# Enriching the Internet By Acting and Reacting

Sam Van den Vonder
Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium 1050
svdvonde@vub.ac.be

Florian Myter
Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium 1050
fmyter@vub.ac.be

Joeri De Koster
Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium 1050
jdekoste@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium 1050
wdmeuter@vub.ac.be

## ABSTRACT

A recent trend in application development for the web is a move towards *rich internet applications* (RIAs). As more and more of the application logic is moved to the client, RIAs can benefit from concurrency in order to increase overall performance as well as responsiveness of the application. Additionally, RIAs are often written in an event-driven style of programming to react to incoming events of a multitude of services that are integrated within the application. In this paper we argue that, while individual technologies exist to tackle both concerns, these technologies cannot easily be integrated in an ad hoc way. To increase the modularisability and composability of RIAs we propose a new programming model based on *actors* and *reactors* that encapsulate different parts of the application. We show that our model is able to exploit some of the available concurrency while reducing the required amount of event-driven code. Both actors and reactors are modular components that can be glued together via a unified communication mechanism. We evaluate our approach by means of a motivating example of a collaborative code editor.

## CCS CONCEPTS

• **Software and its engineering → Data flow languages**; **Concurrent programming languages**; *Distributed programming languages*; *Frameworks*;

## KEYWORDS

Reactive programming, Actors, Reactors, Rich internet applications, Distributed programming, JavaScript

**ACM Reference format:**
Sam Van den Vonder, Florian Myter, Joeri De Koster, and Wolfgang De Meuter. 2017. Enriching the Internet By Acting and Reacting. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079407

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*Programming '17, Brussels, Belgium*
© 2017 ACM. 978-1-4503-4836-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079407

## 1 INTRODUCTION

The Internet's transition from a thin to a thick-client model has led to a wide variety of new web-based applications commonly called rich internet applications (RIA). Examples of RIAs include web applications (Gmail, Office Web Apps), desktop applications (Slack, Visual Studio Code), mobile applications (Facebook, Discord), etc. One can distil the essence of a RIA in two characteristics:

**Computationally Intensive** There has been a trend in modern day application development towards RIAs that are increasingly required to execute long lasting computations. Moreover, long lasting computations are expected not to impede the responsiveness of the application. Examples include in-browser games rendering graphics, online code editors performing syntax highlighting, etc.

**Interactive and Event-driven** To provide a semblance of interactivity, web applications used to refresh entire pages in order to provide users with novel information. RIA programmers are stepping away from this methodology by implementing their applications as single page instances. This single page can be divided into multiple parts, where each part is responsible for one of the application's functionality. Each of these parts is to independently react to a plethora of events, which can either originate from external sources (e.g. a server pushing new data) or from user interaction (e.g. mouse events). As such, RIAs are highly event-driven.

As is the case for the majority of the web, the predominant language used to implement RIAs is JavaScript. Given JavaScript's applicability to both client as well as server-side web development (e.g. via Node.js [4]), RIAs are often entirely implemented in it. However, JavaScript lacks constructs allowing programmers to elegantly deal with the two characteristics of RIAs listed above.

**Concurrency** The computationally intensive nature of RIAs requires the use of concurrency. To this end JavaScript provides either web workers for client-side development or child processes for server-side applications. Both are semantically similar to the actor model: each worker or child process runs in its own thread of control (dedicated program threads in the case of web workers and child processes), they lack shared state, and communicate exclusively via asynchronous message passing. However, as detailed in [10], both constructs are limited in the expressiveness of their API. By default it is only possible for workers to communicate in a parent-child relationship where the parent is the spawner and the child is the spawned worker. References to workers are not first class and can therefore not be exchanged. Furthermore, since workers have no shared state the developer must manually serialise and deserialise objects to pass them between workers. This is often difficult due to objects referencing their lexical scope, and the process of (de)serialisation is error-prone. Finally, while both constructs are applied in an inherently distributed context, they lack built-in communication capabilities (i.e. web workers and child processes cannot natively exchange messages).

**Reactivity** Due to the event-driven nature of RIAs their source code is often riddled with callbacks to handle various events. However, implementing event-driven applications using the observer pattern or callbacks is known to cause a number of issues [7]. Most notably it inverses the control flow: instead of being lexically determined, the flow of a program is now determined by the order of incoming events [5]. Moreover, callbacks are not composable and must therefore be nested leading to the "callback hell" [3], which further complicated code readability and maintainability. The reactive programming paradigm [1] is an elegant solution to the aforementioned problems. It allows programmers to declaratively specify how events should be handled and composed, leaving the underlying language or framework with the responsibility of tracking changes to event sources and updating the application state. In this paradigm events are reified as first class citizens called *signals*. Using signals in function applications creates dependencies that are managed by the reactive runtime. When the occurrence of an event changes the value of a signal, the reactive runtime uses the dependencies to update the application state accordingly.

RIAs require both concurrency *and* reactivity. Web workers can be used to some extent to improve concurrency within an application, and reactive programming frameworks alleviate some of the issues with event-driven programming. However, in this paper we argue that the sum of both parts does not equal the whole. Solving both problems within a single programming framework requires careful integration of both programming models.

In this paper we propose a model to develop RIAs which elegantly tackles *both* concerns. Our solution is focussed around two abstractions, namely *actors* and *reactors*. Concurrency is provided by actors, and reactivity is provided by reactors. We provide an implementation of our model in JavaScript. The actor portion of our model is an extension of the Spiders.js library [10], which is an enrichment of standard web workers. Reactors are a new abstraction that encapsulate a reactive program written using the Reactive Extensions library for JavaScript (RxJS) [11]. Our model is innovative in the following ways:

- To the best of our knowledge this is the first instance of a model that unifies reactive programming with actors. To achieve this we extend actors with a publish/subscribe mechanism, and introduce a new reactor abstraction that supports reactive programming.
- Our model and implementation brings high-level support for concurrency to reactive programming by directly mapping actors and reactors to web workers.
- Actors and reactors can be distributed over client and server. In other words: besides concurrency, our model supports distributed reactive programs in the context of the web.

## 2 PROBLEM STATEMENT

There already exist a number of technologies to exploit concurrency within a RIA. The same is true for enabling a reactive style of programming. However, we argue that enabling both within the same application requires careful integration of both programming models. Our programming model provides language constructs for exploiting concurrency using actors, and reactivity using reactors. Both actors and reactors are modular components that can be glued together via a unified communication mechanism. This section gives a brief overview of the available technologies and motivates the need for a unified programming model.

### 2.1 Concurrency

For enabling concurrency, JavaScript provides either web workers for client-side development or child processes for server-side applications. However, it has been shown that web workers have some severe limitations that restrict their use for developing RIAs [10]. Firstly, web workers are restricted to hierarchical communication. By default the only type of communication that is supported between workers is bi-directional message passing between the spawner of a worker (the parent) and the worker itself (the child). In other words, references to web workers are not first class citizens and cannot be exchanged, and thus there is no built-in support for communication among children. As such, developers have to modify their application to fit the parent-child relationship. Secondly, web workers have no shared state. All non-primitive datatypes passed in a message must be manually serialised on the sender side and deserialised on the receiver side. This can easily lead to serialisation errors, either due to objects that reference items in their lexical scope, or copies of objects that are not synchronised. Lastly, while web workers are actor-like constructs that operate in a distributed context, client to client communication or client to server communication is not supported.

The aforementioned issues were solved in an actor library that implements the Communicating Event Loop (CEL) actor model [9] on top of web workers [10]. This actor framework enables the creation of actors both on the client as on the server side, and unifies the communication mechanism between those actors. However, this framework does not support reactive programming, and as such does not compose well with reactive programming libraries or frameworks. For this paper we extended this framework with new constructs that enable the specification of reactive components that can be composed with actors.

### 2.2 Reactive Programming

There exist a number of frameworks for reactive programming on the web such as Elm [2], Facebook React [6], and RxJS [11]. A straightforward approach for enabling concurrency within a reactive application would be to simply spawn web workers from within the reactive program to offload heavy computations. However, web workers are typically not part of the reactive execution graph. As such, an ad hoc integration of web workers within any of these technologies requires manual propagation of updated values to the web workers and vice versa. The only exception to this is an extension of RxJS where spawning a new web worker returns a special value called a *subject*. Contrary to regular reactive values,

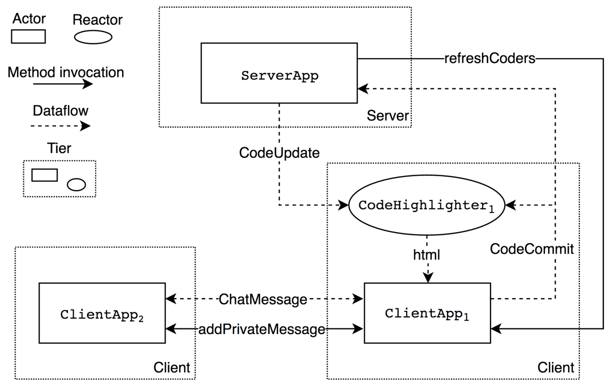

**Figure 1: Communication/dataflow diagram of the CoCode application with two clients (one client fully drawn).**

```
1  class ClientApp extends actorreactor.Application {
2   initialize() {
3     this.coders = new clientManager.ClientStore();
4
5     codeCommits.broadcastAs("CodeCommit");
6     publicChatMessages.broadcastAs("ChatMessage");
7     publicChatMessages.subscribe(chatMessage =>
          this.addChatMessage(chatMessage));
8   }
9   refreshCoders(coders) {
10    let references = coders.getClientReferences();
11    coders.getClientNames().forEach((name, idx) =>
          this.addCoder(name, references[idx]));
12  }
13  addCoder(name, coderReference) { /* Omitted until later */ }
14  sendPrivateMessage(receiver, message) { /* invoke
        addPrivateMessage on receiver */ }
15  addPrivateMessage(sender, message) { /* GUI */ }
16  addChatMessage(message) { /* GUI */ }
17  updateCode(rawHTML) { /* GUI */ }
18 }
```

**Listing 1: Defining the CoCode client actor**

anyone with a reference to a subject can push new events to it. A subject created specifically for a web worker forwards any values it receives to the mailbox of the worker, and conversely any values received from the worker are pushed on the asynchronous stream of the subject. However, while RxJS allows developers to spawn workers, it does not provide any facilities to compose them, and the web workers are still subject to the issues discussed in Section 2.1.

Another approach to enable concurrency within a reactive application would be to parallelize [2] or distribute [8] the execution of reactive program. While this is certainly possible, both approaches focus on fine-grained reactive abstractions, whereas our aim is to provide a more coarse grained concurrency mechanism that is more suitable for the coarse grained concurrency model of JavaScript. Our approach abstracts over reactive parts of an application as modular components that each have their own execution environment. Each of these reactive components, called reactors, can be glued together with the active parts, called actors, using a unified communication mechanism.

## 3 ACTORS AND REACTORS BY EXAMPLE

To demonstrate actors and reactors we implemented an application that is a typical for rich internet applications. CoCode is a collaborative code editor that features automatic syntax highlighting and peer-to-peer chat. All users connected to the application share the same view of the code that is being edited. Users can easily edit and commit new code to the server to share it with the other users. In order to reduce network traffic and increase the efficiency of the application, code is sent in its raw form rather than the syntax-highlighted HTML which is significantly larger in size. In practice this entails that syntax highlighting must occur on the client, preferably concurrent to the rest of the application in order to avoid freezing the browser window. Furthermore, users can communicate via a peer-to-peer private and public chat.

Figure 1 depicts the communication- and dataflow diagram of the CoCode application with two clients. Note that for clarity we omit many aspects of the leftmost client, which are effectively the same as the rightmost client. Without going into the details of communication and dataflow, the diagram shows the general structure of the application, which is split up in a client- and server tier. The main entry point of both client and server is always defined as an actor, namely `ServerApp` and `ClientApp`. These entry points can be used to retrieve a reference to other actors (e.g. the client contacting the server), spawn additional actors or reactors, and specifically on the client they have exclusive access to the DOM. On the client we encapsulate the syntax highlighting service in a reactor, since syntax highlighting is always a reaction to new code becoming available, either locally or as an update pushed by the server. Furthermore, the server provides every client with references to the other clients, enabling peer-to-peer interaction and data flow.

### 3.1 Actors

Every application consists of at least one actor that is used as the main entry point of the application. An actor is represented as a JavaScript ES6 class where the methods of the class define the interface of the actor, and spawning an actor from a class returns a reference to the new actor. References are first-class citizens and can be used to invoke methods on the actor using the standard dot-notation. Our actor implementation reuses these features from the Spiders.js library, and extends them with the ability to *broadcast* or *publish* arbitrary values. For now we do not discuss the complementing feature to integrate publications in the rest of the program, and leave this for Section 3.3.

Listing 1 implements the main entry point of the CoCode application that runs in the browser. We omit code related to interfacing with the DOM and the sending of private messages, as to not distract from the main contributions. The interface of `ClientApp` defines 7 methods that are mostly related to interaction with the DOM. The initialisation method (defined on line 2) first sets up a local store for keeping a reference to the active users in the application (line 3), which will be used for implementing the private chat. Since the main application actor is responsible for the DOM, it integrates with the RxJS library. Both `codeCommits` (line 5) and `publicChatMessages` (line 6, 7) are RxJS streams of values that originate from the user interface, which produce strings that contain code and chat messages respectively. We extend standard RxJS streams with a `broadcastAs` operator that automatically broadcasts new values on the stream.

For instance, when the user click the 'commit' button, the raw code is automatically broadcasted by the actor with the `CodeCommit` label. Finally, line 7 causes a local method invocation whenever a new value appears on the RxJS stream of public chat messages, such that a message send is also reflected in the local user interface.

When the client contacts the server (not shown in Listing 1) it receives a list of all active clients, handled by `refreshCoders` on line 9. Here we add every active user to the local client store via the `addCoder` method which we omit until Section 3.3. To summarise the role of the client actor in the application: it is the main entry point of one client, which broadcasts raw code and public chat messages.

## 3.2 Reactors

Reactors are a new abstraction that encapsulate a fully capable RxJS reactive program. Reactors extend the built-in `Reactor` class and must implement two methods: `imports` and `react`. The `imports` method is called during initialisation to load external scripts in the execution environment of the reactor. The `react` method implements the main body of the reactor as an RxJS program. A reactor is spawned with a static set of inputs (other actors/reactors) whose output is made accessible in the body of the reactor as RxJS streams. These streams are used to invoke the `react` method *once* to construct the RxJS dataflow graph.

Unlike actors, reactors do not communicate via method invocations, and instead they are exclusively data-driven. Reactors can broadcast the results of their computations (similar to actors), which can be used by other actors and reactors. In the context of our CoCode application, to prevent the browser from freezing we offload syntax highlighting of code to a separate process. Since syntax highlighting is always reaction to new data (unhighlighted code), we encapsulate it in a reactor that takes raw code as input and publishes syntax highlighted code.

Listing 2 implements the code highlighting reactor that receives input from two sources, namely code from the local application, and code from the server (i.e. other users). In both cases the code is syntax highlighted and broadcasted. On line 2 we import a library that handles the computationally intensive syntax highlighting, and the `react` method on line 4 implements the main body of the reactor. The arguments given to `react` (`localCode` and `serverCode`) directly map to the inputs given to the reactor when it is spawned. Both `localCode` and `serverCode` are asynchronous streams of values in the RxJS paradigm. Every time the reactor receives a publication from another component, the publication is translated by the reactor to an event on the corresponding asynchronous stream. This allows developers to make use of the extensive set of purely functional RxJS stream operators such as map, filter, and concat to compute new values from the input. This implies that the body of a reactor is also completely functional, as is often the case for reactive programs. The reactor in Listing 2 receives code publications from both the local client and the server, and merges the two streams on line 5. The raw code of both sources is then syntax highlighted on line 6 by applying the `highlightAuto` function to each element on the stream. Finally, line 7 extracts the syntax highlighted code as HTML from the resulting object, which is broadcasted by the reactor with the `html` label. Similar to the `Application` actor of Section 3.1, the `broadcastAs` operator is automatically added to the RxJS library to integrate RxJS streams with reactors.

```
1  class CodeHighlighter extends actorreactor.Reactor {
2   imports() { importScripts('highlight.js'); }
3
4   react(localCode, serverCode) {
5     localCode.merge(serverCode)
6       .map(code => self.hljs.highlightAuto(code))
7       .pluck("value")
8       .broadcastAs("html");
9   }}
```

**Listing 2: Defining the CoCode syntax highlighting reactor**

```
1  let app = new ClientApp();
2  app.remote("127.0.0.1", 8000).then(server => {
3    app.initialize();
4    server.registerClient(myName, app);
5
6    let newClientSource = [server, "NewClient"];
7    app.reactTo(newClientSource, "addCoder");
8
9    let codeSources = [[app, "CodeCommit"], [server, "CodeCommit"]];
10   let hjsService = app.spawnReactor(CodeHighlighter, codeSources);
11   app.reactTo([hjsService, "html"], "updateCode");
12 });
```

**Listing 3: Initializing the client application**

## 3.3 Reactive Program Composition

The CoCode client and code highlighter of Listings 1 and 2 are defined as standalone components that perform a well-defined function. However, in order to create meaningful applications we must be able to compose them. We previously discussed that actors and reactors can publish values with a certain topic. To complement this, actors and reactors can subscribe to these publications and react to new values becoming available.

Listing 3 shows the initialization code of the CoCode client which wires together the different components. On line 1 the client of Listing 1 is created, which is then used to obtain a reference to the server actor (line 2). Since `remote` returns a `Promise`, we install a callback that is invoked when the promise resolves with a reference to the server. On line 3 the client is initialized, and registered with the server on line 4.

Whenever a new client connects to the application, it is broadcasted by the server with the `NewClient` label. Lines 6 and 7 set up the client actor to react to these publications via a `reactTo` primitive that is defined on actors. Whenever a new client is published, the `addCoder` method of `app` is invoked with the received values. Since publications are translated to method invocations, it is easy for actors to add and remove dependencies at runtime.

Reactors require a different way of reacting, since the dependency graph of a reactive program is often static. We therefore create reactor dependencies when spawning the reactor, and do not allow them to change at runtime. The highlighting service of the CoCode application reacts to both code produced by the local application as well as code produced by the server (line 9). Line 10 spawns an instance of the highlighting service with a reference to these input sources. Finally, on line 11 we create a dependency between the output of the highlighting service and the client such

```
1  addCoder(name, coderReference) {
2   if (this.name !== name) {
3     this.coders.addClient(name, coderReference);
4     this.reactTo([coderReference, "ChatMessage"],"addChatMessage");
5  }}
```

**Listing 4: Implementation of addCoder of Listing 1**

```
1  class ServerApp extends actorreactor.Application {
2   constructor() {
3     super();
4     this.clients = new clientManager.ClientStore();
5   }
6   registerClient(name, clientReference) {
7     this.clients.addClient(clientReference, ref);
8     clientReference.refreshCoders(this.clients);
9  
10    let codeSource = [clientReference, "CodeCommit"];
11    this.reactTo(codeSource, "receiveCodeCommit");
12 
13    this.broadcast("NewClient", name, clientReference);
14  }
15  receiveCodeCommit(rawCode) {
16    this.broadcast("CodeCommit", rawCode);
17 }}
18 new ServerApp();
```

**Listing 5: Initializing the cocode server application**

that newly highlighted code causes an invocation of updateCode on the client.

An interesting consequence of composition via publish/subscribe is that it allows new dependencies to be added at runtime, be it via spawning a new reactor, or adding a new reaction for an existing actor. We can use this feature to implement the public chat. Listing 4 implements the addCoder method of the client actor which we previously omitted. The method is invoked every time a new client joins the application. Besides adding the new client to the local client store (line 3), a new reaction is set up such that the local actor reacts to public chat messages that are published by the new client (line 4). Whenever a new chat message is received, the addChatMessage method is invoked with the message. Similarly, since the newly added client also receives a list of all clients in the application, it will also react to chat messages from the local client.

## 3.4 Distribution

An important aspect of the CoCode application is interaction with a server, and in Section 3.3 we discussed how the server can be contacted, and how the client can react to publications of the server. Listing 5 shows the implementation of that server as an actor. Similar to the Application actor on the client, the server extends the built in Application class since it is the main actor that serves as the main entry point of the server. The server has a simple interface with two methods: registerClient and receiveCodeCommit. When a new client joins the application it is added to the local client store (line 7), which is then sent to the new client (line 8). To receive code updates from clients, the server reacts to their CodeCommit publications (lines 10-11) and handles them via its receiveCodeCommit method, which simply rebroadcasts the code. Finally on line 18 we create an instance of the server to start the application. The takeaway message is that actors and reactors can easily react to other components, regardless of where an actor or reactor has been spawned. By using a publish/subscribe mechanism for reactive input/output of components we provide developers with the means to glue together distributed components, without hard-wiring dependencies in the components themselves. This loose coupling makes it easy to add, remove, and replace components in the application without modifying the dependencies themselves.

| LOC | Forwarding | Handling | Total |
|---|---|---|---|
| Native | 86 (67%) | 39 (30%) | 128 |
| Spiders.js | 28 (35%) | 46 (58%) | 80 |
| ActorReactor.js | 22 (28%) | 42 (54%) | 78 |

**Table 1: Comparison of he amount of lines of code dedicated to message forwarding and message handling as a percentage of the total application (lower is better).**

## 4 EVALUATION

Evaluating the impact of reactive programming on applications is notoriously difficult, since its benefits are difficult to measure in code metrics. Nevertheless, we evaluate our approach using our implementation of the CoCode application in native JavaScript, Spiders.js (only actors), and our library ActorReactor.js (actors and reactors). While the Spiders.js and ActorReactor.js applications are equivalent, the native JavaScript implementation does not feature peer-to-peer communication between clients. This simplifies its code, since there is no need to manually exchange web sockets between clients (which is not possible). In practice this entails that there are fewer lines of code on the client because it need not keep a reference to the other clients, but this is somewhat compensated on the server which is now responsible for relaying public and private messages.

We measure the complexity of the application with two metrics: Message forwarding and message handling.

**Message forwarding** This includes all code that is required for communication, including code that simply receives and forwards messages. This includes manual message sends (for the native application), method invocations, broadcasts, and react-to statements. We also include surrounding code if the communication occurs in a for loop, e.g. when sending a message to all clients. Message forwarding can give an indication of the complexity of communication.

**Message handling** Here we consider all code related to handling messages. This can sometimes overlap with message forwarding, since handling a message may invoke new communication. In the native implementation we count the functions that serve as callbacks for the communication protocol, i.e. the callback that is invoked when a certain type of message is received. In the library implementations we count the methods defined in classes. Ideally the message handlers only concern themselves with application logic. More lines of code can thus give an indication of more boilerplate code that distracts from the main application logic.

Table 1 shows the lines of code dedicated to message forwarding, message handling, and the total lines of code of the application (GUI code is always omitted). With respect to code forwarding, a significant portion of the native implementation is dedicated to communication. This is a consequence of manual message sending, because every receiver needs to dispatch over the type of message in order to invoke the correct message handler. Message forwarding in Spiders.js and ActorReactor.js is expectedly lower than the native version since they abstract over message sends as method invocations. While there is a small difference between Spiders.js and ActorReactor.js, we can not draw any conclusions from it, since in our small application a direct method invocation is often directly replaced with a broadcast and react-to. Larger applications may show bigger benefits due to reduced code complexity.

With respect to message handling, all three implementations have little overhead. While we see no significant difference in lines of code between Spiders.js and ActorReactor.js, one of the key advantages of reactive programming is difficult to measure in lines of code, especially in small applications. That is, reactive programming has been shown to increase program comprehension compared to object-oriented programming [12]. Additionally, when comparing our implementation to Spiders.js we see that there are a minimal number of direct method invocations hard-wired between components. Loose coupling of components has benefits for code reuse, separation of concerns, maintainability, etc.

## 5 CONCLUSION

With the advent of Rich Internet Applications the internet is transitioning to a new era where web applications often exhibit the following characteristics: On the one hand, RIAs often integrate a plethora of different services that constantly publish new data relevant to the application. This requires the application to be written in an event-driven style. On the other hand, as more and more of the application logic is pushed to the client, these thick clients become more and more computationally intensive. To remain responsive to incoming GUI events, it might become necessary for these applications to execute more heavy weight computations in parallel to the main application logic. In this paper we motivate that current solutions for both concerns are not always easily integrated.

In this paper we presented a programming model based on two abstractions, actors and reactors. The key insight for these two abstractions is that actors are driven by the control flow of the program, and reactors are driven by data. In other words, the application logic of a RIA can be specified in our model as a number of concurrently running actors, while the reactive (data-driven) parts of the application can be specified in terms of reactors. Actors and reactors are naturally concurrent components that provide a uniform communication model based on publish/subscribe, where both actors and reactors may publish data, and subscribe to data by others. Furthermore, actors and reactors may be distributed over the client and server, allowing applications to react to events on the server.

We implement our approach on top of the Spiders.js actor library, and use it to implement a collaborative code editor (CoCode). We compared our implementation of CoCode with the same application written in native JavaScript and Spiders.js. We have shown that our approach reduces the number of lines of code related to communication with respect to the native JavaScript implementation, and remains similar to the Spiders.js implementation. Additionally, while not measurable in lines of code, our approach reduces the amount of hard-wired dependencies between components, which benefits code reuse, separation of concerns, and maintainability of the resulting application.

## ACKNOWLEDGMENTS

This work is partially funded by Flanders Innovation & Entrepreneurship (VLAIO) with the FLAMENCO project (FLAnders Mobile ENacted Citizen Observatories) under grant No.: 150044, and the Brussels Institute for Research and Innovation (Innoviris) with the Doctiris programme under grant No. 15-doct-07. The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

## REFERENCES

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (2013), 52:1–52:34.

[2] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422.

[3] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. 1999. Calling Hell From Heaven and Heaven From Hell. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, Didier Rémi and Peter Lee (Eds.). ACM, 114–125.

[4] Node.js Foundation. 2017. Node.js. (2017). http://web.archive.org/web/20170129014504/https://nodejs.org/en/ Accessed: 2017-01-30.

[5] David Garlan and David Notkin. 1991. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 1: Conference Contributions (Lecture Notes in Computer Science)*, Søren Prehn and W. J. Toetenel (Eds.), Vol. 551. Springer, 31–44.

[6] Facebook Inc. 2017. React: A JavaScript Library for Building User Interfaces. (2017). http://web.archive.org/web/20170117003017/https://facebook.github.io/react/ Accessed: 2017-01-16.

[7] Ingo Maier and Martin Odersky. 2012. *Deprecating the Observer Pattern with Scala.React.* Technical Report. École Polytechnique Fédérale de Lausanne, EPFL-REPORT-176887.

[8] Alessandro Margara and Guido Salvaneschi. 2014. We have a DREAM: distributed reactive programming with consistency guarantees. In *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, Umesh Bellur and Ravi Kothari (Eds.). ACM, 142–153.

[9] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. 2005. Concurrency Among Strangers. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Rocco De Nicola and Davide Sangiorgi (Eds.), Vol. 3705. Springer, 195–229.

[10] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2016. Many spiders make a better web: a unified web-based actor framework. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci (Eds.). ACM, 51–60.

[11] ReactiveX. 2017. An API for asynchronous programming with observable streams. (2017). http://web.archive.org/web/20170128174958/http://reactivex.io/ Accessed: 2017-01-31.

[12] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 564–575.