013].

[8] R. Dolan. Long-Poll Server. `https://sites.google.com/site/gopatterns/web/long-poll-server`, 2009. [Online; accessed 04-April-2013].

[9] C. Elliott. Composing reactive animations. *Dr. Dobb's Journal*, 1998.

[10] D. Friedman and D. Wise. *The Impact of Applicative Programming on Multiprocessing*. Technical report (Indiana University, Bloomington. Computer Science Dept.). Indiana University, Computer Science Department, 1976.

[11] Juha Paananen. Bacon.js. `https://github.com/raimohanska/bacon.js`, 2011.

[12] M. Kowal. Q Library. `https://github.com/kriskowal/q`, 2009. [Online; accessed 18-March-2013].

[13] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, Mar. 1988.

[14] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[15] 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 *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[16] T. Mikkonen and A. Taivalsaari. Web applications: spaghetti code for the 21st century. Technical report, Mountain View, CA, USA, 2007.

[17] C. Severance. Javascript: Designing a language in 10 days. *Computer*, pages 7–8, 2012.

[18] The jQuery Foundation. Deferred Object – jQuery. `http://api.jquery.com/jQuery.Deferred/`, 2011.

[19] Twitter Inc. Fignale. `http://bit.ly/fignalefutures`, 2011.

[20] A. E. Walsh. Understanding scene graphs. `http://www.drdobbs.com/jvm/understanding-scene-graphs/184405094`, 2002. [Online; accessed 22-April-2013].

[21] K. Zyp. Promises/A Specification. `http://wiki.commonjs.org/wiki/Promises/A`, 2009. [Online; accessed 27-March-2013].