Many Spiders Make a Better Web

A Unified Web-Based Actor Framework

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

Christophe Scholliers
Universiteit Gent
281 S9, Krijgslaan
Gent, Belgium
christophe.scholliers@ugent.be

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

Abstract

JavaScript is the predominant language when it comes to developing applications for the web. Many of today's web-based systems are implemented solely in JavaScript due to its applicability to both client and server-side development. This use of JavaScript as a general purpose programming language has sparked a number of efforts to provide programmers with the means to write parallel web applications. For the most parts these efforts have based themselves on actor-based parallelism, given the similarities between actors and the JavaScript runtime model. We argue that these solutions are based on actor models which do not optimally fit web development. Concretely, existing solutions fail to provide programmers with an actor framework which embraces both parallelism and distribution. To this end we present Spiders.js, an actor framework providing both high-level parallelism and built-in distribution. In Spiders.js, programmers can easily specify the coarse-grained parallelism needs of modern web applications. Moreover, Spiders.js' built-in distribution features allow programmers to express client/server, server/server and client/client distribution simply by using actors. We show the performance characteristics of our approach by detailing Spiders.js' results for the Savina benchmark suite.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Concurrent Programming*]: Distributed Programming

Keywords Actor Framework, Web, Distributed Programming, Communicating Event Loops

AGERE'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4639-9/16/10...$15.00
http://dx.doi.org/10.1145/3001886.3001892

1. Introduction

Roughly 94% of all websites use JavaScript [1]. Although it was originally designed as a scripting language to be used in web browsers, JavaScript has since evolved into a general purpose language adopted in a range of contexts far beyond client-side web applications (e.g. server-side applications [14], mobile applications [19], ...).

Similarly, the genre of applications written in JavaScript evolved since its inception. Where the web used to be comprised of static HTML pages, it has now evolved into a platform for full-fledged distributed applications. Google's Gmail and WebOS (a complete operating system built entirely in JavaScript) are examples of such applications. To enable this transition from the web 1.0 to the web 2.0, web browsers have evolved from simple graphic displayers to interpreters which execute complete JavaScript programs.

In order to maintain the responsiveness of these in-browser applications, a lot of work has focused on optimising the execution of JavaScript code. Examples of these optimisations include just-in-time JavaScript compilers [6] and thread level speculation [10]. Besides language or runtime-level optimisations, JavaScript shows great promise to optimise application-level code through the use of parallelisation [5]. JavaScript programmers are only able to fulfil this promise through the use of two actor-based parallel constructs: *web workers* for client-side JavaScript and *child processes* for server-side technology (i.e. Node.js, which is the most prominent server-side implementation of JavaScript). However, these constructs severely limit the programmer in three ways:

Hierarchical Communication Message sends between actors (i.e. web workers or child processes) are natively supported only between parent and child. Upon spawning an actor the spawning actor obtains a reference to the newly spawned actor through which messages can be sent. Similarly the spawned actor is able to reference the spawning actor. However, such references cannot be exchanged between actors as this will result in a runtime exception. This forces programmers to either adapt their

application to this hierarchy or circumvent it (e.g. using web sockets or message channels).

Object Passing Only primitive data types (i.e. numbers, strings, ...) passed between two actors are automatically serialised. Other data types (e.g. functions, objects) must be serialised manually by the programmer which quickly leads to a number of error-prone situations. For example, manual serialisation of objects forces the programmer to take care of possible scoping issues (e.g. an object having a method which refers to variables defined in its lexical scope).

Distribution JavaScript is a language used majoritarily in a distributed context. However, the built-in parallelism constructs provided by JavaScript fail to embrace this setting in two ways. First, the APIs used for web workers and child processes differ. This complicates code reuse between the client and server tier. Second, message passing is only provided for local actors (i.e. a web worker cannot natively send messages to a child process).

In this paper we argue the need for a re-design of the parallelism constructs provided by JavaScript. To prove our point we detail the design and implementation of an actor framework called Spiders.js [1]. Concretely, Spiders.js solves the aforementioned issues with native JavaScript actors in the following three ways:

- Actor references are first class. This entails that all actors are able to exchange references between and send messages to each other. Moreover, Spiders.js offers a registry mechanism allowing actors to look-up references to one another.
- Programmers are freed from the burden of needing to manually serialise objects. Objects are either passed between two actors by reference (e.g. a function object) or by copy (e.g. a numeral value). In both cases the programmer is unaware of the underlying serialisation.
- Our actor framework exposes the same API and semantics regardless of the tier (i.e. client or server) in which it is used. Moreover, Spiders.js' underlying message passing system can handle both *vertical* as well as *horizontal* distribution. The former allows for traditional client/server interaction while the latter allows for server/server and client/client communication.

2. Problem Statement

Besides JavaScript's built-in parallelism constructs, a number of other actor-based solutions have been proposed as a way to parallelise web applications [13, 18]. Inherently, JavaScript's runtime (i.e. its event-loop) and the actor model are akin to each other. Actors are parallel entities operating under their own thread of control. In contrast to other parallel models (e.g. threads), actors are unable to share state. Instead, actors coordinate through the exchange of messages. A part of the issues with the way actors are currently conceived in JavaScript stems from the kind of model they employ. According to [4] one can discern four actor models:

Original actor model This model is characterised by three primitives : *send*, *create* and *become*. The first two primitives are used to send messages between actors and create actors respectively. Changes to the actor state are modelled through *become* statements which effectively alter the *behaviour* of the actor.
For the most part, this model has been implemented in functional or procedural paradigms [2, 9, 15]. This functional approach to actors is in sharp contrast with JavaScript's imperative and object-oriented programming style.

Processes The *processes* model (e.g. Erlang [17]) defines actors as single processes which execute an actor's behaviour from start to finish. The difference with the previous model being that the process drives the execution of the actor, rather than the messages received by the actor. In order to deal with messages the model offers primitives to synchronously await messages. After receival of the awaited message the process continues the execution of the actor's behaviour. Once the process has run through the code the actor dies and is no longer able to receive messages.
The synchronous message handling of the process model conflicts with JavaScript's inherent asynchrony. JavaScript disallows the use of blocking primitives in order to make sure that the underlying event loop remains responsive.

Active Objects The *active objects* model (e.g. Salsa [16]) can be seen as an object-oriented version of actors. Each actor represents its behaviour as an object. The interface of that object will become the actor's interface, while changes to the actor's state are done by mutating the active object's instance variables. Objects which are passed between actors (e.g. an object is contained within a message to a certain actor) are automatically serialised.
In the case of JavaScript not all objects can easily be copied between actors. For example, function objects might retain pointers to their original lexical scope. One would either need to impose restrictions on the kind of function objects which can be passed between actors or copy the function's lexical scope as well.

Communicating Event-loops The CEL model (e.g. E [12]) differs from the previous actor models in three ways. First, actors do not provide a single interface to other actors (e.g. a single active object). Rather, actors encompass a multitude of objects to which other actors can obtain references. Second, all objects are no longer passed between actors by copy. Rather, actors obtain *far references*

[1] The implementation is available at `https://github.com/myter/Spiders.js`

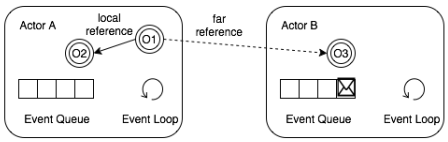

Figure 1. Actors as communicating event loops [4]

to objects owned by other actors. Exceptions are made for primitive objects (e.g. numbers, strings) which are copied between actors. Third, all previous models were designed and implemented with fine-grained parallelism in mind. The CEL model, on the other hand, is more coarse-grained in its parallelism approach.

As far as we know, all implementations of actors in JavaScript have thus far focussed on the active objects model (e.g. web workers and child processes, generic workers [18]) or a hybrid between the original model and the active objects model (e.g. Akka.js [13], Syndicate.js [7]). However, the CEL model better fits JavaScript for three reasons. First, parallelism in JavaScript is inherently coarse-grained. Most browsers rigorously limit the amount of web workers one can spawn in a single page in order to keep all open pages responsive. Unlike other models, the CEL model employs a coarse-grained parallelism scheme. Second, some objects in JavaScript cannot be copied (e.g. sockets) while copying others cannot be done automatically (e.g. function objects). The CEL model differentiates between objects which must be passed between actors by reference and those that can be passed by copy. Third, JavaScript is mostly employed in a distributed context relying on asynchronous event handling. The CEL model tightly fits this context by solely providing non-blocking (i.e. asynchronous) primitives. Moreover, the way CEL actors handle events closely resembles JavaScript's internal event-loop model.

3. Communicating Event Loops

We first provide a more in-depth explanation of the CEL model before detailing our approach. Readers already familiar with the workings of this model may skip this section.

Figure 1 gives an overview of the CEL model. Each actors is an independent entity supported by its own thread of control and contains a heap of objects (i.e. actors are conceived as vats). Each actor encompasses a number of objects (the circles in the figure). Moreover, each actor contains an event loop and a message queue. Each message is the result of an invocation on an object by another actor. The event loop continuously picks the first message from the queue and performs the invocation the message contains.

The CEL model specifies two kinds of references: *local* references and *far references*. As Figure 1 shows, the reference that *O1* holds to *O2* is a local one since both objects are owned by *Actor A*. This entails that all invocations of methods by *O1* on *O2* happen in a standard sequential fashion. On the other hand, the reference which *O1* holds to *O3* is a far reference since *O3* resides in *Actor B*. When *O1* invokes a method or accesses a field of *O3* this invocation is translated in a message sent asynchronously to *Actor B* (i.e. the message is queued in *Actor B's* message queue). Eventually *Actor B's* event loop will dequeue the message and the invocation will be executed.

```
uiActor({
 highlighter: null,
 init: function(){
  var fut = this.getRef("highlighter")
  this.onResolve(fut,function(ref){
    this.highlighter = ref
  })
 },
 newCode: function(code){
  var fut = highlighter.highlight(code)
  this.onResolve(fut,function(highlighted){
   gui.updateCode(highlighted)
  })
 }
})

actor({
 imports: ['./highlightLib.js'],
 highlight: function(code){
  return this.highlightLib.highlight(code)
 }
},"highlighter")
```

Listing 1. Spawning client-side actors in Spiders.js

4. Spiders.js by Example

In order to showcase the applicability of our approach we outline an example application which stereotypes the kind of applications that benefit from Spiders.js. The application, called *CoCode*, allows programmers to collaboratively code in their favourite language. To do so, each programmer logs in to the application after which she/he can start coding in a dedicated part of the webpage. Each coder has a consistent view of the piece of code which is collaboratively edited by all logged-in coders. Moreover, the code is syntax highlighted on the fly.

To keep our application efficient each client performs the syntax highlighting on its own view of the code. This allows the underlying synchronisation of code to work with pure text rather than highlighted code, which is substantially larger in size and would slow down communication. Moreover, each client spawns an actor dedicated to performing the syntax highlighting. This is to avoid blocking the UI thread and render the application unresponsive. Besides this core functionality, CoCode also provides coders with ways to discuss in both a public chat room as well as through private messages.

4.1 Basic Spiders

Listing 1 shows how the standalone functionality of each CoCode client (i.e. highlighting code in the webpage as a user types) is implemented. Each client consists of two actors: an actor dedicated to updating the user interface with

```
actor({
 coders: {},
 register: function(name,ref){
  for(var i in coders){
   coders[i].newCoder(name,ref)
  }
  this.coders[name] = ref
 },
 codeSync: function(newCode){
  for(var i in coders){
   coders[i].codeSync(newCode)
  }
 },
 publicMessage: function(message){
  for(var i in coders){
   coders[i].newPublicMessage(message)
  }
 },
})
```

Listing 2. Spawning server-side actors in Spiders.js

highlighted code and an actor which will perform the actual highlighting. Ensuring that only a single actor is able to alter the user interface allows us to avoid data races and glitches in the interface. Each actor is created by providing it with a behaviour object which serves as the entry point for all other actors.

The UI actor, spawned by invoking the *uiActor* function provided by Spiders.js, has an *init* method which will be invoked upon creation. This function will acquire a reference to the highlight actor through Spider.js' name-based actor registry. At each key stroke the *newCode* function of the UI actor will be invoked (the HTML code responsible for this is omitted for the sake of brevity). Subsequently, the UI actor invokes the *highlight* method on the highlighting actor's behaviour object (line 10 in the code). All method invocations on remote objects (i.e. objects owned by other actors) are translated to asynchronous message sends and return a future. Actors are able to register callbacks on futures, through the *onResolve* function, which will be called with the return value of the asynchronously invoked method. In our case the UI actor registers a callback which will update the user interface with the highlighted code. This update happens by invoking *updateCode* on the *gui* object which represents the webpage.

4.2 Distributed Spiders

So far our CoCode application is standalone, the UI actor only responds to new code being produced by the local user. The first part in making CoCode a true web application is setting up a server. The code to do so is given in Listing 2.

Our server consists of a single actor which will handle the synchronisation of the clients' code and public chat messages. The server's behaviour object implements our application's functionality with three methods which will all be invoked by client-side actors. *Register* allows client-side actors to register themselves to the highlighting service. This method stores the reference to the client's behaviour object (i.e. *ref*) in a hashmap and notifies all pre-registered coders

```
uiActor({
 highlighter: null,
 serverRef: null,
 coCoders: {},
 init: function(){
  this.highlighter = this.getRef("highlighter")
  var fut = this.getRemoteRef(serverAddr,serverPort)
  this.onResolve(fut,function(serverRef){
   this.serverRef = serverRef
   serverRef.register(gui.name,this)
  })
 },
 newCoder: function(name,ref){
  this.coCoders[name] = ref
 },
 newPublicMessage: function(msg){
  gui.newPublic(msg.from,msg.text,msg.date)
 },
 newPrivateMessage: function(msg){
  gui.newPrivate(msg.from,msg.text,msg.date)
 },
 newCode: function(code){
  var fut = highlighter.highlight(code)
  this.onResolve(fut,function(highlighted){
   gui.updateCode(highlighted)
   this.serverRef.codeSync(highlighted)
  })
 },
 codeSync: function(code){
  var fut = highlighter.highlight(code)
  this.onResolve(fut,function(highlighted){
   gui.updateCode(highlighted)
  })
 },
 makeMessage: function(text){
  return this.isolate({
   from: gui.name,
   date: gui.now(),
   text: msgString
  )}
 },
 sendPublicMessage: function(msgString){
  var msg = this.makeMessage(msgString)
  this.serverRef.publicMessage(gui.name,msg)
 },
 sendPrivateMessage: function(to,msgString){
  var msg = this.makeMessage(msgString)
  this.coCoders[to].newPrivateMessage(gui.name,msg)
 },
})
```

Listing 3. Making CoCode clients distributed

that a new peer has joined the session. *CodeSync* is invoked by a client as soon as its user has entered new code through the user interface. Subsequently all other coders are notified of this event in order to highlight this new code. Lastly, *publicMessage* broadcasts a message from a particular user to all coders.

The addition of a server requires us to update the implementation of our client-side CoCode actors. Listing 3 shows the additions to the original client-side code of Listing 1 that are needed in order to make CoCode a full-fledged web-application. The code for the highlighting actor remains unchanged and is therefore omitted.

Each client first acquires a reference to the server actor (see line 7) through the *getRemoteRef* primitive. As is the case for *getRef* this primitive returns a future which will be resolved with a reference to the requested actor. In contrary to *getRef*, *getRemoteRef* takes the address and port of the

Table 1. Spiders.js API

Function	Arguments	Return
actor	object, (string)	far ref
uiActor	object, (number), (string)	far ref
getRef	string	future
getRemoteRef	number, number	future
.	far ref, selector, arguments	future
onResolve	future, function	null
onRuin	future, function	null
isolate	object	object

server actor as arguments. This reference to the server actor has two purposes. First, each client needs to register itself as a coder in order to get code updates from other clients. This is done by invoking the *register* method on the obtained reference (see line 10). Second, coders are able to send messages to each other. Through a dedicated section in the interface a user can either publicly broadcast a message (i.e. the UI invokes the *sendPublicMessage* method) or send a private message to a particular coder (i.e. through the *sendPrivateMessage* method). In both cases an *isolated* object is created which contains the name of the sender , the date and the actual text of the message. In contrast to other objects, isolated objects are sent by copy rather than by reference. As such, when a client receives a message (either through the *publicMessage* or *newPrivateMessage* methods), the message's data can directly be read from the copied object. If the message were to be sent by reference one would obtain a future for each field access which would be impractical for this use case.

5. Spider.js Runtime

This section exhaustively describes the functionality provided by Spiders.js. For each provided function we detail its semantics as well as how it is implemented using native JavaScript constructs. We divide this functionality into three categories: actors, referencing and message passing. An overview of Spiders.js' API is given by Table 1 which provides the arguments for each of Spiders.js' functions (arguments between parentheses are optional).

5.1 Actors

As mentioned in Section 2, actors in Spiders.js adhere to the CEL model. These actors form the cornerstone of our framework. Although we cannot prohibit programmers from using web workers or child processes in combination with our framework, it is Spider.js' goal to provide a complete solution to parallelism in JavaScript.

Application Actors can be spawned through one of two primitives: *uiActor* or *actor*. Both these functions take a behavioural object as argument and return a reference to the spawned actor. An optional naming argument can be provided in order to register the actor in Spider.js' registry system. If the naming argument is not provided the actor will not be registered in the registry system.
The server-side versions of these primitives accept an additional optional numeral argument which indicates the port on which the actor will listen for incoming messages (which is 8080 by default). If the behavioural object of the actor specifies an *init* method it will be invoked at creation time.
Part of the strength of parallelism through actors is that programmers need not deal with data races. In Spiders.js this is achieved by denying an actor's behavioural object access to its lexical scope. Concretely, the behavioural object can only access its own fields and methods. All other accesses result in a runtime exception. For an actor to make use of external libraries it can specify the path to these libraries using the *imports:* field. Moreover, Spiders.js enforces single actor access to the user interface on the client-side. Only actors spawned with the *uiActor* primitive are able to access the DOM and only one of such actors can be spawned per page.

Implementation The implementation of a Spiders.js actor depends on whether it was spawned client or server-side. Client-side actors are built atop web workers. We differentiate between the UI actor which runs on the client's main thread (and is therefore not an actual web worker) and regular actors, each of which are supported by a single web worker. Server-side actors are implemented as child processes, which entails that each server-side actor is a full-fledged Node.js instance.

5.2 Referencing

Actors in Spiders.js are conceived as communicating event loops and therefore come with a built-in referencing mechanism. Concretely, objects within the same actor reference each other locally and are able to access each other's fields and methods synchronously. Objects residing in different actors can obtain far references to each other and access each other's fields and methods through asynchronous messaging. However, the dynamic nature of JavaScript and the fact that Spiders.js is implemented as a library entails that one cannot lexically determine whether a reference is local or far.

Application Two objects within the same actor can reference each other locally using JavaScript's referencing mechanism (i.e. using *this*). An actor can obtain a far reference to an object owned by another actor implicitly or explicitly. Obtaining a far reference to an actor's behaviour object can be done explicitly with *getRef* or *getRemoteRef*. The former is used to obtain a reference to the behaviour object of an actor residing on the same physical machine. The function takes the actor's name as parameter and returns a future which will be resolved with the reference to the actor's behaviour object. The latter is used to obtain a reference to the behaviour object of an actor residing on a different machine. This function takes an ip-address and port number and returns a future which will eventually be resolved with the far

reference. This functionality can either be employed by a client-side actor to obtain a reference to a server-side actor or between two server-side actors. Other configurations (e.g. a client-side actor requiring a remote reference to another client-side actor) are unsupported since web clients lack a static ip-address.
Since references in Spiders.js are first-class entities one can also obtain far references implicitly. For example, in CoCode (see Section 4) the server actor allows all clients to obtain far references to each other. When a new coder connects to the server actor it provides a far reference to its own behaviour object. The server actor then forwards this reference to all other clients. In general an actor can implicitly obtain a far reference in two ways. First, if the future resulting from an invocation on a far reference is resolved by a far reference. Second, if a method of one of its objects is invoked remotely (i.e. by an object residing in a different actor) with a far reference as argument.
By default objects are passed between actors as far references. Exceptions are made for primitive objects (i.e. numerals,strings, booleans, ...) and isolates. A programmer can explicitly create such an isolate through the *isolate* construct which takes a regular object and returns an object which will be passed by copy rather than by far reference. Given that isolates are passed by copy it is the programmer's responsibility to make sure that the isolated object does not rely on its lexical scope (e.g. a method accessing a variable defined outside the object).

Implementation Far references are essentially proxies to objects owned by an actor other than the one owning the reference. This reference should provide the same methods and fields as the proxied object. Moreover, calling a method or accessing a field should be translated into an asynchronous message to the actor owning the original object. To achieve this we need to be able to infer which methods and fields the proxied object offers. Furthermore, we need to be able to intercept invocations and accesses to this proxy object
In Spiders.js far references are implemented using JavaScript's proxies. This reflective construct allows us to define *traps* which intercept method invocations and field accesses. Given that a far reference holds an exhaustive list of the referenced object's properties, the wrapping proxy is able to mimic the object's behaviour. If an actor accesses a property known to the referenced object, the proxy will forward the call to the actor owning the referenced object through an asynchronous message and return a future. If the referenced object lacks the accessed property an exception is thrown to notify the calling actor accordingly.

5.3 Message Passing

Although Spiders.js provides the same API towards programmers regardless of whether they are implementing server or client-side actors, the implementation of these actors differs. Similarly, the way in which messages are sent depends on what type of actors are interacting. However, from a programmer's point of view these complexities are hidden in favour of a simple and unified actor model.

Application Programmers never need to deal directly with message sends in Spiders.js . It suffices to have a far reference to an object and access one if its fields or methods, Spiders.js will perform the actual message send in the background. Such an access or invocation is done using JavaScript's dot operator, after which a future is returned. Actors can register a listener which will be called with the return value of the invoked method or accessed field using *onResolve*. Moreover, an exception handler can be registered to a future using *onRuin* in order to catch any exception thrown by the remote object.

Implementation From an implementation perspective we discern two types of far references: *internal* and *external* references. The former type is used for objects owned by actors residing in the same JS environment while the latter is used for objects owned by actors from different environments. We define a JavaScript environment as either a web-page in a browser (for client-side actors) or a Node.js instance (for server-side actors). Given that each server-side actor is implemented as its own Node.js instance, each server-side actor is its own JS environment.
On one hand, internal references (i.e. actors running in the same web-page) communicate through the messaging system provided by web workers and message channels. All actors possess internal references to each other, each of these references contains a message channel used to send messages to the corresponding actor. On the other hand, external references communicate through web sockets. The use of these sockets depends on the kind (i.e. client or server) of the actors communicating. We differentiate between the actor owning a particular referenced object (i.e. the issuing actor) and the actor accessing a field or invoking a method on the far reference (i.e. the receiving actor).

Server/Server In this case both the receiving actor and the issuing actor are server-side actors. Since each server actor runs a socket server at startup, the issued far reference keeps track of the IP address as well as the port on which the owning actor listens for messages. When the receiving actor accesses a property of said reference the actor opens a connection to the issuing actor's socket and forwards the access.

Server/Client In this case the receiving actor is a server-side actor while the issuing actor is a client-side actor. As a consequent the client must have requested a remote far reference to the server-side actor and has therefore already opened a connection to the actor's web socket. This connection is used by the server-side actor to forward all property accesses to the client-side owning actor. The same hold for the reverse case where the client-side

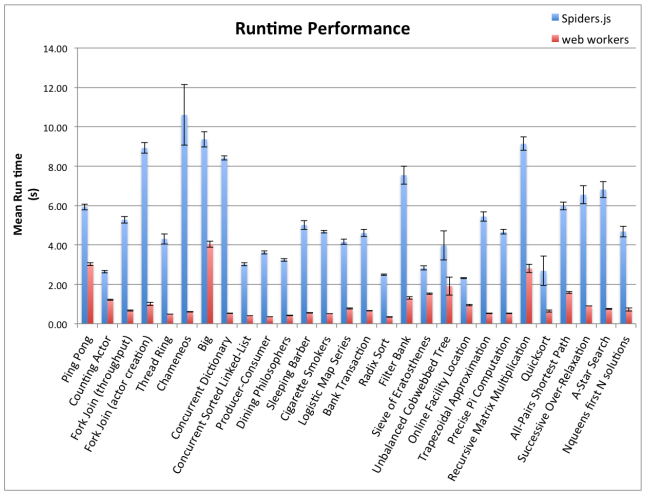

Figure 2. Comparing Spiders.js and web workers in the Savina Benchmark Suite. Error bars indicate the 95% confidence interval

actor received a reference to an object owned by a server-side actor.

Client/Client Both the receiving actor as well as the issuing actors are client-side actors. This case can only take place as a result of the receiving actor initially getting the reference from a server-side actor (since client-side actors can only use *getRemoteRef* to acquire a server-side far reference). When a property of the far reference is accessed, the receiving client-side actor will request this intermediate server-side actor to route the access to the issuing client-side actor. This layer of indirection is needed given JavaScript's lack of client-to-client communication primitives.

6. Evaluation

We implemented the Savina [8] benchmark suite twice: once in the client-side version of Spiders.js and once using JavaScript's native web workers. Since Spiders.js uses web workers for its client-side implementation we performed these experiments to assess the runtime performance impact of the added functionality atop web workers. Moreover, we compared the coding complexity associated with each approach.

All benchmarks were conducted using Benchmark.js [2] in Mozilla Firefox (version 47.0) on a Macbook Pro with a 2,8 GHz intel core i7 processor, 16GB 1600 MHz DDR3 of RAM memory running Mac OSX Yosemite (version 10.10.5)

6.1 Runtime Performance

The results for all benchmark applications in the suite are given in Figure 2. For each application we provide the mean

[2] `https://benchmarkjs.com/`

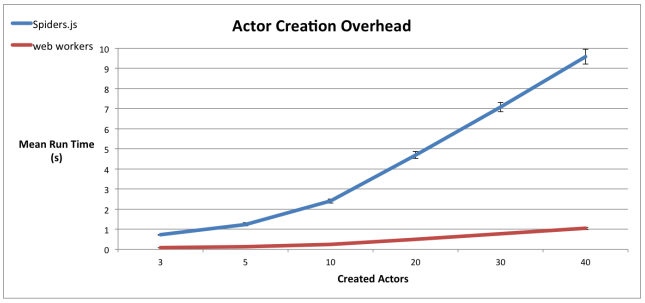

Figure 3. Actor creation overhead as measured in the *fork join* application. Error bars indicate the 95% confidence interval.

time to completion in seconds for both the Spiders.js implementation (indicated by the blue bars) and the web workers implementation (indicated by the red bars). One clearly distinguishes that Spiders.js incurs a significant overhead compared to the web workers approach. In the best case, for the *ping pong* application, Spiders.js is roughly twice as slow with a mean time to completion of six seconds compared to three seconds for the web workers implementation. In the worst case, for the *chameneos* application, Spiders.js is roughly seventeen times slower with a mean to completion of ten seconds compared to a completion time of under a second for the web workers implementation.

This overhead stems from two parts of the Spiders.js implementation: actor creation and message passing. Two applications clearly showcase this overhead. First, the *fork join (actor creation)* application which aims to measure actor creation and destruction overhead. Second, the *big* application which is aimed at measuring messaging overhead. In order to further investigate this source of overhead for Spiders.js we performed additional experiments focused on these two applications.

6.1.1 Actor Creation Overhead

Figure 3 provides the results for the experiments aimed at measuring actor creation overhead (error bars for the web worker results are hardly visible due to the low error rate). The overhead which web workers incur is clearly orders of magnitudes smaller than the one incurred by Spiders.js. Moreover, as the amounts of spawned actors increases, this overhead considerable slows down the Spiders.js implementation.

The reason for this overhead can be found in the way both approaches provide actors with their behaviour objects. In order to spawn a web worker one must provide the path to a JavaScript source file containing the worker's behaviour. In Spiders.js the programmer provides this behaviour as an object, after which the object is serialised and sent to a blank web worker by Spiders.js' implementation. Upon reception of this serialised object the web worker must evaluate the object and instantiate it as its behaviour. Both the serialisation

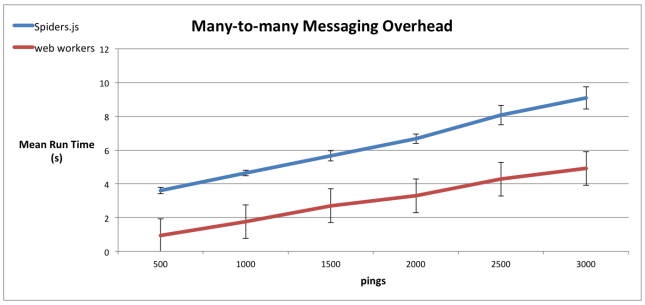

Figure 4. Message passing overhead as measured in the *big* application. Error bars indicate the 95% confidence interval.

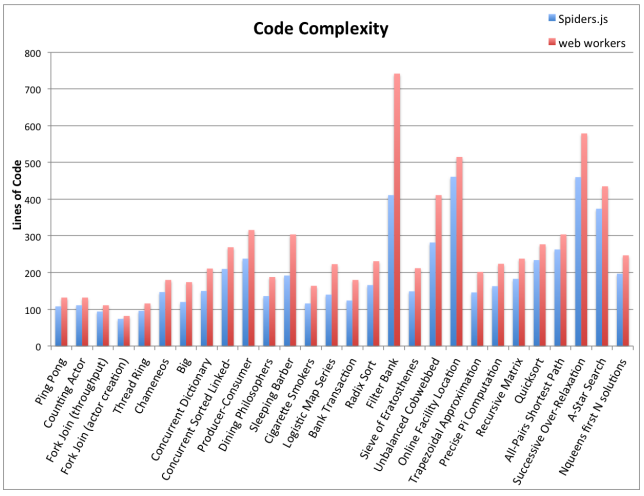

Figure 5. Code complexity associated with web workers and Spiders.js in the implementation of Savina.

of this behaviour object and its instantiation have a clear and significant impact on the overhead associated with spawning actors.

6.1.2 Message Passing Overhead

We used the *big* application to conduct our experiments involving message passing overhead. This application is an extension of the traditional *ping pong* scenario in which a number of actors continuously send *ping* messages to each other. Figure 4 shows the results of these experiments. One clearly denotes that the message passing overhead is a constant factor greater in Spiders.js than it is in the web workers implementation.

This overhead stems from the fact that messages in Spiders.js are reified as objects, and therefore need to be manually serialised by the implementation. In the web workers implementation the ping message is a simple string which can be sent between workers without this manual serialisation. As a result Spiders.js messages incur a constant overhead over web workers sending primitive values.

6.2 Coding Complexity

As stated in [8], the Savina benchmark suite allows to compare actor frameworks with regards to both runtime performance as well as the complexity of the code with which programmers have to deal with using the framework. In order to showcase that Spiders.js simplifies the implementation of parallel applications for the web we compared the web workers implementation of the suite with the implementation in Spiders.js. Figure 5 showcases the lines of code needed to implement each application comprised by Savina. One clearly denotes that the applications written in Spiders.js are significantly smaller (roughly 25% on average).

This reduction in code complexity on Spiders.js' behalf mainly stems from two of its features. First, unlike web workers, Spiders.js actors do not need to explicitly define a message handling function. Rather, the actor's objects implicitly act as its message handler. Second, communication between two web workers requires a message channel to be created after which each of the participants needs to obtain a port of said channel. In contrast, Spiders.js actors solely need to obtain a far reference to objects owned by other actors to communicate.

7. Related Work

This work is strongly motivated by the limitations of built-in parallelism features of JavaScript. Although we are not the first to propose an actor framework for web applications we are the first to fully implement and explore the CEL model in this setting. What follows is a discussion of the more prominent actor-based solutions for JavaScript.

web workers Since HTML5, client-side JavaScript developers can employ *web workers* to execute code in parallel. At its core web workers are limited versions of actors: Given a URL to a piece of JavaScript code, the main thread is able to spawn web workers which will execute the code in their own thread of control. Moreover, web workers run in a completely isolated environment which entails that they do not have access to the scope in which they are created. This scope isolation also includes graphical elements such as the DOM, since these would also be race condition sensitive. This ensures that race conditions between workers are avoided. However, web workers limit programmers in a number of ways which we discuss in detail in Section 1.

Child Processes Server-side JavaScript (i.e. Node.js) offers *child processes* which can be used to execute any system command . They also provide a built-in wrapper (*fork*) which spawns a new node.js instance and returns an object used to send messages to the spawned instance. However, child processes limit programmers in the same way as web workers do.

q-connection The integration of the CEL model has already been discussed by previous work [11]. So far, the most

notable step towards this integration comes in the form of the Q-connection [3] library. As is the case for Spiders.js, q-connection differentiates between local and far references for objects. Moreover, far references can be exchanged between web workers. However, q-connection lacks the vat-semantics of Spiders.js: a web worker must explicitly export an object before another worker can acquire a far reference to it. Furthermore, q-connection does not solve the problems related to hierarchical communication identified in Section 1.

Akka.js Akka.js [13] is an actor framework that allows one to deploy Akka actors in any JavaScript environment. To do so it employs Scala.js to compile the Scala/Akka code to JavaScript.
Akka.js' main goals closely resemble ours. First, it strives for in-browser parallelism by mapping actors onto web workers. Second, it allows for different actor runtimes (i.e. server and client runtimes) to seamlessly communicate.
However, Spiders.js differs from Akka.js in two major ways. First, with Spiders.js we strive to provide JavaScript developers the means to easily write parallel applications. On the other hand, Akka.js aims to provide Akka/Scala programmers the means to easily deploy their application to JavaScript runtimes. Second, as the name suggests Akka.js is built atop the Akka actor-model which is a hybrid between the original and the active objects model. As we argued in Section 2, the CEL model better matches both the distributed needs for a web-based actor framework and the coarse-grained parallelism offered by web workers.

Generic Workers Generic workers [18] strive to unify the way in which communication happens between parallel entities (i.e. web workers) and distributed entities (i.e. client/server) in JavaScript. To do so, it introduces the notion of a *generic* worker which can run both on a client as well as a server. Furthermore, generic workers provide the same communication API regardless of the tier in which the communication partner resides.
Although we share the vision that a unified parallelism framework is needed for web applications, Spiders.js explicitly steps away from the traditional web worker interface in favour of a more expressive API through CEL actors.

Syndicate Syndicate [7] is a novel actor language tailored towards reactive programs. It extends upon functional actors with a number of reactive and event-driven features. Furthermore, it provides a JavaScript implementation of its model which provides the same features atop active objects-like actors.
Syndicate features a very expressive API which suits JavaScript applications well due to its event-driven and interactive nature. However, it reuses the underlying event-loop in order to provide concurrency and therefore lacks the parallel capabilities sought after in Spiders.js.

Connect.js Connect.js [3] is a JavaScript/Titanium [4] library which allows the development of cross-platform mobile applications. Spiders.js resembles Connect.js in two ways: both are heavily influenced by AmbientTalk and both are JavaScript based. However, both the aim as well as the implementation of Connect.js differs widely from Spiders.js. First, Connect.js operates in the context of mobile applications where peers are homogeneous (i.e. there is no client/server separation). Second and most importantly, actors in Connect.js do not operate under their own thread of control and therefore do not provide real parallelism.

8. Conclusion

As the web continues its evolution from a thin to a thick client model, the need for efficient web application increases. As is currently the case JavaScript constitutes the programming language used by most, if not all, web applications. JavaScript provides two constructs allowing programmers to write parallel web applications: *web workers* and *child processes*. However these two constructs suffer from three major deficiencies. First, sending messages between actors can only be done between the spawning actor and the spawned actor. Second, objects sent between actors are not always automatically serialised burdening the programmers with manual serialisation and deserialisation. Third, even though JavaScript programs are traditionally deployed in a distributed context the actors which it provides are unable to communicate over the client/server boundary.
A number of other incarnations of the actor model [7, 13, 18] have been proposed as more expressive means for parallelism in JavaScript. These frameworks have mostly been mirrored on two strains of the actor model: active objects (e.g. generic workers [18]) or variants of the original model [13]). We argue that the CEL model provides a better fit for actors in JavaScript. As is the case for JavaScript's native actors, CEL actors are tailored towards coarse-grained parallelism. This contrasts with other actor models which favour a finer-grained parallel approach. Moreover, the CEL model was designed to asynchronously handle events between possibly distributed actors. This design choice closely maps onto JavaScript's philosophy, which prohibits programmers from blocking the language's internal event-loop.
In this paper we unveil Spiders.js, an incarnation of the CEL model for the web. Spiders.js differentiates itself from other JavaScript actor frameworks in three ways. First, references to actors are first class and enable many-to-many communication between actors. Second, Spiders.js specifies two clear ways for objects to be passed between actors. Isolated objects are passed by copy without programmers needing to manually intervene in the serialisation. All other object are passed by reference for which Spiders.js ensures asynchronous messaging. Finally, actors in Spiders.js provide built-in distribution. Programmers are freed from the burden

[3] `https://github.com/kriskowal/q-connection`

[4] `http://www.appcelerator.com`

of manually managing distribution of actors between client and server. In Spiders.js all actors are able to communicate with each other (i.e. client/server, server/server and client/-client) using a single set of abstractions.
We compare the results of two implementations of the Savina benchmark suite: a first implementation in Spiders.js and a second implementation using JavaScript web workers. This comparison clearly indicates that further research needs to be conducted in order to improve the efficiency of Spiders.js' message passing and actor creation functionality. However, comparing the complexity associated with programming the suite in both frameworks clearly showcases that using Spiders.js makes programming parallel web applications significantly easier.

Acknowledgments

This work has been supported by Innoviris (the Brussels Institute for Research and Innovation) through the Doctiris program (grant number 15-doct-07).

References

[1] Usage of javascript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all. Accessed: 2016-06-3.

[2] G. Agha and P. Thati. *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*, pages 26–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-39993-3.

[3] E. G. Boix, C. Scholliers, N. Larrea, and W. De Meuter. Connect. js. Technical report, Vrije Universiteit Brussel, 2015. URL `http://soft.vub.ac.be/AGERE15/papers/AGERE_2015_paper_20.pdf`.

[4] J. De Koster. *Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems*. PhD thesis, Vrije Universiteit Brussel, 2015.

[5] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-9297-8.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[7] T. Garnock-Jones and M. Felleisen. *Coordinated Concurrent Programming in Syndicate*, pages 310–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-49498-1.

[8] S. Imam and V. Sarkar. Savina–an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control,AGERE!*, 2014.

[9] H. Lieberman. Thinking about lots of things at once without getting confused: Parallelism in act i. Technical report, Massachusetts Institute of Technology, 1981.

[10] J. K. Martinsen, H. Grahn, and A. Isberg. An argument for thread-level speculation and just-in-time compilation in the google's v8 javascript engine. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 25:1–25:2, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2870-8.

[11] M. S. Miller and T. Van Cutsem. Communicating event loops, an exploration in javascript. http://soft.vub.ac.be/ tvcutsem/talks/presentations/WGLD_CommEventLoops.pdf, 2011.

[12] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30007-4, 978-3-540-30007-6.

[13] G. Stivan, A. Peruffo, and P. Haller. Akka.js: Towards a portable actor runtime environment. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 57–64, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3901-8.

[14] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80, 2010.

[15] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent systems architecture. *SIGPLAN Not.*, 24(4):91–93, Sept. 1988. ISSN 0362-1340.

[16] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, Dec. 2001. ISSN 0362-1340.

[17] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

[18] A. Welc, R. L. Hudson, T. Shpeisman, and A.-R. Adl-Tabatabai. Generic workers: Towards unified distributed and parallel javascript programming model. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 1:1–1:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0544-0.

[19] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1851-8.