JavaScript Framework for Actor-Based Programming

Andrew Buhagiar

Supervisor: Prof. Kevin Vella

Faculty of ICT

University of Malta

April 5, 2022

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

JavaScript Framework for Actor-Based Programming

submitted is my work, except where acknowledged and referenced.

Andrew Buhagiar

May 20, 2022

Acknowledgements

your acknowledgements

Abstract

an abstract

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

JavaScript [1] is widely used for client applications on the browser and benefits from a growing popularity of server-side applications using environments such as Node.js. It is a single threaded language which lacks an intuitive way to program in a parallel and distributed fashion. This dissertation presents a prototype that implements a JavaScript framework which allows developers to build actor-based systems. Developers can use actors as concurrent units of computation which can be deployed either locally or remotely on multiple node runtimes and browsers. This will allow developers to intuitively distribute work amongst multiple processors and devices to fully utilise the hardware usually available in servers and modern computers.

Actors [2][3] communicate with each other through messages which are stored in the receiving actor's message queue. A message is processed by executing its defined actor behaviour. The Actor Model is a good fit for JavaScript's event loop [4][5] since they are both event driven. The Actor Model has already achieved success in the telecommunications industry and is more recently used for implementing distributed systems in languages such as Erlang and Scala/Akka [6].

1.2 Objectives

This dissertation will explore the suitability of the Actor Model when used to reason about distributed and concurrent systems in JavaScript through the use of one or multiple prototypes. The framework's suitability is based on its performance, scalablity, and intuitiveness to use. The objectives of the artifact are as follows.

1. Allow developers to easily define and spawn actors through the framework API on Node.js or a browser. Actors should be able to send messages to other actors as well as spawn more actors. Full interoperability should be offered when using the framework across the two environments.

2. Allow for spawning and interacting with actors on different Node.js processes through different network links. Node.js cluster[7] can be used for IPC between node processes, while WebWorkers[8] can be used for communication between the primary thread and its spawned workers. When such communicaiton links are not available, the network stack is involved by using WebSockets for flexible communication to link remote processes and devices.

3. Provide location transparency when dealing with actor references. Interacting with a local actor should be the same as interacting with a remote actor when using the API

4. Benchmark the performance and scalability of the developed prototype.

2. Background

2.1 Concurrency and Parallelism on JavaScript

Concurrency allows one to switch the order of the execution of tasks without yielding an incorrect output. Since the order in which a program is executed would not an issue, one can run these code segments in parallel, reducing the time taken to solve a particular problem. JavaScript and Node.js rely on a single threaded non-blocking event loop [4][5] which do not support parallel execution of such tasks, raising an issue when high performance [9] computing is required. If JavaScript had to block when input or output was required, it would stop a page from being responsive. Instead, JavaScript handles I/O using events and callbacks which were posted on the event queue. The runtime will eventually process all the messages in a FIFO order, where each event has a corresponding callback or event listener function to execute. Each message is processed to completion without pre-emption by the processing of a different message, allowing for predictable concurrency of code. Web Workers[8] and child processes[7] can be used to bring parallel computation to JavaScript [10][11] and uses a similar philosophy to that of the actor model. Both involve independently running tasks which communicate through messaging in order to allow for collaboration in solving a particular problem.

JavaScript is an implementation of the ECMAScript design. The ECMA-262 (ECMAScript 2021) language has multiple published editions, each of which serving

as the blueprint for JavaScript's next stable release. ECMAScript 8 provides the SharedArrayBuffer [12][13] constructor which allows for concurrent access for a set of bytes. This allows for memory to be shared across agents in different cluster processes or webworkers. The process which created the SharedArrayBuffer need only pass the object to the workers to allow them to access and manipulate the same data block. SharedArrayBuffers allow for different workers to have access to the same memory which promotes collaboration when working on the same data points. However, parallel code accessing the same data may lead to data races and additional care must be taken by the developer.

2.2 The History of the Actor Model

The Actor Model was first introduced by Carl Hewitt [2][3] in 1973 for research in artificial intelligence. It defines actors as computational agents which execute a uniform behaviour when sent a message. Hewitt argued that the actor metaphor can be applied to processes and daemons amongst other things. Two years later, Carl Hewitt helped in writing a draft of PLASMA, the first Actor Model language. In this language, actors communicate with each other using messages. The receiver processes the received message using its pre-defined computation. Based on the message's contents, the actor may choose one from different behaviours which may involve sending messages to other actors.

The Actor Model was simplified by Gul Agha [14] in 1986. While Carl Hewitt set the foundation of the potential applications of the actor metaphor, Gul Agha focused on how actors can be used to create expressive, simple, and intuitive programming languages. Gul Agha believed that it is more cost effective to use many smaller processors rather than rely on an individual powerful processor. His paper focused on the linguistic issues of a programming language which made concurrency intuitive when reasoning about collaborating processors. Gul Agha identified actors as computational agents which map incoming messages to a behaviour. Such

behaviour may include communicating with other actors, deciding how the next incoming communication will be processed as well as creating more actors. He favoured buffered asynchronous communication when it came to sending messages as it allows an actor to communicate with itself without waiting, as well as to promote efficient use of the processing power rather than wait for the receiver to accept communication. Gul Agha also recognised the fact that delivery of messages is not guaranteed on a network, and that all receiving buffers are bounded.

In the same year as Gul Agha's paper got released, a programming language called Erlang [15] made its first appearance. It was designed to address the highly concurrent nature of telephony applications. A high degree of fault tolerance was at the core of Erlang's design to minimise failure in telephone systems when changing the system's software. Erlang uses the actor model to allow for concurrent programming, such that each independent process communicates with other processes through the sending of messages. This makes "actors" and Erlang's lightweight "processes" interchangeable in the remaining discussion of Erlang. Each process has a FIFO mailbox (queue) which processes the messages in the order they were received. Erlang adopts asynchronous message passing, promoting developers to "fire and forget" messages. The language is also designed to let processes crash, as external processes can easily observe each other. When a process which provides the service crashes, the monitoring process can take over and resume the service. Erlang was continuously updated and found success in large scale mobile networks thanks to the actor model's potential to build reactive systems[16]. Erlang processes allow for elastic systems such that one can add more processes to address a larger workload, while monitoring processes allows for fault tolerance and responsiveness of the system.

Nowadays, several variants of the actor model are used to fit the requirements of modern languages and frameworks. Popular languages such as Scala [17] with Akka [18] and Elixir [19] (built on top of Erlang) use the actor model to allow developers to build scalable and distributed systems.

2.3 Similar Work

Several JavaScript frameworks which implement the Actor model are already available on the node package manager (npm) and in public git repositories.

Clooney [20] is described as an actor library for the browser by the Google Chrome team. It offers an API that takes in classes with functions inside which will be instantiated in a worker. Developers can call the defined functions inside the actors as if they were a regular class. This library acts more as an intuitive way around using the Web Worker API to offer parallelism rather than a fully fledged actor library.

Nact [21] is a more faithful implementation to the Actor model as it relies on message passing for communication between isolated actors rather than Clooney's function calls. One spawns an actor by defining a function with the state, message and system context as the parameters. The function will be called for every message that is sent to the spawned actor. Network transparency has not been implemented for nact, making it more complex to have actors spawned on remote devices over the internet.

Akka.js [22] allows developers to cross-compile Akka to JavaScript browsers and server-side runtimes. This framework allows developers to build distributed applications using separate browsers using WebRTC . It also incentivises developers to only be knowledgeable with Scala to deploy actors on both the browser and the server.

A project similar to this dissertation's prototype is called **Spiders.js** [11]. The paper identifies the problem of different API's being used for web workers and child processes, both of which are inspired by the actor model for constructing parallel systems using JavaScript. This project is focused on defining a single actor model API no matter if it's a client or server-side application. The paper benchmarks the usage of Spiders.js against JavaScript native web workers as well as the actor creation overhead for both. The results are in favour of using web workers for such

tasks. This is no surprise as the project is an abstraction built on top web workers on the client side.

TigerQuoll [23] took a different approach when providing parallelism to the language. It acknowledged the actor model as too limited for the requirements of more complex patterns that occur in modern applications. Instead it allows developers to use regular event based programming to register events for parallel processing. Other popular implementations on GitHub are Drama [24] and Comedy [25], both of which allow for sending messages to spawned actors. The latter implementation allows for spawning of actors on remote machines.

3. Design

The design chapter is split into three sections. The first section presents a high level overview of the API's features. The following section presents examples on how to use the framework's features through the use of code segments and explanations. The final section delves into the design decisions of the internal framework code which is abstracted by the API.

3.1 API Features

The API's goal is to allow developers to intuitively engineer concurrent and parallel code in JavaScript through the actor model. The developer can define how an actor will process each message using a callback function, and spawn it on a JavaScript runtime. This runtime can be either local or remote, and it can run on a Node.js instance or any modern browser with full interoperability. Programs will be made up of multiple spawned isolated actors, similar to the micro-services approach adopted by the industry. Each actor will be serving incoming requests (messages) and can optionally send messages to other actors such as the sender. Furthermore, an actor can spawn other actors and send the reference to the newly spawned actors to other actors embedded inside a message.

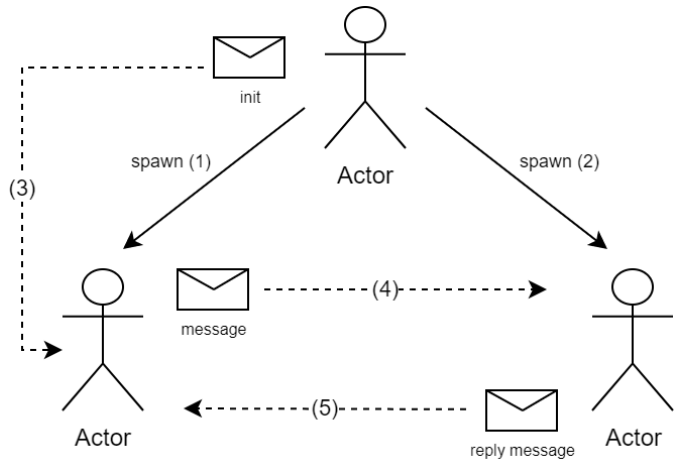

Figure 3.1: Actor spawning a set of communicating Actors with labelled order

By sending messages to locally spawned actors, each message will be processed to completion in a FIFO order. Actors differ from regular functions as each spawned actor maintains a state which can be manipulated by its defined behaviour when executed in response to incoming messages. The actor's behaviour can depend on the incoming message's structure or the actor's current state, offering seemingly non-uniform behaviour across messages. An actor can then be terminated where it will stop listening for incoming messages.

Since the spawned actors are isolated computational units, they are concurrent and thus potentially parallelisable. This API offers a mechanism to remotely spawn actors on other processes (shared memory) as well as over the web (distributed memory). One could host a website which remotely spawns actors on visiting clients to help speed up a computationally intensive task such as finding the next prime number. The more traction the website gains, the more computational power available.

Providing an intuitive way to remotely spawn and send messages to actors proved to be challenging. The prototype offers a WebSocket server which accepts a predefined number of clients. Each client is assigned a number, and when the number of expected clients connect the server sends a 'ready' message to all clients. This releases the barrier which waited for all clients to connect and allows commu-

nication between any two connected clients.

With the current implementation, the developer needs to connect the clients in a certain order to keep track of their incremental network numbers. The network number can then be used to indicate to the WebSocket server which client to communicate with. This is done when remotely spawning actors, as the developer needs to indicate on which connected client the actor will be spawned. Location transparency is offered when sending messages to such remotely spawned actors, as all the information required to establish communication is embedded within the actor reference. The API also allows the developer to make use of Node.js Cluster as well as Web Workers for faster communication, as it avoids adding the network stack and the overhead introduced when communicating with a potentially remote WebSocket server. All Node.js child processes and spawned web workers still need to establish a connection with the WebSocket server so that they are discoverable by remote devices.

3.2 Using the API

This section explores each of the functions that the framework exposes which developers will interact with to make use of the Actor Model in JavaScript. Using ES6 Modules the functions are imported as follows.

```
1 import actors from './actors.js';
2 const { init, spawn, spawnRemote, terminate, send} = actors
```

3.2.1 Spawning Local Actors

When the developer calls the **spawn** function, they will pass in the actor's initial state as the first parameter. The actor will maintain this state throughout every processed message and it can be manipulated by its behaviour. The actor's state allows for it to maintain memory throughout its lifetime when processing messages.

As a second parameter the developer must specify the actor's behaviour which has the actor's current state, current processed message as well as the actor's self reference as parameters. Using these parameters, the developer can define how the behaviour should manipulate the actor's state or how the message contents should be processed. It also allows the actor to pass a reference of itself to other actors using the behaviour's third parameter.

The example below defines the actor behaviour to print out each of the parameters. It then uses this function to spawn an actor with that behaviour, and a reference to that actor is returned

```
1 //Actor behaviour
2 const pongBehaviour = (state, message, self) => {
3     console.log("My state object is " + state);
4     console.log("I'm processing the message object " + message);
5     console.log("Actor self reference " + self);
6 };
7
8 //Spawn an actor with the above behaviour and an initial state
9 const pongReference = spawn({stateElement: "hello"}, pongBehaviour);
```

Note that with this implementation nothing is printed out, as the actor behaviour is only executed in response to a message which is sent to the spawned actor.

3.2.2 Sending Messages to Actors

The **send** function is used to send a message to an actor. It takes in the actor reference and message object to send as parameters.

```
1 send(pongReference, {messageVal: "This is a message!"});
```

One of the framework's key features is location transparency. Whether the actor resides locally or remotely on a different process or device, the framework internally

identifies the fastest medium to use for message transportation and sends it through that link.

3.2.3 Terminating Actors

An actor can be terminated using the **terminate** function. The function takes in the actor to terminate as the first parameter. An actor is terminated by removing its message listener as well as the internal record of the actor. The actor will however process its remaining messages as these are events already queued in the browser or Node.js's event loop. To forcefully terminate an actor, the second parameter can be set to true which will deactivate the actor. When the events are processed, the actor will perform minimal work ignoring the messages.

```
terminate(pongReference, false)
```

3.2.4 Connecting Processes to the Network

The **init** function handles the spawning of worker processes as well as connections with other Node.js and browser runtimes through WebSockets.

```
const webSocketServer = 'ws://localhost:8080';
const timeout = 10000;
const numberOfWorkers = 4;

init(webSocketServer, timeout, numberOfWorkers).then(ready => {
    if (ready.yourNetworkNumber === 1) {
        //Code for node 1 to execute
    }
})
```

WebSocket Network

Separate from the actor framework implementation, the prototype includes a WebSocket server implementation which handles a fixed number of expected WebSocket connections. As Node.js or browser runtimes connect to the WebSocket server through the invocation of **init**, the WebSocket server replies with acknowledgements. When the number of expected WebSocket connections are met, the server broadcasts to all connected clients that it is ready to receive and forward information. This effectively turns into a star network where each client can communicate with any other connected client through the WebSocket server. The server allocates and broadcasts unique identifying numbers to each connected client. This allows the developer to uniquely and easily identify remote hosts when required. The first parameter of **init** is the WebSocket server link while the second parameter specifies the timeout, which is the amount of time in milliseconds the client is willing to wait until it receives the indication that the server is ready to receive and forward information.

The WebSocket network could have been peer to peer to eliminate the network hop required for one client to communicate with another through the server. However, this would have required multiple or all hosts to be a WebSocket server listening for incoming connections. Furthermore, complex negotiation between peers would have been required when establishing WebSocket links and unique host identifiers on the fly. Another design alternative would be to establish WebSocket connections on the fly when required. However, this would mean that there may be a slowdown in performance during runtime as connection establishment is being done on demand.

Spawning Processes

The third parameter of the **init** function is the number of workers that are to be spawned. In the case of Node.js it spawns Cluster [7] child processes while in the browser it spawns Web Workers [8]. These created workers will also establish a

WebSocket connection with the server and are assigned a network number. The primary node or the browser's main thread will exchange their network number with the workers that they spawned so that they are aware on which nodes can be communicated with through the respective API's. With Node.js this would be Cluter IPC while with browsers it would be the Web Worker **postMessage** function. Both these methods are faster than WebSocket connections as you are avoiding the network stack which provides the flexibility for remote devices to communicate. The primary node/browser main thread is assigned with the task of forwarding messages between worker nodes as direct links are only established between the primary node and each of the workers. The workers are made aware of the assigned network numbers of neighbouring workers so that they can forward messages to the primary node when possible (rather than making use of the WebSocket network)

3.2.5 Remotely Spawning Actors

After invoking **init**, the developer can use **spawnRemote** to remotely spawn actors in other nodes. This function takes the network number as the first parameter, the initial state and behaviour as the other two parameters.

```
1  const pingPongBehaviour = (state, message, self) => {
2      console.log(message.val);
3      if(!(message.val-1 < 0))
4          send(message.replyTo, {val: message.val-1, replyTo: self});
5  };
6
7  //Specify timeout and number of workers to spawn
8  init('ws://localhost:8080', 10000, 2).then(async ready => {
9      //The primary node is always 1
10     if(ready.yourNetworkNumber === 1){
11         const ping = await spawnRemote(2, {}, pingPongBehaviour);
```

```
12          const pong = await spawnRemote(3, {}, pingPongBehaviour);
13
14          //Send ping a message. Output will be decrementing values from 5
                to 0
15          send(ping, {replyTo: pong, val: 5})
16      }
17  });
```

The **spawnRemote** function sends a request to spawn an actor to the remote node. The request has embedded within it the string representation of its behaviour which is reconstructed as a function on the recipient end. Just like the **send** function, it chooses the fastest transport mechanism to send the spawn request. In the example above, nodes 2 and 3 are the workers that the primary node spawned. The primary node is internally aware of the network numbers of the worker nodes it spawned, so it will make use of Cluster IPC to send these requests rather than the slower established WebSocket link through the network.

Location transparency is provided when obtaining the remotely spawned actor references. The developer can invoke **send** on the ping actor and the framework will use Cluster IPC to forward the message to the node. When the recipient actor processes message, the actor's behaviour will then send a message to the pong actor using the reference that it was sent by the primary node. The sending actor resides in a spawned worker node and it wishes to send this message to the recipient actor residing in a different worker node in the same cluster. The **send** function will send the message to the primary node (network number 1) which will then forward the message to the recipient worker node.

3.3 Actor Runtime

The JavaScript event loop is responsible for executing the application code. The runtime has a queue of messages where each message is mapped to a function

which processes that message. The event loop waits for the arrival of a message and processes the queue of messages when there is a backlog. Each message is processed to completion before other messages are processed. The Actor Model also maps messages onto functions which act as message handlers. Therefore, the framework's implementation takes advantage of the similar philosophies between the JavaScript runtime and the Actor Model.

The actors maintain their own message queue in an array. Once a message is received by an actor, it will schedule the processing of the message on the JavaScript event loop. This is done by scheduling a microtask which executes before the start of the next event loop [4][5]. The microtask is scheduled by creating a promise which is immediately resolved. This behaviour is similar to using Node.js' **process.nextTick()** [26]. This is more efficient than using the JavaScript **setTimeout()** function as that is scheduled as a macrotask which would execute in the following event loop.

```
1 messageEmitter.on(name, () => {
2     Promise.resolve().then(() => {
3         const message = actor.mailbox.shift();
4         if (message !== undefined)
5             cleanedBehaviour(actor.state, message,
6             {name: actor.name, node: actor.node});
7     })
8 });
```

With this implementation, actors only process messages only when other actors are not in the middle of processing a message. When the developer chooses to send a message to an actor, JavaScript schedules a microtask to be executed before the end of the current event loop. When an actor sends a message to another, it also schedules a microtask. Since JavaScript will process the current microtask to completion before fetching the next, the implementation ensures that at most one actor is in the middle of processing a message.

4. Evaluation

4.1 Single Threaded Performance

4.2 Parallel Execution Performance

5. Conclusion

A. This chapter is in the appendix

A.1 These are some details

```
this is some code;
I hope you found this template useful.
```

References

[1] E. International, "Standard ecma-262 - ecmascript 2021 language specification." https://262.ecma-international.org/12.0/, 2021. Accessed: 2022-04-03.

[2] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3, p. 235, Stanford Research Institute Menlo Park, CA, 1973.

[3] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 years of actors: A taxonomy of actor models and their key properties," in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pp. 31–40, 2016.

[4] "The event loop." https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. Accessed: 2022-03-18.

[5] "The node.js event loop, timers, and process.nexttick()." https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. Accessed: 2022-03-18.

[6] P. Haller, "On the integration of the actor model in mainstream technologies: the scala perspective," in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pp. 1–6, Association for Computing Machinery, 2012.

[7] "Node.js v17.7.2 documentation." https://nodejs.org/api/cluster.html. Accessed: 2022-03-19.

[8] "Web workers api." https://developer.mozilla.org/Web/API/Web_Workers_API. Accessed: 2022-03-19.

[9] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[10] D. Namiot and V. Sukhomlin, "Javascript concurrency models," *International Journal of Open Information Technologies*, vol. 3, no. 6, pp. 21–24, 2015.

[11] F. Myter, C. Scholliers, and W. De Meuter, "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*, pp. 51–60, 2016.

[12] "Ecmascript® 2017 language specification (ecma-262, 8th edition, june 2017)." https://262.ecma-international.org/8.0. Accessed: 2022-04-01.

[13] "Sharedarraybuffer." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer. Accessed: 2022-04-01.

[14] G. A. Agha, "Actors: A model of concurrent computation in distributed systems.," tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[15] J. Armstrong, "A history of erlang," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (New York, NY, USA), p. 6–1–6–26, Association for Computing Machinery, 2007.

[16] R. K. Jonas Bonér, Dave Farley and M. Thompson, "The reactive manifesto." https://www.reactivemanifesto.org/. Published: 2014-09-16, Accessed: 2022-03-17.

[17] "The scala programming language." https://www.scala-lang.org/. Accessed: 2022-04-03.

[18] "akka." https://akka.io/. Accessed: 2022-04-03.

[19] "elixir." https://elixir-lang.org/. Accessed: 2022-04-03.

[20] "Clooney." https://github.com/GoogleChromeLabs/clooney. Accessed: 2022-04-01.

[21] "Nact." https://nact.xyz/. Accessed: 2022-04-01.

[22] 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*, pp. 57–64, 2015.

[23] D. Bonetta, W. Binder, and C. Pautasso, "Tigerquoll: parallel event-based javascript," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 251–260, 2013.

[24] "Drama." https://github.com/stagas/drama. Accessed: 2022-04-02.

[25] "Comedy." https://github.com/untu/comedy. Accessed: 2022-04-02.

[26] "Understanding process.nexttick()." https://nodejs.dev/learn/understanding-process-nexttick. Accessed: 2022-03-18.