With the advent of JavaScript, at the turn of the century web applications were developed using the traditional client-server model. These applications were developed using distributed programming models in which programmers can express that a particular part of their application's state is local or remote within the executing code. In contrast, modern so-called distributed rich internet applications (DRIAs) distribute both their application logic and state across multiple servers and clients. In these applications the distinction between local and remote application state no longer holds. For example, a DRIA's clients and servers require parts of the distributed global state to be locally available.

This lack of expressiveness on behalf of distributed programming models unnecessarily burdens programmers. More precisely, programmers are forced to tackle the distribution of both logic and state across multiple clients and servers using distributed programming models that lack the necessary abstractions. For example, this requires programmers to manually synchronise the parts of the application's state distributed across the clients.

In this dissertation we present Triumvirate, a DSL tailored towards the development of DRIAs. Triumvirate provides data types specifically designed to represent various kinds of distributed state. These data types differ in the way they behave under concurrent updates and in their parameter passing semantics. Moreover, Triumvirate provides abstractions that allow programmers to deploy application logic across multiple servers and clients.

Concretely, Triumvirate provides a multitude of data types that allow programmers to implement distributed state with various consistency guarantees. Triumvirate automatically enforces these guarantees using state-of-the-art consistency mechanisms. Moreover, Triumvirate also allows for the implementation of distributed reactive state.

To do so we develop a novel propagation algorithm for decentralised reactive programs. We validate the different facets of Triumvirate through various proofs, benchmarks and real-life use cases.

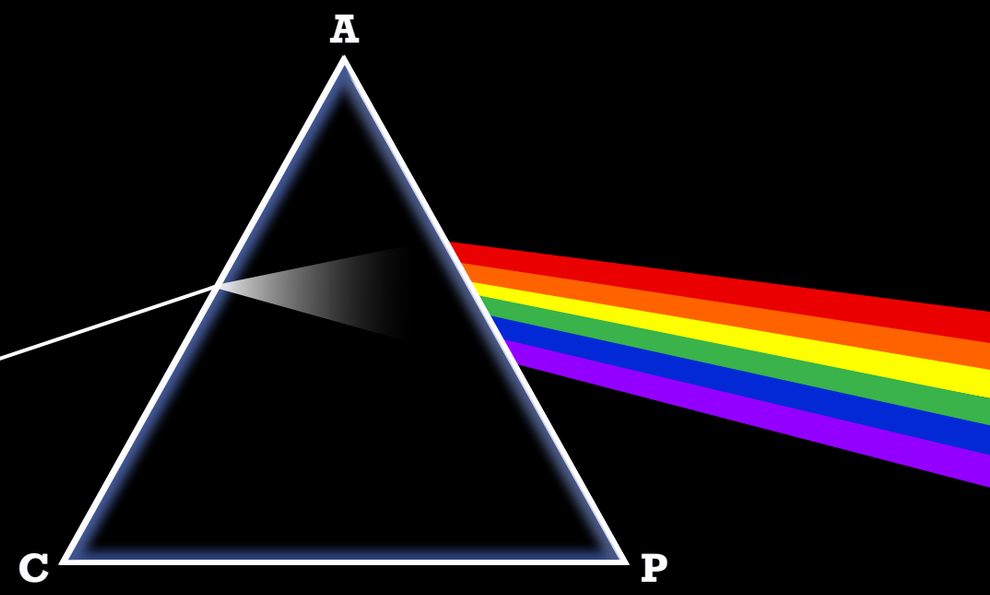
VRIJE UNIVERSITEIT BRUSSEL
UNIVERSITEIT GENT

ISBN 978-9-49307-953-3

TRIUMVIRATE
A Programming Language Design for Distributed Rich Internet Applications
Florian Myter

TRIUMVIRATE

A Programming Language Design for Distributed Rich Internet Applications



Florian Myter

December 2019

Dissertation Submitted in Fulfillment of the Requirement for the Degree of Doctor of Sciences

Promotors:

Prof. Dr. Wolfgang De Meuter
Vrije Universiteit Brussel

Prof. Dr. Christophe Scholliers
Universiteit Gent

Triumvirate: A Programming Language Design for Distributed Rich Internet Applications

Florian Myter

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

December 13, 2019

Promotors:
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel
Prof. Dr. Christophe Scholliers, Universiteit Gent

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Beat Signer, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium
Prof. Dr. Theo D'Hondt, Vrije Universiteit Brussel, Belgium
Prof. Dr. Heather Miller, Carnegie Mellon University, United States of America
Prof. Dr. Guido Salvaneschi, Technische Universität Darmstadt, Germany

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079533
NUR 989

The most educated person in the world
now has to admit
– I shall not say confess–
that he or she knows less and less
but at least knows less and less
about more and more.

CHRISTOPHER HITCHENS - *GOD IS NOT GREAT: HOW RELIGION POISONS EVERYTHING*

Abstract

With the advent of JavaScript, at the turn of the century web applications were developed using the traditional client-server model. These applications were developed using distributed programming models in which programmers can express that a particular part of their application's state is local or remote within the executing code. In contrast, modern so-called *distributed rich internet applications* (DRIAs) distribute both their application logic and state across multiple servers and clients. In these applications the distinction between local and remote application state no longer holds. For example, a DRIA's clients and servers require parts of the distributed global state to be locally available.

This lack of expressiveness on behalf of distributed programming models unnecessarily burdens programmers. More precisely, programmers are forced to tackle the distribution of both logic and state across multiple clients and servers using distributed programming models that lack the necessary abstractions. For example, this requires programmers to manually synchronise the parts of the application's state distributed across the clients.

In this dissertation we present Triumvirate, a DSL tailored towards the development of DRIAs. Triumvirate provides data types specifically designed to represent various kinds of distributed state. These data types differ in the way they behave under concurrent updates and in their parameter passing semantics. Moreover, Triumvirate provides abstractions that allow programmers to deploy application logic across multiple servers and clients.

Concretely, Triumvirate provides a multitude of data types that allow programmers to implement distributed state with various consistency guarantees. Triumvirate automatically enforces these guarantees using state-of-the-art consistency mechanisms. Moreover, Triumvirate also al-

lows for the implementation of distributed reactive state. To do so we develop a novel propagation algorithm for decentralised reactive programs. We validate the different facets of Triumvirate through various proofs, benchmarks and real-life use cases.

Samenvatting

Sinds de introductie van JavaScript rond de eeuwwisseling werden webtoepassingen ontwikkeld volgens het traditionele client-servermodel. Ontwikkelaars gebruikten hiervoor zogenaamde "gedistribueerde" programmeermodellen, waarin voor elk deel van de programmatoestand bepaald wordt of deze intern dan wel extern is ten opzichte van de uitvoerende code. Moderne, zogenaamde gedistribueerde rijke internet toepassingen (GRIT) distribueren echter zowel hun toestand als hun logica over meerdere servers en clienten. In zulke toepassingen bestaat het onderscheid tussen intern en extern niet langer. Een GRIT kan bv. vereisen dat sommige delen van de gedistribueerde globale toestand lokaal beschikbaar is op zowel de server als de client.

Het gebrek aan expressiviteit van gedistribueerde programmeermodellen maakt de jobs van programmeurs onnodig moeilijk. Programmeurs worden gedwongen om de distributie van programmalogica en -toestand te organiseren aan de hand van een programmeermodel dat de nodige abstracties mist. Bijgevolg moeten programmeurs bijvoorbeeld handmatig de toestand van verschillende clienten gelijkzetten.

In dit proefschrift presenteren we Triumvirate, een op maat gemaakte domein-specifieke taal voor het ontwikkelen van GRIT. Triumvirate biedt de gegevenstypen aan waarmee verschillende soorten gedistribueerde programmatoestand kunnen voorgesteld worden. Deze gegevenstypen onderscheiden zich van elkaar in de manier waarop ze omgaan met gelijktijdige veranderingen, alsook in de semantiek die ze toeschrijven aan het doorgeven van parameters. Bovendien biedt Triumvirate de nodige abstracties om programmalogica te verdelen over meerdere servers en clienten.

Concreet biedt Triumvirate een veelvoud aan gegevenstypen aan waarmee programmeurs gedistribueerde toestand kunnen implementeren met

verschillende consistentiegaranties. Triumvirate handhaaft deze garanties automatisch aan de hand van state-of-the-art consistentiemechanismen. Bovendien biedt Triumvirate ook ondersteuning voor gedistribueerde reactieve toestand. Hiervoor ontwikkelen we een nieuw propagatiealgoritme. We valideren de verschillende facetten van Triumvirate door middel van verschillende bewijzen, experimenten en gebruikstoepassingen.

Acknowledgements

I started my master thesis' acknowledgements by stating that John Donne was wrong and that I was, in fact, an island. I'll start these acknowledgements by stating that *I* was wrong and that Maynard is right:

> You were never an island
> Unique voice among the many in this choir
> Tuning into each other, lift all higher
>
> (A Perfect Circle — Disillusioned, Eat the Elephant)

Amongst all the songs I've sung over the years the one you're about to hear, or rather read, has been one of the more challenging. I would first like to thank the members of my jury, Viviane Jonckers, Beat Signer, Theo D'Hondt, Kris Steenhaut, Heather Miller, and Guido Salvaneschi for the time they spent reviewing this song.

The last five years have been rhythmed by the baton of two conductors: Wolfgang De Meuter and Christophe Scholliers. It goes without saying that this song only exists because of their incessant guidance throughout my academic career and throughout the composition of this song. What does not go without saying is that both conductors also significantly contributed to who I am and where I stand professionally today. For that I am more than grateful. Lastly, I thank my third (unofficial) conductor Eliza-Boit Gonsales for sparking my interest in distributed systems in the first place and for letting me be the worse teaching assistant she'll ever have. May her dream of a unified Spain one day come true.

Besides conductors, my academic choir also consists of fellow vocalists. A part of the ideas incorporated in this song were Tim Coppieters', who left the choir prematurely and gave me his blessing to bring them to fruition for which I thank him. I thank Jesse Zaman for meticulously

proofreading my song and for being foolish enough to cooperate on future ones. Of all my achievements in the past four years I am most proud of founding the jostiband-esque sub-choir known as the *church of iron*. I thank its ungrateful (De Troyer) disciples for helping me bring onto these heathens the lord's swoledom.

Last but surely not least I also thank a number of people in my familial choir. I would not be writing this if it weren't for my uncle Jacques, which got me interested in computers to begin with. I would also not be writing this without the love and support of the rest of my family, Marie-Paule, Katharina, Philippe, Nicolas, Aurélie and Chloé. Finally, there's nothing I could every say which would adequately thank Pauline, wife, best friend, love of my life and future mother of my child(ren).

Arnold is numero uno,
Florian Myter

Contents

Acronyms

CEL communicating event loops.

DOM document object model.

DRIAs distributed rich internet applications.

DRP distributed reactive programming.

QPROP queued propagation.

RIAs rich internet applications.

RPC remote procedure call.

Glossary

derivable Reactive state in distributed rich internet applications. Derivable state updates automatically as changes are made to the state from which it is derived..

derivation An instance of derivable state in a distributed rich internet application.

duplicable Locally active state in distributed rich internet applications. A node's control flow determines when and how updates are issued to duplicable state. Moreover, updates to a specific copy of duplicable state do not impact or modify other copies of the same state.

duplicate An instance of duplicable state in a distributed rich internet application.

horizontal distribution Distribution of logic or state across nodes that either reside solely on a web application's servers or solely on its clients.

microservice A cohesive independent process interacting via messages [DGL+17].

node A process or program that communicates with other processes or programs over a network to form a distributed system.

pass-by-copy An argument passing semantics in which the receiving node's method is invoked with a copy of the object sent by the sending node.

pass-by-derivation An argument passing semantics in which the receiving node's method is invoked with a derivation of the object sent by the sending node. The original object maintains a reference to its derivation such as to guarantee reactive state updates.

pass-by-replica An argument passing semantics in which the receiving node's method is invoked with a replica of the object sent by the sending node. The original object and the replica maintain references to each other such as to guarantee remotely active state updates.

replica An instance of replicable state in a distributed rich internet application.

replicable Remotely active state in distributed rich internet applications. A node's control flow determines when and how updates are issued to duplicable state. Moreover, updates to a specific copy of replicable state equally impacts all other copies of the same state.

vertical distribution Distribution of logic or state across (part of) a web application's servers and clients.

List Of Definitions

List Of Laws

Chapter 1

Introduction

Programming is the art, science and engineering of combining individual computer instructions into executable computer programs. These instructions are given semantic meaning by a programming language that also determines which combinations of instructions are valid (i.e. a grammar). When these instructions allow for executable programs to run across multiple machines we speak of *distributed* programming and *distributed* programming languages.

A common misconception about distributed programming is that it boils down to local (i.e. non-distributed) programming enhanced with communication instructions (e.g. *send* and *receive*). The reality of distributed programming and its differences with local programming is best exemplified by the following quote by Waldo et. al. in [KWWW94]:

> The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with differences in memory access paradigms between local and distributed entities.

Remote procedure call (RPC) [BN84] is one of the first programming paradigms to incorporate distribution into its design. In this design a program is able to invoke procedures defined by other programs residing on physically distributed machines. In general this dissertation uses the

generic term *node* to describe processes or programs that communicate with each other over a network to form a distributed system.

In the RPC model programmers are unable to syntactically distinguish between local (i.e. intra-node) or remote (i.e. inter-node) procedure calls. By now, it is generally accepted that obfuscating distribution from the programmer is bad programming language design [KWWW94]. Although adaptations to the object-oriented programming model have been made (CORBA [Gro12], Java RMI [Mic98], etc.) these have all inherited RPC's fallacies.

The RPC model is heavily critiqued for treating all data (i.e. procedures, objects, etc.) as local. Another distributed programming model from the same era, called *tuple spaces* [Gel85], modelled distribution inversely. In this model all data is treated as remote: programs read and write data to and from a datastore which is conceptually shared across the network[1].

More recent distributed programming models and languages tend to make the distinction between local and remote data explicit to the programmer. In contrast to older models, programmers are always aware whether a piece of data is local or remote. These modern distributed programming models are best exemplified by AmbientTalk [CGS+14], an object-oriented distributed programming language which heavily influenced this dissertation. AmbientTalk makes it lexically and semantically explicit to the programmer whether an object is local or remote within the executing code. Method invocations or field accesses on local objects (i.e. so-called *isolated* objects) happen synchronously through AmbientTalk's dot (.) operator, as is commonly the case in object-oriented languages. Method invocations or field accesses on remote objects (i.e. so-called *far references*) happen asynchronously through AmbientTalk's arrow (←) operator.

In this dissertation we argue that the binary data classification (i.e. local or distributed) provided by distributed programming models fails to meet the needs of modern distributed systems. More concretely, this dissertation focuses on the most widely used distributed system: the world wide web. Modern web applications comprise a large number of nodes spread across clients and servers. Each node executes a part of the appli-

[1]The TOTAM [GSMD14] tuple space model forms an exception to this rule. We discuss TOTAM in more detail in Section 3.1.2 of Chapter 3

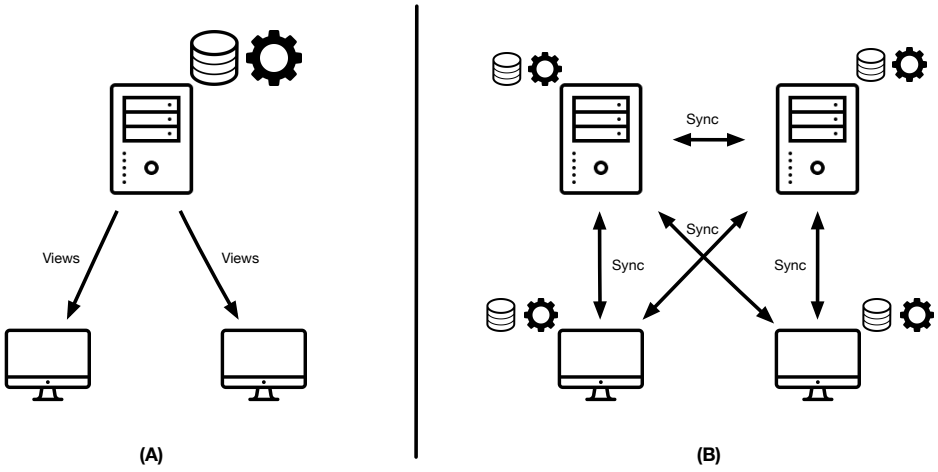


Figure 1.1: Early (A) versus modern (B) web applications.

cation's logic and therefore requires local access to the relevant parts of the application's state (i.e. for the sake of efficiency, privacy, offline availability, fault tolerance, etc.). However, this local data inherently models distributed (i.e. remote) state. As such, modern web applications blur the lines between what constitutes local and remote state.

In this dissertation we present a distributed programming model which tailors towards the distributed data needs of modern web applications. We identify three fundamental categories of distributed data. Our model specifies an API as well as parameter passing semantics for each data category. We implement this model in an object-oriented domain-specific language for web applications called *Triumvirate*.

1.1 Distributed Rich Internet Applications

Figure 1.1 compares two eras of web applications. Figure 1.1 (A) shows the architecture of early web applications. These applications mainly offer static web pages or limited user interfaces (e.g. hyperlinks, forms, etc.). In general a central server sequentially executes the application's logic (indicated by a gear icon in the figure) and maintains its state (indicated by a database icon in the figure). The clients in this architecture allow users to view the application's current state in their browser.

Rich internet applications (RIAs) extend these early web applications with functionality such as client-side storage, partial page updates, multimedia content, etc [FRSF10]. More precisely, in this dissertation we focus on *distributed rich internet applications (DRIAs)* [TCPL11].

Early web applications barely classify as distributed systems, given that the server exclusively manages the application's logic and state. This heavily contrasts with the architecture of DRIAs, shown in Figure 1.1 (B). Examples of DRIAs include Google's office suite [2], Facebook [3], webtops [SYS07], etc. DRIAs provide distribution on two levels:

Distributed Logic A DRIA's logic is distributed amongst different servers (a.k.a. *horizontal distribution*) as well as amongst servers and clients (a.k.a. *vertical distribution*). Horizontal distribution typically allows for servers to be spread across the globe in order to reduce latency on the client side (i.e. geo-replication). Vertical distribution allows clients to be responsible for parts of the application logic. In other words, a client connecting to a server-side service not only receives data to display but also computations to perform locally on said data.

Servers and clients also provide internal distribution. On one hand, servers are typically implemented as collections of autonomous (micro) services. On the other hand clients also provide support for multiple services (i.e. web workers) to run in parallel.

Distributed State The application's state is also distributed horizontally and vertically. There are two reasons for this distribution. First, each component in the network requires part of the application's state to execute its piece of the distributed logic. Second, for the sake of functional requirements: offline availability, privacy, performance, etc.

In other words, DRIA programmers not only tackle the complexities tied to web applications but they also tackle those tied to distributed systems. For example, the distributed state of DRIAs forces programmers to face the trade-off made famous by the CAP theorem [Bre00, GL02].

[2] `https://gsuite.google.com/` (last accessed: 05-12-2019)
[3] `https://www.facebook.com/` (last accessed: 05-12-2019

1.1.1 Distributed State and the CAP Theorem

In this dissertation, we follow the definition of the CAP theorem as given by Seth Gilbert and Nancy Lynch [GL02]. Assume a distributed system running atop a network of nodes that conceptually share a piece of readable and writeable memory. Given this system, we define *consistency*, *availability* and *partition tolerance* as follows:

Definition 1: Consistency

A distributed system is consistent if after a value is written to the shared memory, all subsequent reads return that value until it is overwritten by another value. This property is also known as *linearizability* [HW90].

Definition 2: Availability

A distributed system is available if nodes are guaranteed to receive *meaningful* results for all operations performed on the shared memory (e.g. a request-timeout exception is not considered meaningful). In other words, nodes are able to read from and write to the shared memory at any point in time.

Definition 3: Partition Tolerance

A partition separates the network of nodes into disjoint subnetworks. Communication across sub-networks is impossible for the duration of the partition. A distributed system is partition tolerant if it is able to maintain its consistency or availability properties in the face of these partitions.

The CAP theorem states that it is impossible for any distributed system to ensure that the operations on shared memory are consistent, available and partition tolerant. Distributed systems can only guarantee two out of these three properties. Because any real-world distributed system faces partitions at some point in time the programmer is left with the choice between offering availability or consistency. While the CAP theorem proves that sharing data in a partition-tolerant distributed system is either available or consistent this choice can be made for *each* shared piece of data.

1.1.2 Beyond Local and Remote Data

Reconsider Figure 1.1 (A). In such an architecture a binary model for distributed data (i.e. data is either considered local or remote to a specific node) suffices. From the server's perspective all data is local: the server is the sole component in the network to create, modify and delete data. From the clients' perspective all data is remote: clients render the state residing on the server.

A data model that classifies data as either local or remote fails to express the complexities of DRIAs. Each node in the network requires parts of the global, remote, application state to be locally available to execute its logic. In other words, the lines between local and remote data are blurred. Moreover, nodes can concurrently update the same piece of the application's state. Depending on the trade-off made with regard to the CAP theorem these concurrent updates might have different semantics. For example, a social media feed might prefer the availability over the consistency of concurrent updates. Conversely, bank accounts probably prefer the consistency of concurrent updates over their availability.

1.1.3 Problem Statement

The lack of a DRIA-oriented programming model forces DRIA developers to tackle two accidental complexities:

Collocation of Distributed Logic and State Programmers need to determine which node in the network is to execute which part of the DRIA's application logic. Moreover, programmers need to determine which parts of the application's state are required to execute said logic. In other words, programmers need to distribute their DRIA's application logic and state across multiple servers and clients. Moreover, each client and server potentially runs a different part of the application logic that requires a different part of the application's state. As such, programmers are forced to tackle the complexity that arises from **heterogeneity of logic across nodes in the network**.

Combination of Distributed State The various parts of a DRIA's distributed state can have differing consistency requirements. In other words, programmers must consider the trade-off imposed by the

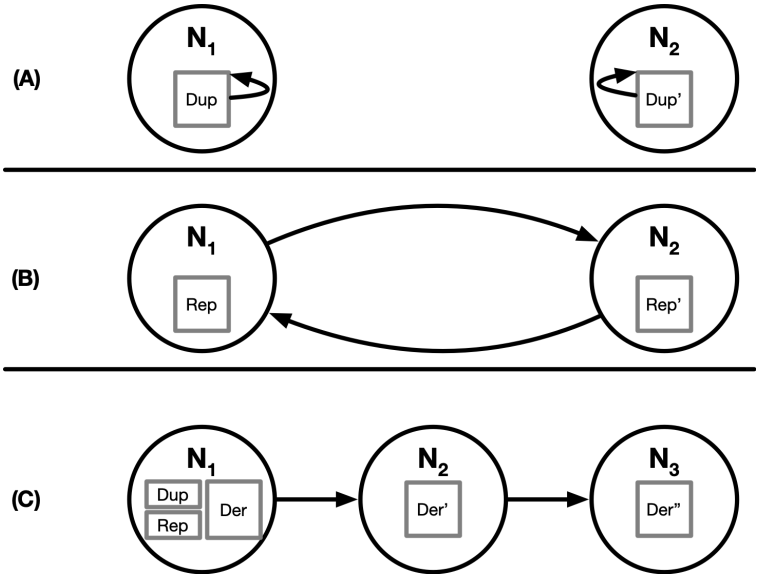


Figure 1.2: (A) Concurrent modifications on duplicates (B) Concurrent modifications on replicas (C) Concurrent modifications on derivations.

CAP theorem for each part of the application's state. Moreover, programmers must ensure that consistency guarantees are not broken by combining different parts of the state (e.g. by combining available with consistent state). As such, programmers are forced to tackle the complexity that arises from **heterogeneity of state within a single node in the network**.

1.2 Modelling Distributed State as Triumvirs

It no longer suffices to treat distributed state as either local or remote to a certain node in the network. In this dissertation we propose three categories that represent the different kinds of distributed state in DRIAs. These states differ in the way they respond to concurrent updates. More precisely, we introduce the following categories:

Duplicable Assume a DRIA running atop a network of two nodes N_1 and N_2, as shown in Figure 1.2(A). Furthermore, assume that both nodes have copies of the DRIA's state (i.e. *dup* and *dup'* in figure). This state is duplicable if two conditions are met. First, nodes ***actively***

update the state. In other words, the node's control flow determines the updates to the state. Second, nodes only update their ***local*** copy of the duplicable state. Such copies (i.e. instances of duplicable state) are called *duplicates*.

In our example *dup* and *dup'* conceptually represent the same part of a DRIA's state. However, duplicates provide locally active updates. In other words, updates made by N_1 to *dup* (indicated by the arrow in the figure) do not change N_2's *dup'* and vice versa.

Replicable Figure 1.2(B) assumes that two nodes N_1 and N_2 have two copies (i.e. *rep* and *rep'*) of a *DRIA*'s state. This state is replicable if two conditions are met. First, nodes ***actively*** update the state. We previously explained this condition. Second, updates on a local copy of replicable state impact all ***remote*** copies of the same state. Such copies (i.e. instances of replicable state) are called *replicas*.

Replicable state and replicas support remotely active updates. For example, updates made by N_1 to *rep* also apply to N_2's *rep'* and vice versa. In other words, *rep* and *rep'* are subject to updates issued by both N_1 and N_2.

Derivable Assume the network supporting a DRIA consists of three nodes, as shown in Figure 1.2(C). Each node has a copy of the DRIA's state (i.e. *der*, *der'* and *der"*). This state is derivable if its copies ***reactively*** update as a result of updates to duplicates or replicas. Such copies (i.e. instances of derivable state) are called *derivations*.

In contrast to duplicable or replicable state, nodes do not actively update derivable state. For example, assume the value of *der* is derived from a duplicate *dup* in node N_1. As soon as N_1 actively updates *dup* this triggers a chain of reactions that update *der*, *der'* and *der"* in order. In other words, derivable state and derivations support reactive updates.

This dissertation presents Triumvirate, a domain-specific language for modern web applications. The core of Triumvirate are the aforementioned three triumvirs (i.e. duplicable, replicable and derivable state), which allow programmers to easily manage complex and concurrent updates to state shared across nodes in a DRIA. Triumvirate defines an API, parameter passing semantics and interaction rules for each triumvir.

1.2.1 Triumvirate: Goals and Approach

Through the design and implementation of Triumvirate we aim to accomplish the following goals:

- We investigate which distributed programming model best fits DRIAs. This model serves as the foundation of our research. It provides the abstractions needed for programmers to reason about their DRIAs. In other words, it allows programmers to elegantly distribute their application's logic and state. Moreover, the model serves as a Petri dish for our research: it allows us to easily extend its semantics in order for us to

- devise a number of programming language abstractions tailored to distributed shared data. These abstractions enable programmers to express concurrent changes to the distributed state of DRIAs. Each abstraction defines parameter passing semantics and concurrent update semantics. To underpin our chosen distributed programming model and its state-oriented abstractions,

- we research how to *coordinate* the execution of the distributed logic and the concurrent updates to the distributed state. This involves research into the parameter passing semantics needed to distribute both logic and state. Moreover, this also involves research into conflict resolution strategies in the face of concurrent distributed state changes.

To achieve these goals we implement Triumvirate as a domain-specific language (DSL). More precisely, Triumvirate is a DSL implemented in JavaScript. We choose this approach over the library or middleware approach for the following reason. A (domain specific) programming language enforces a mental straitjacket onto the programmer. For example, it is impossible to reason about mutating state in a purely functional language. This frees the programmer from having to deal with, for example, race conditions in concurrent programs written in such a purely functional language. In our case the straitjacket forces the programmer to reason about their programs according to our distributed programming model. As such, all their programs' state must fit into the abstractions that we provide. However, by using our abstractions programmers are guaranteed a number of (data consistency) properties about their programs.

We prefer the *internal* DSL approach over the *external* one. The ecosystem of libraries, tools and runtimes for the web is vast and ever changing. It is therefore paramount that code written in our DSL interacts seamlessly with the myriad of available libraries and frameworks and vice versa. Achieving this compatibility with existing JavaScript code using an external DSL approach would require the addition of a foreign function interface or a substantial symbiosis layer with JavaScript. However, the internal DSL approach does allow programmers to break through Triumvirate's straitjacket. Programmers are able to break through Triumvirate's abstractions by using standard JavaScript abstractions.

1.3 Research Context

The work presented in this dissertation is funded by Innoviris through its Doctiris program[4]. In a nutshell, the goal of this program is to enhance the cooperation between academia and industry. This dissertation and the work that supports it are academic in nature. However, it is heavily influenced by its industrial driver scenarios. We therefore situate this work both in its industrial as well as academic context.

1.3.1 Industrial Context

Our project's industrial partner is Emixis, a Brussels-based fleet management company. Emixis equips their clients' fleet (e.g. trucks, cars, construction equipment) with sensor beacons. These beacons regularly upload a plethora of sensor readings (e.g. GPS coordinates, fuel levels, speed, etc.) to Emixis' servers. Through the Emixis platforms (e.g. Web, Mobile, etc) their clients are able to acquire business knowledge about their fleet (e.g. eco-driving statistics of employees, theft alerts, etc.).

Emixis' products inherently classify as DRIAs: data is produced by remote clients, collected and processed by a number of servers and finally visualised on mobile devices or browsers. In other words, the application's logic and state are distributed amongst various services running on multiple servers. Moreover, various parts of this distributed state have specific requirements with regards to concurrent updates. The Emixis case served as real-life motivation for the work presented in this dissertation.

[4] `https://innoviris.brussels/applied-phd`

Overall the software stack supporting their applications is implemented in languages and frameworks which lack built-in distribution abstractions. More specifically, the various update semantics for the distributed state are implemented in an ad-hoc fashion by Emixis developers through manual serialisation, polling, buffering, etc.

For example, their server is mostly implemented as a monolithic Java application. This server-side application's logic is driven by HTTP requests issued by clients. These clients are implemented using state-of-the-art JavaScript libraries (e.g. React [5]). However, these libraries focus solely on the user interface part of the client and lack real distribution functionality.

1.3.2 Academic Context

The work presented in this dissertation cross-cuts three fields of research. Our research relies on and contributes to the state of the art in each of these following fields:

Distributed programming Distributed programming languages provide language abstractions dedicated to distribution. However, the field of distributed programming in general also aims to support programmers of distributed systems through tools, custom language runtimes, dedicated database technology, etc. In this dissertation we focus exclusively on providing dedicated distributed programming language abstractions. We discuss this choice in Section 1.2.1 of this chapter.

Data consistency models and algorithms Concurrently mutating distributed and shared state leads to data consistency issues. Two components in the network might see different values for the same state. These inconsistencies and how to resolve them have been extensively studied within the database community. This has led to a wide range of consistency models: eventual consistency [SK09], sequential consistency [DSB86], etc. Moreover, this has also led to a wide variety of mechanisms that implement these consistency models (e.g. CRDTs [SPBZ11], the global sequence protocol [BLPF15], etc.). In our research we seek to apply these models and their mechanisms to coordinate concurrent state changes in web applications.

[5] `https://reactjs.org/` (Last accessed: 9-12-2019)

(Distributed) Reactive Programming Event-driven applications are traditionally written using callbacks or the observer pattern. As shown in [SAPM14] these techniques negatively impact code readability and maintainability. Reactive programming is a programming language paradigm which aims to avoid these drawbacks. In a nutshell, it does so by representing time varying values as composable first-class language constructs. This dissertation investigates how reactive programming principles can be applied to distributed applications. More precisely, we investigate how reactive propagation algorithms can be used to coordinate concurrent updates to distributed state.

We discuss existing distributed programming models and their shortcomings with regard to modern web applications in Chapter 3. Section 4.6 in Chapter 4 compares Triumvirate's replicables to various data consistency models and algorithms. Lastly, Section 5.10 in Chapter 5 compares Triumvirate's derivable state to distributed reactive programming approaches.

1.4 Research Vision and Realisations

The main vision behind this dissertation is that the distributed state of DRIAs can be classified as either duplicable, replicable or derivable. Moreover, we envision that future distributed programming languages and models should make this categorisation of their system's distributed state explicit. Triumvirate is an initial prototype that concretises this vision and primarily serves to convince the reader of its novelty, elegance and usefulness. However, this vision is too broad to realise in the span of a single doctoral dissertation and too vague to adequately validate. We therefore divide this dissertation's vision into three concrete contributions which have been realised and validated along the course of our research:

Actors for The Web We adapt the communicating event-loop model of actors to the context of web development. This results in Spiders.js, a DSL for distributed web programming which applies to both server-side as well as client-side code (cf. Chapter 6). Spiders.js is the first language to unify both distribution and parallelism for both server-side as well as client-side JavaScript. Using

Spiders.js programmers are able to elegantly distribute their web application's logic over multiple client and server-side actors with an acceptable performance overhead. We provide a qualitative analysis of Spiders.js' expressive power compared to standard JavaScript actors. Moreover, we measure Spiders.js' performance overhead over standard JavaScript actors through a set of benchmark applications.

Distributed State Classification We develop a classification of distributed and shared data in DRIAs: state is either *duplicable*, *replicable* or *derivable*. For each class of state we specify the distributed parameter passing semantics, the consistency guarantees it provides and how to handle concurrent and conflicting updates (cf. Chapter 4 and Chapter 5). We validate one of these classes (i.e. replicable) by developing an interactive slide deck used in front of a real-world audience during the Onward! 2018 conference[6].

Decentralised Reactive Change Propagation Triumvirate relies on state-of-the-art consistency mechanisms to handle concurrent changes to duplicable and replicable state. With regards to derivable state we present a novel mechanism inspired by distributed reactive programming that allows for reactive updates in decentralised applications. We showcase that this algorithm outperforms existing solutions and prove its correctness (cf. Chapter 5).

1.5 Supporting Publications

A number of publications support the work presented in this dissertation. We consider the following publications to be essential to the ideas presented in this dissertation:

- **Parallel and Distributed Web Programming with Actors**
  [MSDM18b]
  Programming with Actors
  Florian Myter, Christophe Scholliers and Wolfgang De Meuter

  ---

  This paper motivates the need for a distributed programming model

[6] see `https://youtu.be/17cHhDDpJbg` for a video of the presentation

for web applications. It introduces Spiders.js: a JavaScript implementation of the communicating event-loops actor model. Spiders.js allows web programmers to elegantly express both parallelism as well as distribution. Spiders.js serves as the technological foundation on which this dissertation is built.

- **Distributed Reactive Programming for Reactive Distributed Systems**
  [MSDM19]
  The Art, Science, and Engineering of Programming 3,Volume 3, Issue 3, Article 5
  Florian Myter, Christophe Scholliers and Wolfgang De Meuter

  ---

  Introduces a novel change propagation algorithm called QPROP. The novelty of QPROP lies in the fact that it supports reactive distributed systems (i.e. responsive, resilient, elastic and message driven) while providing a number of essential guarantees (i.e. glitch freedom, eventual consistency, etc.). QPROP guarantees the correctness of concurrent changes to derivable state.

- **A CAPable Distributed Programming Model**
  [MSDM18a]
  In Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software
  Florian Myter, Christophe Scholliers and Wolfgang De Meuter

  ---

  This paper argues the need for language-level abstractions to represent distributed and shared state. It introduces two object-oriented abstractions (i.e. *availables* and *consistents*) which represent the trade-off distributed programmers need to make with regard to the CAP theorem. Moreover, it provides two naive implementations for these abstractions and details how they integrate in the Spiders.js model. *Availables* and *consistents* serve as the basis for replicable state.

Besides these main publications a number of smaller workshop publications [MSDM16, MCSDM16, VdVDKMDM17, VdVMDKDM17, MSDM17, BMG18] helped form the ideas presented in this dissertation.

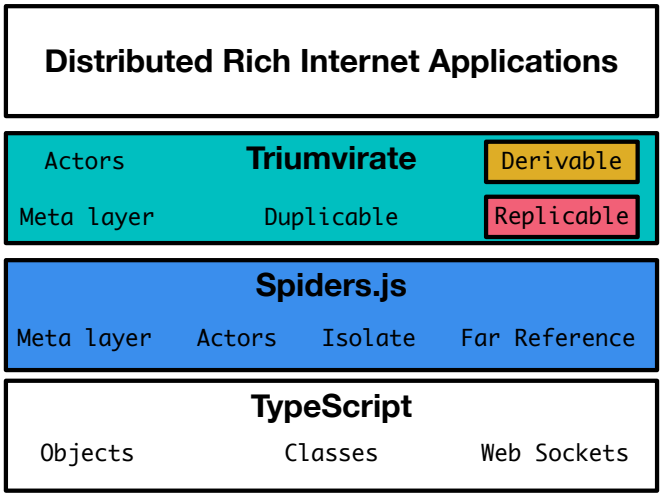


Figure 1.3: Overview of the Triumvirate technology stack.

1.6 Dissertation Roadmap

Figure 1.3 provides an overview of this dissertation's artefacts. More specifically the figure shows the major concepts introduced by each technological layer. Starting from the bottom, Spiders.js relies on fundamental concepts offered by TypeScript (e.g. objects, sockets, etc.). In turn, Spiders.js offers an actor-based abstraction layer over standard TypeScript. Triumvirate exposes a modified version of Spiders.js' actors and introduces the concepts of duplicable, replicable and derivable state. Finally, DRIAs programmers implement their applications atop Triumvirate.

The colors used in Figure 1.3 indicate the chapter in which a particular concept is discussed. More specifically, the remainder of this dissertation is organised as follows:

Chapter 2: Developing Web Applications in Triumvirate introduces our use case at Emixis and the running example for this dissertation. We discuss four requirements for distributed programming model to support DRIAs. Finally, we give the reader an overview of programming with Triumvirate by partially implementing the running example.

Chapter 3: State of the Art in Distributed Programming for Web Applications gives an extensive overview of existing dis-

tributed programming models. We use the requirements distilled in Chapter 2 to compare these approaches with each other and Triumvirate in the context of DRIAs.

Chapter 4: Replicating Remotely-Active State discusses replicable state in detail. More specifically, we start by discussing the need for two kinds of replicable state (i.e. strong replicable and eventual replicable state) that represent two ends of the CAP spectrum. We then discuss how these two kinds of replicable state are used to develop DRIAs by partially implementing our running example. This is followed by an overview of the rules that govern the interaction between strong replicable and eventual replicable state. Finally, we discuss the implementation of both kinds of replicable state using state-of-the-art replication and consistency mechanisms.

Chapter 5: Deriving Reactive State discusses derivable state in detail. We start by discussing the parallels between derivable state and distributed reactive programming and motivate the need for a novel distributed reactive runtime. We then show a practical example of derivable state by partially implementing our running example. Finally, we provide a complete specification of the reactive propagation algorithm used by Triumvirate to update derivable state.

Chapter 6: A Platform for Distributed Web-oriented Programming Languages gives an overview of Spiders.js, the actor framework atop which Triumvirate is implemented. We start by motivating the need for an actor framework in the context of the web generally and JavaScript specifically. We discuss the implementation of a collaborative code editor in Spiders.js and we provide an exhaustive overview of Spiders.js' base and meta-level constructs.

Chapter 7: Conclusion summarises the main vision behind Triumvirate, provides an overview of the individual realisations behind this dissertation and discusses avenues for future work.

Chapter 2

Developing Web Applications in Triumvirate

This dissertation presents three core contributions. First, a distributed programming model for DRIAs. Second, a classification of distributed and shared state based on their update semantics. Third, a novel algorithm to support updates to a specific category of distributed state.

Each of these contributions stands on its own. We discuss each contribution in detail in the following chapters. However, they are part of our overarching vision for a distributed programming model for DRIAs which takes shape as the Triumvirate DSL. We therefore start by giving the reader an overview of the rationale behind Triumvirate as well as a highlight of its most important features.

The problems which Triumvirate aims to solve are best explained using an example. To this end we introduce a running example employed throughout this dissertation. The example directly stems from our cooperation with Emixis and is a simplification of the system they operate in production.

2.1 Running Example

Imagine a technical service company. This company dispatches technicians to on-site jobs to repair or maintain the company's clients' equipment (e.g. generators, boilers, etc.). Through Emixis' platform this technical service company is able to track and manage both its technicians as well as the

technicians' equipment (e.g. the fleet of vans, specific instruments, etc.). An online dashboard provides dispatchers with an overview of the state of the technicians and the state of their equipment. This dashboard updates as soon as either states (i.e. that of the technicians or that of the equipment) change. For example, the dashboard shows a real-time overview of each equipment's location. Furthermore, the dashboard enables dispatchers to view and update data sent by technicians in the field (e.g. planning, pictures and descriptions from a particular job, etc.).

There are two main sources of data in the system. First, equipments are tagged with beacons that periodically upload sensory input data (e.g. positions, fuel levels for vehicles, etc.). Second, technicians are equipped with mobile phones installed with the system's mobile application. This application provides two main functionalities. First, it allows the technicians to view and edit their job schedule. Second, it allows the technicians to view and edit information about specific jobs (e.g. job descriptions, pictures, etc.).

2.1.1 Application Requirements

The fleet management applications provided by Emixis fulfil a multitude of functional and non-functional requirements. In the case of our running example it is paramount for its user (i.e. the technical service company) that the application fulfils the following two non-functional requirements:

Reactivity A part of the dashboard functionality is to send alerts to dispatchers upon occurrence of certain events. For example, an alert is issued whenever a van leaves the company premises outside of working hours. As such, our application's server must respond to requests as fast as possible. In other words, the application's server-side must be able to scale with the amount of members of the fleet and technicians changing their state.

Offline availability The conditions in which the technicians work vary (e.g. tunnels, etc.). This means that the application might lose connection with the server for extended periods of time. Technicians must be able to use the application even while being offline. More precisely, a technician must always be able to edit the information of the job he is currently working on, regardless of network connectivity. However, the job schedule should only be available when

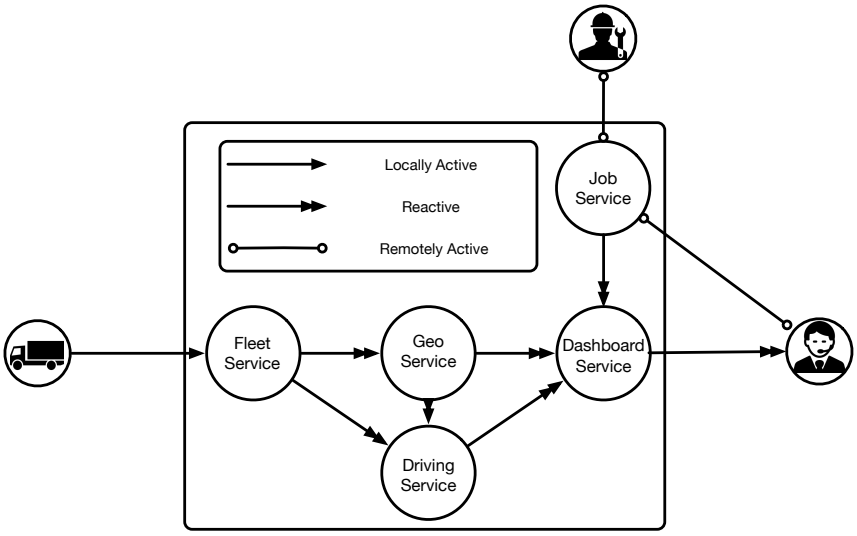


Figure 2.1: Overview of the fleet management application.

the technician is online. This is to avoid the situation in which the dispatcher and the technician have different views over the same schedule.

2.1.2 Fleet Management as a Distributed Rich Internet Application

Figure 2.1 gives an overview of the fleet management system. More precisely, the figure provides an architectural overview of the system's implementation. Circles represent the different nodes atop which the application runs. Nodes labelled with *"service"* reside on the application's server. The arrows in the figure represent the flow of data across these nodes.

The fleet management application adheres to the criteria for DRIAs (see Chapter 1). More precisely, its logic and state are distributed amongst the nodes in the network.

2.1.2.1 Distributed Logic

The application's logic is distributed between the server which mostly consumes data, and the clients which mostly produce data. There are three kinds of clients in the application (indicated with the truck, technician and

dispatcher icons in Figure 2.1). First, vehicles are equipped with beacons that regularly upload sensory data to the server. These clients solely produce data, and their logic is therefore rather limited. Second, technicians are equipped with smartphones. The mobile application running on these phones contains a part of our system's entire logic. More specifically, the application contains logic to interact with all job-related functionality (i.e. update job descriptions, view job schedule, etc.). A third and last kind of client are the dispatchers, which have an overall view of the system's state through the dashboard. Moreover, dispatchers are able to modify data related to the technicians' jobs (e.g. job description, planning, etc.). Both technicians and dispatchers therefore produce and consume data.

Besides being distributed amongst clients and server, our application's logic is also distributed amongst multiple server-side nodes. As discussed in the requirements, our application's server-side logic must scale with the amount of clients and fleet members issuing requests. Microservices have recently gained traction as an architecture which facilitates the development of scalable web application servers [VGC+15]. Each microservice is an autonomous unit of computation responsible for a particular piece of the server's functionality. To ensure scalability, our server distributes its logic over multiple microservices (e.g. a service which converts the GPS coordinates sent by fleet members into street addresses). Figure 2.1 gives an overview of the most important microservices in our application. The *fleet service* serves as an entry point for fleet members to upload their data to the server. The service deserialises and persists the uploaded data. The *geo service* converts a fleet member's GPS coordinates into physical street addresses. The *driving service* calculates eco-driving scores and generates alerts based on a fleet member's data and physical location. The *job service* allows technicians to query and modify job-related data. Moreover, the service also accepts modifications to job data from dispatchers. Lastly, the *dashboard service* combines fleet member and technician data into a coherent whole.

2.1.2.2 Distributed State

The fleet management application contains three kinds of distributed state. The first kind, indicated by the simple arrow in Figure 2.1, represents the raw data uploaded by fleet members. Fleet members periodically send this data as GPRS packets to the *fleet service*. Conceptually this

means that fleet members periodically ***duplicate*** their state over to the server. We say that the raw fleet data is ***duplicable*** and ***locally active*** with regards to updates. Updating a duplicate of the raw fleet data only affects the concerned duplicate. For example, modifications by the fleet service do not affect the fleet member's state.

A second kind of data, indicated by the double arrows in Figure 2.1, concerns the internal (i.e. within the server-side of the application) representation of the fleet data. Microservices distribute this data amongst themselves by ***deriving*** streams from it. For example, the *fleet services* derives two new output streams by serialising its input stream of fleet data. These two output streams are then derived by the *geo* and *driving* service into addresses and driving related data respectively. Finally, the *dashboard* service derives a dashboard from the *geo* and *driving* streams. We say that the internal fleet data is ***derivable*** and ***reactive*** with regards to updates. Updates to the source fleet data affects all its derivations. As soon as new data is uploaded by a fleet member all derivations made by the microservices are recomputed.

The last kind of data, indicated by the lines ending with dots in Figure 2.1, represents all job related information. This data is ***replicated*** amongst technicians, the job service and dispatchers. In other words, each of the aforementioned services has a local copy of the data. This ensures that the offline availability requirement of our application is fulfilled. As a result nodes are able to concurrently update their local replica, which can lead to conflicts (e.g. a technician and a dispatcher concurrently editing a job description). A synchronisation mechanism must therefore ensure that these conflicts are avoided or automatically resolved. We say that the job data is ***replicable*** and ***remotely active*** with regards to updates. Updates by one service to the job data results in an equal update for all replicas. For example, if a technician updates the description of a job this update also becomes visible for the dispatcher.

2.2 Requirements for a DRIA-oriented Programming Model

As we discuss in the previous section, the fleet management application exhibits the characteristics of a DRIA (i.e. its logic and state are distributed amongst nodes in the network). We argue that we currently lack

the programming technology to elegantly implement all facets of this application. We identify four requirements that are essential for a future DRIA-oriented distributed programming model.

R_1: **The model provides modular abstractions for distributed logic**

A DRIA's logic is distributed amongst two axes. First, *vertically* between client and server. Second, *horizontally* between different autonomous nodes on the server and on the client. This requires the programmer to divide the application's logic into distributed components and deploy these components across the various nodes in the network.

Such abstractions are commonplace in most distributed programming models (e.g. actors, tuple spaces, etc.). However, this is far from trivial in the context of web applications where clients are essentially implemented as scripts of standalone JavaScript code. Hence, a DRIA-oriented programming model should aid the developer in this task by providing modular abstractions for distributed logic.

These abstractions should enable the programmer to divide the application's logic into modular units of distributed logic. Moreover, the abstractions are location agnostic: they allow for the development of both server and client-side distributed logic. Finally, a set of built-in communication primitives accompany the abstractions. Using these primitives, programmers should be able to coordinate the different pieces of logic executing concurrently.

This requirement has also been identified by related work [PLCSF07, FCBC10], albeit in the context of model-driven engineering of RIAs. In a nutshell, these approaches argue that RIA-aware conceptual modelling must provide users the tools to express the placement of distributed logical components.

R_2: **The model guarantees data consistency properties specified by the programmer**

The state of a DRIA is, similarly to its logic, distributed horizontally and vertically. As such, this state can concurrently be updated by multiple nodes in the network. According to the CAP theorem (see Section 1.1.1 in Chapter 1) each of these concurrent updates

trades off consistency, availability and partition tolerance. This requires the programmer to encode this trade-off per operation on a shared piece of state in their DRIA. Moreover, some of these operations might conflict and make parts of the state inconsistent with each other. This further requires the programmer to write conflict-handling code.

Hence, a DRIA-oriented programming model should allow programmers to declaratively specify the CAP trade-off for operations on shared state. Ensuring that this trade-off is respected throughout the application's lifetime should be the responsibility of the model's implementation. Moreover, the model's implementation should also strive to automatically resolve conflicting updates on shared state.

Enabling programmers to explitly deal with the CAP trade-off using language constructs (i.e. the basis for this requirement) is the premise of a multitude of papers in the more general context of distributed programming [GPS16, MVR15, CDMB16, ZN16, MS17, Mei17].

R_3**: The model supports multiple parameter passing semantics**
A DRIA moves parts of its state between various nodes throughout its lifetime (e.g. a vehicle's state flows through various microservices in the fleet management example). Maintaining the state's consistency guarantees requires a significant amount of bookkeeping. This forces the programmer to implement various data dissemination strategies and to maintain the state's bookkeeping. For example, in our fleet management application vehicles and technicians disseminate their state across multiple microservices. The vehicles' state is disseminated as derivations while technicians' state is disseminated as replicas. Derivations require the necessary bookkeeping to react to active state changes. Replicas require the necessary bookkeeping to ensure that an update to one replica impacts all other replicas of the same state.

Hence, a DRIA-oriented programming model should relieve the programmer from writing bookkeeping code manually. Rather, each type of state should automatically be disseminated across nodes with the appropriate bookkeeping code. In other words, a DRIA-oriented programming model should support multiple parameter

passing semantics (e.g. pass-by-replication, pass-by-derivation, etc). More concretely, such a model should at least provide the semantics needed to implement duplicable, replicable and derivable distributed state.

A wide variety of parameter passing semantics (e.g. *pass-by-copy-restore* [TG11], *call-by-move* [JLHB88], etc.) have been introduced over the years by various distributed programming languages. In general, these semantics either serve as means for optimisation (e.g. Emerald's *call-by-move* [JLHB88]) or as an integral part of the language design (e.g. AmbientTalk's *pass-by-far-reference* [CGS+14]). With this requirement we argue the need for parameter passing semantics tailored to the needs of distributed state in DRIAs.

R_4**: The model allows for extensible parameter passing and state update semantics**
In general, the implications of the CAP theorem on distributed systems is still an active field of research [DPMDT+19, LPR18, SBP+18]. As such, new consistency models and conflict resolution algorithms are regularly developed.

Hence, a DRIA-oriented programming model should allow programmers to extend or override its built-in parameter passing and state update semantics. This allows programmers to easily implement novel consistency models by overwriting specific parts of the underlying programming model.

This requirement is inspired by related work in the context of database replication such as Gorda [CJPR+07]. In a nutshell, Gorda is a reflective interface that allow developers to implement custom replication protocols atop existing database management systems (DBMS) by overwriting how transactions are processed by the DBMS. With this requirement we argue the need for a similar approach in the context of DRIA-oriented programming models.

2.3 An Overview of Triumvirate

In this section we give a brief overview of Triumvirate, a DSL that fully embraces the complexities associated with developing DRIAs. Using Triumvirate, programmers express how their application's logic should be

distributed, how its state should be distributed and how specific parts of the distributed state update.

Triumvirate's distribution model has heavily been inspired by the AmbientTalk [CGS+14] language. Triumvirate represents pieces of distributed logic by means of communicating event-loop actors. In other words, both the server as well as the client are implemented as collections of actors. Actors coordinate their execution by means of asynchronous messages.

Triumvirate is an object-oriented language that offers three classes: *Duplicable*, *Replicable* and *Derivable*. Programmers implement the distributed state in their DRIAs by extending these classes, thereby inheriting from their built-in behaviours. Each class specifies how it behaves upon mutation of one of its properties. Concretely these behaviours are the following:

Duplicable The *Duplicable* class implements the most prevalent kind of distributed state. That is, state that crosses actors by copy and which provides locally-active updates. In other words, an actor's control flow determines when updates are applied to a duplicate (i.e. an instance of the duplicable class). Moreover, actors are only able to update their local copy of a duplicate. All native Triumvirate data types (i.e. string, numbers, etc.) are duplicable state.

Replicable The *Replicable* class allows programmers to implement remotely-active state. As is the case for duplicates, it is an actor's control flow that determines when updates are applied to a replica (i.e. an instance of the replicable class). However, when an actor updates a replica this affects all other copies of said replica. Concretely, replicas are sent across the network via pass-by-replication semantics. This entails that a copy of the replica is made whenever it crosses actor boundaries. Moreover, this copy keeps a reference to the original replica. Updates to the original replica are also applied to the copy and vice versa. Chapter 4 discusses the implications of these updates with regard to the CAP theorem at length.

Derivable Reactive state is implemented by means of the *Derivable* class. Reactive state differs from active state in that updates to reactive state are not dictated by an actor's control flow. Rather, reactive state updates automatically as a result of active state changes. Con-

cretely, programmers are unable to extend the *derivable* class or explicitly instantiate new derivable instances. Rather, programmers create derivations (i.e. instances of the derivable class) by *deriving* them from other objects (i.e. duplicates, replicas or derivations). To do so programmers use derivation functions that take a number of objects as input arguments and return a new derivation. As soon as a source object's state is modified the derivation function is re-applied to compute the output derivation's new state. The reason behind this peculiar design decision is explained at length in Section 5.3 of Chapter 5.

Derivations adhere to the following parameter passing semantics. Assume a derivation *d* is sent from one actor to another (i.e. the derivation is part of an asynchronous message). The sending actor starts by creating a new derivation *d'* from *d* and sends this new derivation to the receiving actor. Meanwhile, Triumvirate creates a dependency between *d* and *d'* (i.e. all updates to *d* automatically update *d'*). The combination of these dependencies form a directed acyclic dependency graph that dictates the order in which derivations are to be updated. We call this parameter passing semantics *pass-by-derivation*. Chapter 5 discusses the algorithm responsible for this propagation of updates at length.

It is impossible for a language to provide built-in support for *all* nuances of distributed update semantics. For example, there exists a myriad of synchronisation mechanisms to ensure the consistency of two replicas [LS76, SPBZ11, BLPF15, KKW19]. Although Triumvirate provides a number of built-in update and parameter passing semantics it also allows programmers to implement their own. A mirror-based meta-programming layer, inspired by [MVCT+09], provides a number of overridable hooks into the Triumvirate runtime.

2.3.1 Distributing Logic Using Actors

The following sections give a brief introduction to programming in Triumvirate. We introduce Triumvirate by means of small examples that highlight its most important features. We implement our running example and discuss various aspects of Triumvirate in more detail in the following chapters.

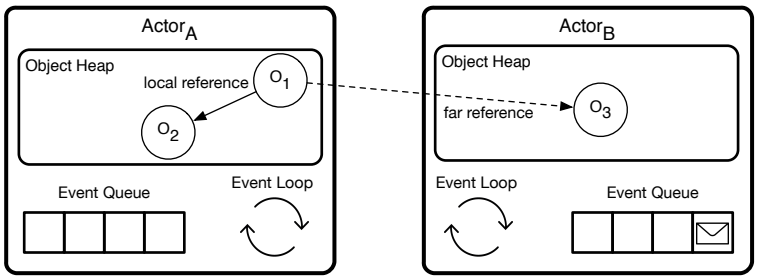


Figure 2.2: Actors as communicating event loops [MTS05].

Triumvirate is the first language to apply the communicating event loops (CEL) actors model [MTS05] to the context of web applications. In Chapter 6 we provide an in-depth explanation of how Triumvirate actors map onto low-level JavaScript constructs. We start by giving a brief explanation of the CEL model in general before highlighting its application in Triumvirate.

2.3.1.1 Communicating Event Loops

Figure 2.2 gives an overview of the CEL model. In this model, actors contain a heap of objects, an event queue and an event loop. We say that an actor *owns* an object if said object resides in the actor's heap of objects. These objects can hold references to other objects, as is the case in traditional object-oriented paradigms.

There are two kinds of references in the CEL model: *local* references and *far references*. Consider the example depicted by Figure 2.2. $Actor_A$ owns objects O_1 and O_2. O_1 holds a *local* reference to O_2, given that both objects reside in the same heap. This entails that all method invocations or field accesses by O_1 on O_2 happen in a standard sequential fashion. The reference which O_1 holds to O_3 is a far reference since O_3 is owned by $Actor_B$. Method invocations or field accesses by O_1 on O_3 are asynchronous. Behind the scene these invocations or accesses are translated into asynchronous messages sent to $Actor_B$. Each such message is enqueued in $Actor_B$'s event queue. $Actor_B$'s event loop repeatedly selects the first event from the queue and performs the invocation or field access contained in the dequeued event.

There are two kinds of objects in the CEL model: those that cross actor boundaries using *pass-by-copy* semantics and those that cross actor boundaries using *pass-by-far-reference* semantics. For example, O_1 invokes a method on O_3 and provides O_2 as argument to this invocation. Eventually O_3's method will be invoked with a far reference to O_2. Conversely, if O_3's method would be invoked with copyable arguments (e.g. numbers, strings, etc.) then O_3's method would obtain local references to the copies of these arguments.

2.3.1.2 Actors in Triumvirate

Listing 2.1 provides the definition of two Triumvirate actors that implement a simple producer-consumer application. In other words, a single producing actor sends data to one or more consumer actors. Consumers start by registering themselves at the producer. They do this in their *init* method (line 5), which the Triumvirate runtime invokes when an actor is spawned. Concretely, consumers invoke the *register* method on the producer actor with a self reference as argument. For the sake of brevity we assume that consumer actors have a *far reference* to the producer actor. The *register* invocation results in an asynchronous message sent to the *producer* actor and returns a promise. This promise resolves with the return value of *producer*'s *register* method (line 16). In turn, the *producer* actor invokes the registering consumer's *consume* method.

This simple example showcases the strength of using actors as a unit of distribution for DRIAs. The code provided in Listing 2.1 remains exactly the same regardless of the distribution axis along which it is deployed (i.e. horizontal or vertical) or locality (i.e. client or server). In other words, the actors can run as independent entities on a single machine (i.e. in the browser or on the server), on multiple servers, on multiple clients or on clients and servers without having to change the source code.

2.3.2 Distributing State Using Triumvirs

For each of the three classes of distributed data we provide a simple example that highlights their semantics as well as their API. These examples all build forth on the producer-consumer actor example.

```
class Consumer extends Actor{
 producer : FarRef<Producer>

 init(){
  this.producer.register(this).then(_=>{
   console.log("Registered")
  })
 }

 consume(data){
  //...
 }
}

class Producer extends Actor{
 register(consumer : FarRef<Consumer>){
  consumer.consume("hello world")
 }
}
```

Listing 2.1: Defining Triumvirate actors.

```
class Message extends Duplicable{
 timeStamp : number
 text : string

 constructor(text : string){
  this.timeStamp = Date.now()
  this.text = text
 }
}
```

Listing 2.2: Defining duplicable state.

2.3.2.1 Duplicates

All native Triumvirate data types (i.e. strings, numbers, booleans and arrays thereof) are instance of the *Duplicable* class. However, programmers are also able to define more complex duplicable data structures. Assume that the *producer* actor generates messages to be consumed by the *consumer* actors. Listing 2.2 defines such a *Message* data structure that contains a field for a time stamp and a field for the actual text.

Listing 2.3 shows how the *Producer* and *Consumer* make use of *Message* instances. We omit unchanged code compared to Listing 2.1 for the sake of brevity. The *producer* creates a new instance of *Message* whenever a *consumer* registers (line 10). Subsequently, this message is sent to the

```
1  class Consumer extends Actor{
2   consume(msg : Message){
3    console.log(msg.timeStamp + " : " + msg.text)
4    msg.text = ""
5   }
6  }
7
8  class Producer extends Actor{
9   register(consumer : FarRef<Consumer>){
10   let msg : Message = new Message("hello world")
11   consumer.consume(msg)
12  }
13 }
```

Listing 2.3: Handling duplicable state.

```
1  class Counter extends Replicable{
2   value : Number
3
4   constructor(){
5    this.value = 0
6   }
7
8   @mutating
9   inc(){
10   this.value += 1
11  }
12 }
```

Listing 2.4: Defining replicable state.

consumer using pass-by-copy semantics. In other words, the *consumer* receives a copy of the duplicate's state and methods. State updates are locally active: the assignment by the *consumer* actor on line 4 has no impact on the message created by the producer on line 10.

2.3.2.2 Replicas

Assume a different scenario for our producer-consumer example. The producer creates a numeric counter, which can be read and incremented by the producer and all consumers. To represent such remotely-active state, Triumvirate programmers extend the *Replicable* class.

Listing 2.4 contains the definition of the counter. The only noteworthy part of this definition is the @*mutating* annotation on line 8. This annotation informs the Triumvirate runtime that invocations of this method

```
1  class Consumer extends Actor{
2   consume(cnt : Counter){
3    cnt.inc()
4   }
5  }
6
7  class Producer extends Actor{
8   cnt : Counter
9
10  init(){
11   this.cnt = new Counter()
12  }
13  register(consumer : FarRef<Consumer>){
14   consumer.consume(this.cnt)
15   this.cnt.inc()
16  }
17 }
```

Listing 2.5: Handling replicable state.

mutate the replica's state. Subsequently, the Triumvirate runtime ensures that this update is applied to all other replicas.

Consider Listing 2.5. The *producer* actor instantiates a new counter (line 11) and sends this counter to all registering consumers (line 14). The counter is sent to the consumers using pass-by-replica semantics. In other words, a consumer receives a copy of the original object Moreover, this copy maintains a reference to the original counter object. The *producer* and all *consumers* are able to increment the counter concurrently (lines 15 and 3), Triumvirate ensures that eventually all counter replicas have the same value.

2.3.2.3 Derivations

Assume yet a different version of our producer-consumer setup. This time, the producer generates a stream of sensor readings that is read and transformed by consumers. The consumers automatically transform all new readings produced by the sensor. To represent such reactive state, Triumvirate programmers use the *Derivable* class.

In contrast to the other classes of distributed state one does not explicitly extend or instantiate objects from the *Derivable* class. Instead, a programmer creates a new derivation by applying a function to exist-

```
1  class Consumer extends Actor{
2   consume(stream){
3    this.libs.derive((streamValue)=>{
4     return transform(streamValue)
5    },stream)
6   }
7  }
8
9  class Producer extends Actor{
10  stream : Derivable
11
12  init(){
13   this.stream = this.libs.derive(_=>{
14    return readSensor()
15   },this.libs.seconds)
16  }
17  register(consumer : FarRef<Consumer>){
18   consumer.consume(this.stream)
19  }
20 }
```

Listing 2.6: Handling derivable state.

ing objects (i.e. duplicates, replicas or other derivations). We call these functions *derivation functions*.

For example, the producer creates the stream of sensor values by deriving from the built-in *seconds* object (line 13). Each change to the *seconds* object triggers a recomputation of the function provided to *derive*. In other words, the producer actor creates a derivation which updates every second by reading the latest sensor value (we assume the existence of a *readSensor* function on line 14).

The sensor stream derivation is sent to registering consumers using pass-by-derivation semantics. In other words, the consumer receives a new stream that depends on the original stream sent by the producer. Updates to the original stream automatically result in updates to this new stream. We discuss the consistency requirements and order of these updates in detail in Chapter 5.

Concretely, the consumer locally derives yet another stream on line 3 (we assume the existence of a *transform* function). Every second the *second* object is changed by the Triumvirate runtime which in turn triggers the producer to read the latest sensor value. Subsequently this triggers all registered consumers to transform the latest sensor value producer by the producer.

2.3.3 Triumvirate and the Rich Internet

Triumvirate integrates aspects of different distributed programming models (e.g. communicating event-loop actors, replicated data structures, reactive programming) into a coherent model for DRIAs. Concretely Triumvirate meets the requirements for a DRIA-oriented programming model as follows:

R_1: The model provides modular abstractions for distributed logic

Triumvirate programmers use CEL actors to implement their DRIA's distributed logic. These actors are modular and composable in typical object-oriented fashion. In other words, actor definitions can be extended and can implement interfaces.

Triumvirate actors are location agnostic: an actor definition can both represent a client or server side piece of logic. As such, Triumvirate supports vertical distribution of application logic. Moreover, given the inherent parallel and concurrent nature of actors these also allow for horizontal distribution on client and server-side.

Actors communicate with each other by invoking each other's methods and accessing each other's fields. These accesses and invocations are translated by the Triumvirate runtime to asynchronous message sends.

Triumvirate is the first implementation of this actor model in the context of web development. We discuss Triumvirate's implementation of the CEL actor model for the web in Chapter 6.

R_2: The model guarantees data consistency properties specified by the programmer

Triumvirate provides built-in support for locally-active, remotely-active and reactive state. Each kind of state is represented by its own class and exhibits a particular API. Moreover, programmers are freed from manually ensuring the consistency guarantees associated to each state or manually handling conflicting updates. Triumvirate ensures the guarantees of each kind of state, even in the face of concurrent and conflicting updates. We discuss the mechanisms employed by Triumvirate to ensure these guarantees for replicas in Chapter 4 and for derivations in Chapter 5.

R_3: **The model supports multiple parameter passing semantics**
Triumvirate provides three parameter passing semantics: pass-by-copy, pass-by-replica and pass-by-derivation. These three semantics suffice to implement the three major categories of distributed state in DRIAs: duplicable, replicable and derivable state. This frees the programmer from manually serialising and deserialising state. Instead, state is sent between actors as arguments to method invocations and Triumvirate automatically takes care of serialising and deserialising the state. Moreover, Triumvirate ensures that the bookkeeping information needed to ensure consistency guarantees of duplicates, replicas or derivations is kept up-to-date as they are sent across actors in the network. We discuss pass-by-replica and pass-by-derivation semantics in detail in Chapter 4 and Chapter 5 respectively.

R_4: **The model allows for extensible parameter passing and state update semantics**
Duplicables, replicables and derivables cover the three major categories of distributed state. However, programmers are able to extend and customise both the parameter passing semantics as well as the update semantics to fit application specific requirements. To this end, programmers override a number of hooks provided by Triumvirate's mirror-based meta programming layer (see Section 6.5).

Figure 2.3 provides an overview of Triumvirate's technology stack and how the different layers fulfil the aforementioned requirements. Concretely, Spiders.js implements the actors and meta layer exposed by Triumvirate to its programmers. As such, the Spiders.js layer of the Triumvirate technology stack fulfils requirements R_1 and R_4. Triumvirate uses these actors and meta layer to fulfil requirements R_2 and R_3.

2.4 Chapter Summary

This chapter introduces our running example: a fleet management system inspired by Emixis' production system. The system allows for dispatchers to interact with technicians in the field as well as track company vehicles. It exemplifies distributed rich internet applications, both its logic as well

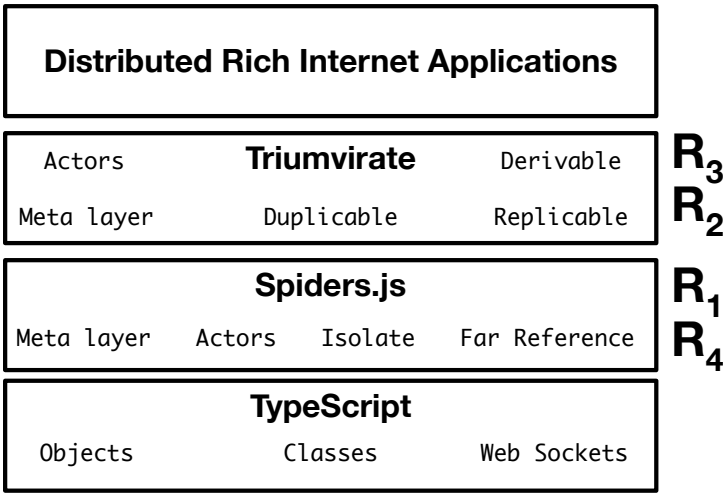


Figure 2.3: Overview of the Triumvirate technology stack and how it meets the requirements for a DRIA-oriented programming model.

as its state are distributed amongst multiple clients and servers. Moreover, different parts of the state require different update semantics.

We identify four key requirements which a distributed programming model should meet to ease the development of DRIAs. First, the model should provide a single abstraction to represent distributed logical components and should provide built-in communication primitives to coordinate these logical components. Second, the model should provide varying parameter passing semantics depending on the kind of data contained in these messages (e.g. pass-by-copy, pass-by-replication, etc). Third, the model should support various update semantics for distributed state (e.g. locally-active, remotely-active and reactive updates). Fourth, the model should allow the programmers to extend its built-in parameter passing and update semantics.

Using the aforementioned requirements we discuss and compare major distributed programming models and approaches. Although all models meet some requirements, none of them are able to fully support the complexity of DRIAs. This forces programmers to deal with complexity which is non-essential to their application domain (e.g. synchronising access to a piece of distributed state).

We introduce Triumvirate, an object-oriented domain-specific distributed programming language tailored towards the development of DRIAs. Triumvirate allows programmers to implement distributed logic using communicating event-loop actors. These actors allow for server-to-server, client-to-client and server-to-client distribution. Triumvirate provides three core classes to support distributed state: *Duplicable*, *Replicable* and *Derivable*. Each of these classes provides its own parameter passing and update semantics.

Chapter 3

State of the Art in Distributed Programming for Web Applications

This chapter uses the requirements discussed in Section 2.2 of Chapter 2 to discuss and compare the state of the art in distributed programming models. We indicate whether or not a specific model satisfies a particular requirement using a system of icons. Table 3.1 provides an explanation of these icons. It is important to note that the comparison in this chapter provides a high-level overview of work related to this dissertation. The following chapters provide more in-depth discussions of the related work for specific fields within distributed programming.

This chapter is divided in two parts. The first part discusses language-agnostic distributed programming models in general. The second part discusses distributed programming languages specifically designed for the web. More precisely, we focus on distributed programming languages that are able to support the peculiarities of web application architectures:

Icon	Explanation
✗	The model fails to meet the requirement
–	The model partially meets the requirement
✓	The model meets the requirement

Table 3.1: Requirement classification icons and their explanations.

- The amount of nodes in the network as well as their physical locations may change throughout the application's lifetime. For example, server nodes can dynamically be added to the network for the sake of scalability.
- Nodes intermittently lose connectivity with the network. This is especially the case for clients running web applications on their mobile devices.
- The address of nodes is not globally known across the network. In other words, not all nodes in the network are able to communicate with each other. For example, a single server node might delegate clients' requests to multiple other server nodes that are hidden to the client.
- Code is dynamically deployable to nodes in the network. For example, client nodes receive and evaluate their source code by connecting to server nodes.

3.1 Distributed Programming Models

To the best of our knowledge no distributed programming model has thus far specifically been designed for web applications. In this section we therefore discuss distributed programming models in general and compare them with regards to the requirements for DRIA-oriented programming models.

3.1.1 Remote Procedure Calls and Method Invocations

There are three generations of the RPC model. The original remote procedure call (RPC) model [BN84] offered synchronous request-response interactions between nodes in a network. In a nutshell, nodes interact with one another by remotely calling each other's procedures. Calling a remote procedure blocked the execution of the caller's program until the remote procedure returned. The idea behind this design choice was to make local and remote procedure application indistinguishable from each other. This design choice has been heavily critiqued for its many flaws (e.g. lack of fault-tolerance, latency and distribution hiding, etc.).

The second generation of RPC is an adaptation to the object-oriented paradigm. This generation of the model is called *remote method invocations* (RMI) (e.g. Java RMI [Mic98], CORBA [Gro12]). For the most parts these versions were direct adaptations of the original model. In other words, nodes in the network now contain a heap of remotely accessible objects. The nodes' stubs are no longer placeholders for remote procedures, but expose these different objects. Nodes are able to invoke methods on these remote objects, through the stubs, as if these were local. However, RMI inherited the flaws for which RPC is critiqued [KWWW94].

A third generation employs the RPC model in the context of web development. Initial attempts suffered from the same flaws as the original model (e.g. XML-RPC [Win99], SOAP [BEK+00]). However, newer implementations (e.g. Google's gRPC [gRP06], Facebook's Thrift [SAK07]) solve a number of these issues (e.g. asynchronous calls, failure handling, distribution hiding, etc.).

We argue that the RPC model fails to adequately deal with all the complexities of DRIAs:

R_1 **: The model provides modular abstractions for distributed logic** ▬

In the RPC model distributed logical components are implemented as sets of remotely callable procedures. This abstraction allows for both horizontal (i.e. multiple servers) and vertical (i.e. client/server) distribution. Moreover, programmers coordinate these components using a single built-in communication primitive: remote procedure calls. However, these sets of remotely callable procedures lack modularity. The RPC model does not provide the mechanisms needed to easily compose and deploy combinations of these sets.

R_2**: The model guarantees data consistency properties specified by the programmer** ▬

In general, the RPC model lacks distributed and shared state and only provides locally-active update semantics. An argument to a remote procedure invocation can be mutated, but this mutation only affects that particular copy. An exception to this is the Dexter [TG11] framework that allows programmers to implement custom state update semantics

R_3: The model supports multiple parameter passing semantics -

RPC/RMI implementations either only support pass-by-copy semantics or pass-by-reference and pass-by-copy semantics (e.g. Dexter [TG11]). To the best of our knowledge current RPC implementations lack the semantics to support the bookkeeping of reactive state.

R_4: The model allows for extensible parameter passing and state update semantics -

Most RPC implementations delegate serialisation and deserialisation of arguments to so-called *stubs*. Given an open [KLL+97] RPC implementation, as is the case for Dexter [TG11], these stubs provide programmers with a single hook to override how data is distributed across the network. However, these stubs do not provide the abstractions to override how objects are mutated and updated. As such, the RPC model does not allow programmers to implement custom update semantics.

3.1.2 Tuple Spaces

In the tuple space model [Gel85], nodes in the network conceptually share a data structure called the *tuple space*. This data structure contains tuples, which are ordered groups of values. A tuple space supports three operations: *out* inserts a tuple into the tuple space, *in* removes a tuple from the tuple space and *rd* reads a tuple from the tuple space without removing it. Moreover, programmers are often given the ability to install callbacks that are invoked as soon as a tuple of a specified pattern is inserted in the tuple space.

In the original model, introduced by the coordination language Linda [Gel85], the nodes in the network maintain the single shared tuple space. Maintaining a central tuple space consistent is often impossible, for example in scenarios where partial failures or disconnections of nodes occur often. As such, a number of approaches [MPR01, MZ04, GSMD14] allow nodes in the network to maintain their own tuple space. Consequently, nodes synchronise their local tuple spaces after each *in* operation in order to keep the conceptual global tuple space consistent. Other mod-

els [MZ04, GSMD14] take the opposite approach: they replicate individual tuples across the network.

The tuple space model meets the requirements for DRIA-oriented programming models as follows:

R_1: The model provides modular abstractions for distributed logic ✓

Tuple spaces abstract application logic as sets of reactions to particular events. These reactions form an abstraction for programmers to specify the distributed logic of a specific node in the network. Moreover, tuple spaces provide a set of communication primitives (i.e. *in*, *out* and *rd*) to coordinate these distributed pieces of logic. The tuple space model, or at least some of its implementations [MPR01, GSMD14], therefore satisfies R_1 of DRIA-oriented programming models.

R_2: The model guarantees data consistency properties specified by the programmer ▬

Some tuple space implementations (e.g. TOTAM [GSMD14], Linda [Gel85]) model tuples as immutable data structures. The tuples in these implementations are therefore trivially consistent, but programmers lack the ability to leverage the tuples' consistency and availability (i.e. as given by the CAP [Bre00] theorem). Other models [DFWB98] favour the availability of tuples by allowing all nodes in the network to read and update tuples. However, updating a tuple during a network partition is restricted to the node that created the tuple.

Finally, some tuple space implementations allow programmers to apply operations with varying degrees of consistency on the tuple space [ADNL15]. This allows programmers to implement remotely-active state. However, to the best of our knowledge no tuple space implementation supports reactive state.

R_3: The model supports multiple parameter passing semantics ▬

Tuple space implementation in which tuples are immutable data structures typically only support pass-by-copy semantics. Other tuple space implementations [DFWB98, ADNL15] support pass-by-replica or even pass-by-far-reference semantics [GSMD14]. However,

current tuple space models and implementations lack support for pass-by-derivable parameter passing semantics.

R_4: The model allows for extensible parameter passing and state update semantics ▬

Some implementations of the tuple space model (e.g. TOTAM [GSMD14] and TOTA [MZ04]) allow programmers to override how tuples are replicated across the network through so-called propagation protocols. These protocols provide programmers with a number of hooks into the tuple's replication process. However, these hooks solely focus on the replication process of tuples. As such, programmers are only able to override parameter passing semantics and are therefore unable to implement custom update semantics.

3.1.3 Actors

Actors are autonomous units of computation. In other words, actors execute independently of each other (e.g. each actor has a dedicated thread) and do not share memory. Since the original actor model, introduced by Hewitt et al. in 1973 [HBS73], a range of different actor models have been proposed [DKVCDM16]. In our discussion we focus on the *communicating event-loops* (CEL) [MTS05] actor model. This model and its implementations most closely meet the requirements for a DRIA-oriented programming model, given that it lies at the foundation of most JavaScript engines.

We discuss the CEL model in Section 2.3.1.1 of Chapter 2. In a nutshell, CEL actors contain heaps of objects. Two objects residing in the same heap hold local references to each other and can synchronously invoke each other's methods. Two objects residing in different heaps hold far references to each other and asynchronously invoke each other's methods. Moreover, objects can be sent between actors as part of method invocations. The CEL model discriminates between objects that cross the actor boundaries *by copy* and those that cross the actor boundaries *by far reference*.

R_1: The model provides modular abstractions for distributed logic ✓

Actors inherently are modular abstractions for distributed logic. Each actor specifies a piece of distributed logic and coordinates with

other actors through asynchronous messages. As such, the actor model fully meets R_1 of DRIA-oriented programming models.

R_2: The model guarantees data consistency properties specified by the programmer ✗

The actor model only supports locally-active state updates. Even in the CEL model, updates only affect the target object. This is regardless of whether the invoker of the update has a local or far reference to said object.

R_3: The model supports multiple parameter passing semantics ✗

The CEL variant of the actor model provides two built-in parameter passing semantics: pass-by-copy and pass-by-far-reference. In other word, programmers are able to express duplicable and some form of replicable state. For example, replicable state which must be offline available cannot be expressed with pass-by-far-reference semantics. In other words, these two semantics do not suffice to express all forms of replicable or duplicable state.

R_4: The model allows for extensible parameter passing and state update semantics ✗

A number of actor-based programming languages offer metaprogramming abstractions [MVCT+09, McA95, WY14, OIT92]. Some of these abstractions fulfil R_4 (e.g. AmbientTalk's mirror-based reflection layer [MVCT+09]). However, these abstractions are implementation dependant and are not inherently part of the actor model, which does not allow programmers to extend built-in parameter passing or state update semantics.

3.1.4 Replicated Data Types

The database community has extensively researched data replication and data consistency. This research focuses primarily on novel consistency models and algorithms to enforce certain levels of consistency in replicated databases. In recent years data replication has also been researched from within the programming language community. In contrast to the database community, this research focuses on language abstractions for data replication. In general the programming-language research on data

replication tackles the following problems: First, how can programmers specify that a piece of state should be replicated amongst two or more nodes in a network? More specifically, how do programmers specify that updates to this state should be remotely active (i.e. an update to one replica triggers an update to all others)? Second, how do programmers specify the data consistency guarantees required for this state? For example, how does one specify that replicas are allowed to temporarily be in different states? We shortly discuss the most prevalent approaches. Chapter 4 discusses the state-of-the-art in abstractions for data replication in more detail.

The programming language research on data replication follow two approaches. First, mostly theoretical approaches [SPBZ11, BLPF15] which focus on researching new kinds of data consistency (e.g. strong eventual consistency) and how these can be enforced for particular data types. A well known example of this research are *conflict-free replicated datatypes* (CRDTs) [SPBZ11]. It constraints the operations on an abstract data type (ADT) to be commutative, associative and idempotent. If these constraints are met, it guarantees that all replicas of this ADT remain strongly eventually consistent in the face of concurrent updates.

A second approach focuses on integrating these data types and consistency models into distributed programming languages. For example, Lasp [MVR15] is a programming language where all data types are CRDTs. Lasp allows programmers to develop distributed systems as collections of processes (i.e. Erlang actors). The state of these systems is represented as CRDTs instances and compositions thereof, which are shared amongst the processes. Composition of CRDTs instances happens through a functional programming API (i.e. *map*, *filter*, *fold*).

In general, the models, frameworks and languages tailored towards replicated data types meet the requirements for DRIA-oriented programming models as follows:

R_1**: The model provides modular abstractions for distributed logic** ✗

Data replication approaches allow programmers to implement distributed logic by means of abstract replicated data types. These abstractions typically specify a data type's consistency in the face of various operations. However, we argue that data replication approaches lack modularity. With the exception of Lasp [MVR15] pro-

grammers are unable to create new abstract replicated data types by arbitrarily combining existing ones.

R_2: The model guarantees data consistency properties specified by the programmer ▬

Data replication approaches inherently provide a myriad of different update semantics, depending on the required consistency guarantees. However, these different semantics only support locally and remotely-active updates. To the best of our knowledge these approaches do not provide solutions for reactive updates.

R_3: The model supports multiple parameter passing semantics ▬

Typically, data replication approaches either assume data to already be replicated across the network (i.e. the actual replication is not part of the model) or solely provide pass-by-replication semantics. As such, these approaches only partially fulfil R_3 for DRIA-oriented programming models.

R_4: The model allows for extensible parameter passing and state update semantics ✗

A number of replicated data approaches [LLC+14, HBZ+16, SKJ15] allow programmers to specify application-specific constraints on distributed state. For example, a constraint that specifies the maximum value of a bounded counter. These approaches then determine the optimal consistency level while upholding these constraints. However, these approaches do not allow programmers to override how data type are updated or how data types are sent across the network.

3.1.5 Distributed Reactive Programming

The reactive programming paradigm [BCC+13] is specifically designed for the development of event-driven applications. It revolves around three core concepts. First, time-varying values (e.g. the user's mouse position) are first-class language constructs commonly called *signals*. Second, programmers are able to create new signals out of existing ones using signal combinators (i.e. lifted functions). Third, the language runtime tracks the dependencies between the various signals that form a graph. Changes to

source signals propagate through the dependency graph, thereby updating the application's state.

Distributed reactive programming (DRP) [MSDM19, DSMM14, SW18] applies the core principles of reactive programming in a distributed setting. Signals (and therefore the dependency graph) can reside on physically distributed machines. Changes propagate through this dependency graph through nodes sending messages to each other.

From a programmer's perspective the distributed reactive programming model remains largely unchanged compared to its non-distributed counterpart. Programmers implement their application's logic by applying lifted functions to signals, regardless of the signals' whereabouts. The underlying runtime ensures that the dependency graph updates as the system receives new data (e.g. clients issuing requests). In Chapter 5 we discuss the state-of-the-art in reactive runtimes and propagation algorithms in detail.

In general, distributed reactive approaches meet the requirements for DRIA-oriented programming models as follows:

R_1: **The model provides modular abstractions for distributed logic** ✓

DRP programmers declaratively specify how parts of their applications depend on each other by applying lifted functions to signals. These signals represent the application's state while lifted functions contain the application's logic. The DRP runtime ensures that changes to the system's state (i.e. to signals) automatically trigger the right parts of the application's logic (i.e. be re-applying lifted functions). Moreover, this propagation of change happens regardless of a signal's whereabouts. DRP therefore provides modular abstractions for distributed logic.

R_2: **The model guarantees data consistency properties specified by the programmer** ✗

Using DRP, programmers are trivially able to express reactive state. Moreover, REScala programmers [MBS+18] are able to implement remotely-active state. Two replicated signals are allowed to temporarily diverge in state and are guaranteed to eventually be consistent through REScala's use of *conflict-free replicated data types* (CRDTs) [SPBZ11]. As such, REScala supports a single flavour of remotely-active state.

R_3: The model supports multiple parameter passing semantics ▬

DRP supports derivable state through signals and provides pass-by-derivation semantics. In other words, whenever a node sends a signal to another node this implicitly creates an edge in the distributed dependency graph. Whenever the sender's signal changes with a new value, this value is sent along the edge using pass-by-copy semantics to update the receiver's signal. REScala is an exception, as it allows signals to be passed by replication amongst nodes in the network for the sake of fault tolerance and offline availability.

R_4: The model allows for extensible parameter passing and state update semantics ✗

To the best of our knowledge, there exists no DRP solutions that allows programmers to extend its built-in semantics. Although approaches to extend the semantics of non-distributed reactive programming have been proposed [WS17], these are yet to be extended to DRP.

3.2 Programming Languages for the Web

A number of programming languages have specifically been designed for the web. These languages radically differ in their programming models and in the nature of the abstractions they provide. We therefore discuss the most prominent of these languages separately. More specifically, this section targets distributed programming languages that fulfil two requirements. First, the language must allow programmers to implement both the server and the client-side of their web application. Second, the language provides built-in abstractions tailored towards distribution.

3.2.1 OPA

OPA [BKS13] is a tierless programming language specifically designed for the web. Web architectures commonly comprise three tiers: a UI, application logic and database tier. In traditional web programming each of these tiers is implemented separately, possibly using different technologies or languages. A tierless programming language allows programmers to

implement each tier as if they were part of the same program. A tier splitter determines which parts of the code will run in which tier.

The basic building blocks of OPA are functions that can be composed into modules. Moreover, OPA provides a number of communication primitives which allow client (i.e. the UI tier) and server (i.e. the application logic tier) functions to coordinate. Programmers can optionally annotate these functions and modules to specify in which tier the specific function should run. Otherwise, OPA comes with a built-in slicing mechanism which determines where a function should run and how data should be duplicated.

OPA meets our requirements for DRIA-oriented programming as follows:

R_1: **The model provides modular abstractions for distributed logic** ✓

OPA's abstraction for pieces of distributed logic are functions that can be remotely called. Moreover, OPA offers three communication primitives to coordinate the execution of database, server and client functions: *Session*, *Cell* and *Network*. The first offers one-way asynchronous communication, the second two-way synchronous communication and the last serves as a broadcasting mechanism.

R_2: **The model guarantees data consistency properties specified by the programmer** ✗

OPA only provides immutable data structures and therefore does not provide any update semantics or data consistency guarantees.

R_3: **The model supports multiple parameter passing semantics** ✗

OPA provides two parameter passing semantics. Native OPA values (i.e. integers, floats, strings and records) adhere to pass-by-copy semantics. OPA implements a custom parameter passing semantic for functions. A function can be sent from one node to another in the network. The receiving node constructs a stub for the original function. This stub ensures that all calls to the received function are translated into remote procedure calls to the node owning the original function.

R_4: The model allows for extensible parameter passing and state update semantics ✗

OPA treats data from other languages and systems as *external*. Programmers are able to explicitly implement serialisers and deserialisers for these external types. However, this mechanism is too crude to implement custom parameter passing or update semantics for OPA data types. The serialisers and deserialisers solely allow programmers to determine how data should be copied between nodes.

3.2.2 Hop.js

Hop.js [SP16] is the runtime for a language called HopScript that is a superset of JavaScript. As is the case for OPA, HopScript is a tierless programming language. Programmers implement both the server as well as the client of their web application using HopScript. Moreover, HopScript provides two tier-switching operators `~` and `$` that allow programmers to manually delimit client-side from server-side code. We discuss the rest of HopScript's particularities as we evaluate it against the requirements for a DRIA-oriented programming model.

R_1: The model provides modular abstractions for distributed logic ✓

HopScript models distributed logic as *services*, which are comparable to remote procedures. In its simplest form, HopScript servers expose a number of services which can be called by clients. Moreover, clients are able to import these services and call them as if they were local. As such, services allow for vertical distribution and horizontal distribution on the server. Programmers coordinate the parts of their web applications by using service applications.

R_2: The model guarantees data consistency properties specified by the programmer ▬

HopScript supports reactive and locally-active state. Programmers declare certain objects or events – called sources – as time-varying through the *reactProxy* method. All DOM elements which depend on a source will automatically be updated as soon as said source changes. However, this reactive state only serves to update a client's UI upon local state changes. In other words, changes to reactive state do not propagate between different nodes in the network.

R_3: **The model supports multiple parameter passing semantics** ✗

HopScript only supports pass-by-copy semantics, as it essentially implements an improved version of JavaScript's standard JSON serialisation.

R_4: **The model allows for extensible parameter passing and state update semantics** ✗

Although HopScript partially supports EcmaScript 6, it does not support its reflexive capabilities. To the best of our knowledge, HopScript does not provide any means for the programmer to extend its built-in semantics, particularly regarding parameter passing and update semantics.

3.2.3 Links

Links [CLWY06] is a tierless programming language for web applications. It differs from other languages by compiling parts of its syntax to SQL queries. This allows programmers to implement the database tier of their applications in the same language as the client and server tier. Links programmers implement their applications as a series of functions that they annotate with either *client* or *server*. Client functions are callable from the server and vice versa. In general, links meets the requirements for DRIA-oriented programming models as follows:

R_1: **The model provides modular abstractions for distributed logic** ✓

Links models distributed logic as functions and provides communication primitives in the form of remote function calls. However, Links programmers are unable to lexically determine whether a function call happens locally or remotely. As such it suffers from the same design flaws as the traditional RPC model [KWWW94].

R_2: **The model guarantees data consistency properties specified by the programmer** ✗

Links only provides immutable data structures and therefore does not provide any update semantics or data consistency guarantees.

R_3: **The model supports multiple parameter passing semantics** ✗

When a remote function call occurs, all arguments are sent by-copy to the remote location. As such, Links only supports pass-by-copy semantics.

R_4: **The model allows for extensible parameter passing and state update semantics** ✗

To the best of our knowledge, Links does not provide any mechanism to extend its built-in semantics.

3.2.4 Stip.js

Stip.js [PDRVCDM14] strives to bring the benefits of tierless programming languages to JavaScript. It is essentially a tier-splitting tool that allows JavaScript programmers to decide on which tier parts of their code are to be deployed. Although Stip.js does not technically qualify as a distributed programming language it does enhance JavaScript with a number of distributed functionalities. More concretely, Stip.js satisfies the requirements for a DRIA-oriented programming language as follows:

R_1: **The model provides modular abstractions for distributed logic** ▬

Stip.js does not provide any new abstractions over regular JavaScript with regards to application logic. We therefore restrict our discussion to JavaScript's support for modular distributed logic abstractions.

Node.js, the most popular server-side implementation of JavaScript allows for rudimentary horizontal distribution using *child processes*. Most client-side implementation also support horizontal distribution in the form of *web workers*. Both abstractions are actor-like constructs which allow programmers to execute application logic in parallel. However, these abstractions do not allow programmers to distribute logic vertically. Moreover, the abstractions differ in their API and semantics.

R_2: **The model guarantees data consistency properties specified by the programmer** ▬

Stip.js provides annotations that allow programmers to specify the

	R_1	R_2	R_3	R_4
RPC	–	–	–	–
Tuple Spaces	✓	–	–	–
Actors	✓	✗	✗	✗
Replicated Data Types	✗	–	–	✗
Distributed Reactive Programming	✓	✗	–	✗
OPA	✓	✗	✗	✗
Hop.js	✓	–	✗	✗
Links	✓	✗	✗	✗
Stip.js	–	–	–	✗

Table 3.2: State of the art in distributed programming for web applications compared to the requirements for a DRIA-oriented programming model.

consistency guarantees required for specific variables [PDMDR15]. After tier splitting Stip.js ensures that these guarantees are upheld in the face of concurrent updates. However, Stip.js only supports remotely-active state and does not provide annotations for reactive state.

R_3: The model supports multiple parameter passing semantics –

Stip.js automatically determines when to use pass-by-replica or pass-by-far-reference semantics (e.g. through the programmer's annotations). However, Stip.js does not support pass-by-derivation semantics.

R_4: The model allows for extensible parameter passing and state update semantics ✗

To the best of our knowledge, Stip.js does not provide any mechanism to extend its built-in semantics.

3.3 Overview of the State of the Art

Table 3.2 summarises how the state of the art in distributed programming meets the requirements for a DRIA-oriented programming model. Most approaches at least partially fulfil these requirements. However, no approach is able to provide an elegant solution for *all* requirements.

Triumvirate strives to integrate the best parts of the aforementioned approaches into a coherent model. Concretely, Triumvirate represents distributed logic using CEL actors (see Chapter 6). These actors are able to share three kinds of distributed state: duplicable, replicable and derivable state. Replicable state is heavily influenced by data replication approaches (see Chapter 4). Triumvirate offers multiple kinds of replicable state, each with its own consistency guarantees. Derivable state is influenced by distributed reactive programming and allows programmers to implement reactive state (see Chapter 5). Moreover, programmers are able to extend the built-in update and parameter passing semantics of duplicable, replicable and derivable state using Triumvirate's metaprogramming abstractions (see Section 6.5 of Chapter 6).

Triumvirate is greater than the sum of its parts. A large part of our work is dedicated to defining how these parts interact with one another and how to maintain their respective guarantees within a single programming model. In the following chapters we show how Triumvirate integrates actors, replicated data types and reactive programming into one coherent DRIA-oriented programming model.

Chapter 4

Replicating Remotely-Active State

DRIAs distribute data across multiple servers and/or clients for the sake of offline availability, performance, security, etc. These data conceptually represents shared state and should therefore provide remotely-active update semantics: If one client changes the shared state locally, this change should become visible to all other clients as well. In our running example (see Chapter 2) technicians are equipped with smartphones installed with a fleet management application. On one hand this mobile application allows technicians to access their job schedule and edit information with regards to a particular job. On the other hand the application provides dispatchers with a general overview of all jobs and allows them to reschedule particular jobs. The dispatcher and technician share two pieces of distributed state: the list of jobs to be done and the job schedule. Distributing these different parts of the application's state that have varying consistency requirements heavily burdens the developer.

In general, distributing state forces the developer to think about *Consistency*, *Availability* and *Partition tolerance* as captured by the CAP theorem [Bre00, GL02]. This theorem states that it is impossible for a distributed system to simultaneously guarantee all three. DRIAs must be partition tolerant as clients disconnect frequently and network failures are recurrent. Programmers are therefore left with a trade-off guaranteeing either availability or consistency for different parts of their web application. For example, a technician is always able to modify data regarding

a particular job (i.e. job-specific functionality is available). However, this entails that the dispatcher and the technician might have diverging information regarding this job (i.e. job-specific functionality is not consistent). This is due to the fact that the technician might modify the data while being offline. Conversely, accessing or modifying the job schedule must happen consistently (i.e. the dispatcher and a technician may never have a diverging view over the job schedule). This entails that schedule-specific functionality is not always available.

Most distributed programming languages and libraries implicitly make this trade-off for the programmer. For example, using E's [MTS05] *eventual references* one implements consistent and partition-tolerant (*CP*) systems. On the other side of the spectrum, Lasp [MVR15] exclusively relies on *conflict-free replicated data types* (CRDTs) [SPBZ11] for distribution which makes it suitable to implement available and partition-tolerant (*AP*) systems. Unfortunately, many applications cannot be categorized as fully *AP* or *CP*. Programmers faced with such mixed *AP*-*CP* applications cannot rely on the high-level abstractions offered by a *single* distributed programming language. Instead they are forced to resort to low level APIs or external libraries to guarantee *either* their system's consistency or availability.

This chapter presents *Replicables*, a class of distributed objects that allows programmers to easily implement the *AP* and *CP* aspects of their distributed systems. *Replicables* come in two flavours: *eventual replicables* and *strong replicables*. Concrete instances of these replicables are called eventual and strong *replicas*. On one hand eventual replicas allow the programmer to implement the *AP* functionalities of their system. The Triumvirate runtime ensures that availability of eventual replicas is guaranteed even in the face of partitions. Moreover, eventual replicas are kept eventually consistent across actors. This unburdens the programmer from manually synchronising diverging state of eventual replicas after a partition heals. On the other hand strong replicas allow the programmer to implement the *CP* functionalities of their system. The Triumvirate runtime guarantees that operations on strong replicas are consistent even in the face of partitions. This comes at the price of these operations not always being available.

Figure 4.1 highlights the concepts discussed in this chapter as well as the requirements that replicables tackle. More precisely, in this chapter we

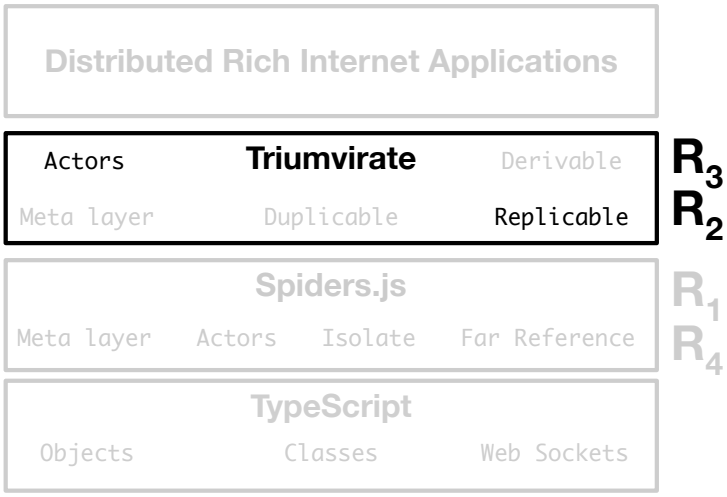


Figure 4.1: Concepts discussed in this chapter.

detail how replicables partially fulfil requirements R_2 and R_3 for DRIA-oriented programming models. In other words, we describe how replicables *guarantee data consistency properties specified by the programmer* and how they provide *support for multiple parameter passing semantics*.

We start by giving an overview of how programmers use replicables in their Triumvirate applications after which we discuss the performance differences between strong and eventual replicas. Subsequently we discuss the implementation of an interactive presentation tool built using replicables that has been used in a real-life situation. Finally we end this chapter by detailing how eventual and strong replicas are implemented in the Triumvirate runtime.

We largely base this chapter on our work presented in [MSDM18a].

4.1 Strong versus Eventual Replication: a Functional Choice

Triumvirate developers implement the remotely-active parts of their applications' state using replicables. Programmers choose between two concrete variants of replicables: eventual or strong. This decision is in part driven by functional considerations, as eventual and strong replicables differ in the guarantees they provide and in the APIs they offer. We start by

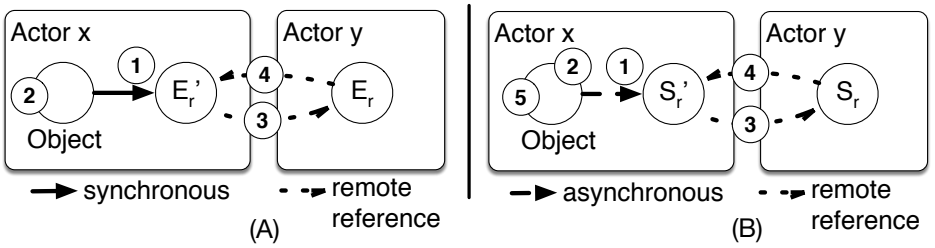


Figure 4.2: (A) Method invocation on an eventual replica. (B) Method invocation on a strong replica.

giving a high-level overview of the programmatic differences between both replicables. Subsequently we implement parts of our running example to showcase their use and expressiveness.

4.1.1 Strong and Eventual Replicas

Eventual replicas implement the *AP* functionality in a distributed application. Figure 4.2 (A) gives a conceptual overview of how they work. An actor x has acquired an eventual replica (marked E'_r). Conceptually, a single actor in the network (i.e. actor y) owns the *master* (marked E_r) eventual replica. An object in x synchronously invokes a method of E'_r (1), the method executes locally on E'_r and returns a value (2). Subsequently, E'_r sends its new local state to its master (3) which merges the state with its own and returns the new global state (4). This mechanisms ensures the eventual consistency of E'_r and E_r's state, which we define as follows:

Definition 4: Eventual consistency

A master replica and its worker replicas are eventually consistent if all of the replicas' fields contain equal values when the system reaches quiescence. A system of master and worker replicas reaches quiescence when no more methods are invoked on replicas and when all worker replicas have received the last returned global state from the master replica.

Eventual replicas guarantee three properties:

Guarantees of eventual replicas

Eventual replicas are ***available*** (see Definition 2). In other words, reading the value of an eventual replica's field or invoking one of its method is guaranteed to always return a meaningful result.
Eventual replicas are ***eventually consistent*** (see Definition 4). In other words, simultaneous reads and method invocations on two replicas might temporarily return diverging results. However, these reads and method invocations return equal results when the system reaches quiescence.
Eventual replicas are ***partition tolerant*** (see Definition 3). The replicas remain available under network partitions and their eventual consistency is guaranteed upon healing of said partitions.

Strong replicas implement the *CP* functionality in a distributed application. Figure 4.2 (B) gives a conceptual overview of how they work. An actor x has acquired a strong replica (marked S'_r). As is the case for eventual replicas, a single actor in the network (i.e. actor y) owns the *master* (marked S_r) replica for all instances of a strong replicable. An object in x asynchronously invokes one of S'_r's methods (1) which returns a promise (2). Subsequently, S'_r sends the invocation (i.e. the method's name and arguments) to its master (3) which locally performs the invocation and sends back the return value (4). Finally S'_r resolves the promise (5) returned in step 2 with the return value received from S_r in step 4.

Guarantees of strong replicas

Strong replicas are ***consistent*** (see Definition 1). In other words, simultaneous reads on two strong replicas either return equal results or no result at all (e.g. because of network partitions).
Strong replicas are ***partition tolerant*** (see Definition 3). The consistency of operations on strong replicas are guaranteed in the face of partitions at the cost of availability (see Definition 2).

Eventual and strong replicas both ensure partition tolerance. The major difference between both kinds of replicas are their consistency and availability guarantees. The state of an eventual replica can always be changed or read, Triumvirate ensures that eventually all state changes

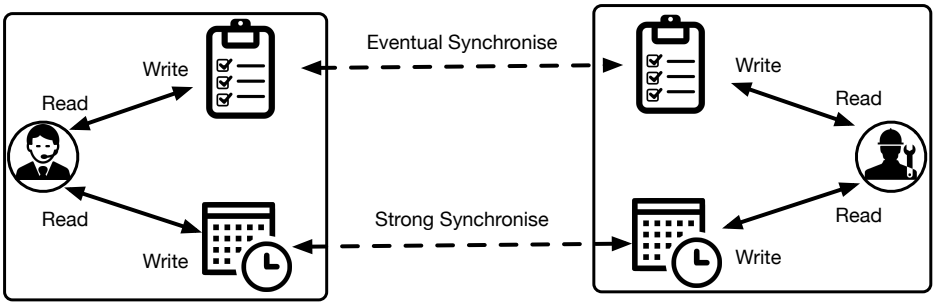


Figure 4.3: Eventual and strong replicas in the fleet management system.

propagate to other replicas. In contrast, the state of a strong replica can only be changed or read if strong consistency amongst all replicas is guaranteed.

4.1.2 Replicas in Practice

Figure 4.3 gives an overview of the parts of the fleet management system that are relevant to this chapter. Concretely, the dispatchers and technicians in our system share two replicas: the list of jobs and the job schedule. The former is an eventual replica: technicians and dispatchers are *always* able to create new jobs, update a job or consult the list of jobs. As a result, the list of jobs might temporarily diverge across technicians and dispatchers (e.g. during network partitions). The latter is a strong replica: a technician and dispatcher simultaneously consulting the job schedule should always see the same schedule. As a result, reading from or writing to the job schedule is not always possible (e.g. during network partitions).

In the following sections we show how to implement this part of the fleet management system using Triumvirate's replicable state.

4.1.2.1 Available State as Eventual Replicas

Our running example requires the functionality related to the technicians' jobs (e.g. updating a job's decription) to be offline available. As such, job-related functionality represents the AP part of our fleet management system. We implement this part of our system using two eventual replicas: one which maintains the state for individual jobs and one which maps

```
class Job extends EventualReplicable{
 id : number
 description : string
 pictures : Array<Buffer>

 constructor(id:number, description:string){
  this.id = id
  this.description = description
  this.pictures = []
 }

 @mutating
 addPicture(picture:Buffer){
  this.pictures.push(picture)
 }

 @mutating
 updateDescription(newDescription:string){
  this.description = newDescription
 }
}
```

Listing 4.1: Defining Individual jobs.

technicians onto jobs. Listing 4.1 provides the definition of the former. *Job* replicas represents the state associated to a single job: a text description of the job and an array of associated pictures. Moreover, these replicas provide two methods: one to add a picture and one to update the job description.

Listing 4.2 provides the definition of the *JobList* replicas. These replicas represent the collection of job-related states in the entire system. In other words, they keeps track of all jobs still to be performed for clients. Dispatchers add jobs to a list through the mutating *addJob* method. Technicians remove jobs (i.e. upon completion) using the *remJob* method.

The functionality provided by *Job* and *JobList* is available by design. Technicians and dispatchers are always able to create new jobs or update the information on a specific job, even in the face of network partitions. However, this entails that the state of specific jobs or the job list might diverge across technicians and dispatchers. Triumvirate ensures that possible divergence between these states are eventually resolved. In other words, Triumvirate ensures the eventual consistency of jobs and the job list. All job and job list replicas will eventually be in the same state, provided that technicians and dispatchers stop issuing updates. Triumvirate

```
1  class JobList extends EventualReplicable{
2   jobs : Map<number,Job>
3
4   constructor(){
5    this.jobs = new Map()
6   }
7
8   @mutating
9   addJob(job:Job){
10   this.jobs.set(job.id,job)
11  }
12
13  @mutating
14  remJob(jobId:number){
15   this.jobs.delete(jobId)
16  }
17 }
```

Listing 4.2: Defining the collection of jobs.

programmers are therefore freed from manually maintaining eventually consistent replicated state.

4.1.2.2 Consistent State as Strong Replicas

Our running example requires the job schedule to be consistent. A technician and a dispatcher should never be able to simultaneously see two different versions of the technician's schedule. As such, the job schedule represents the *CP* part of our running example. Hence, we implement this part using a *strong replica*.

Listing 4.3 gives the definition of *JobSchedule* replicas. Schedules maintain a dictionary which maps technician names onto an ordered list of job ids and provide two main methods. First, *reschedule* either adds a new job to the schedule or moves an existing job in the schedule. In the latter case *reschedule* relies on an auxiliary function *removeJob*. This method will typically be invoked by dispatchers. Second, the *nextJob* method returns the next scheduled job for a particular technician. This method will typically be invoked by technicians.

The functionality provided by *JobSchedule* is consistent by design. Both dispatchers and technicians are able to invoke *reSchedule* and *nextJob*. However, Triumvirate will only perform these invocations if strong

```
class JobSchedule extends StrongReplicable{
 schedule : Map<string,Array<number>>

 removeJob(toRemoveId : number){
  this.schedule.forEach((jobs:Array<number>, technician:string)=>{
   if(jobs.contains(toRemoveId)){
    schedule.set(technician,jobs.filter(jobId => jobId !=
    toRemoveId))
   }
  })
 }

 @mutating
 reSchedule(jobId:number, technician:string, position:number){
  removeJob(jobId)
  this.schedule.get(technician).splice(position,0,jobId)
 }

 nextJob(technician:string){
  this.schedule.get(technician)[0]
 }
}
```

Listing 4.3: Defining the job schedule.

consistency is guaranteed. Triumvirate automatically buffers method invocations on strong replicas if strong consistency cannot be guaranteed.

Triumvirate guarantees that a technician and a dispatcher are never able to read a different schedule for the same job. Consequently, the functionality provided by the job schedule is not always available (i.e. one cannot invoke methods or reads the schedule's state during network partitions).

4.1.2.3 Replicating the Data and Reacting to Change

We represent technicians and dispatchers by means of Triumvirate actors. Listing 4.4 provides the definition of the *Dispatcher* actor. This actor's task is twofold: to disseminate all job-related state and to create new jobs or reschedule existing ones. The former task is accomplished in the actor's *init* method and uses Triumvirate's built-in topic-based publish-subscribe system. In a nutshell, the Triumvirate standard library (available to all actors as *this.libs*) provides two methods. *publish* takes a topic and an object and publishes said object on the network. *subscribe* takes a topic and allows programmers to install callbacks that are invoked whenever

```
1  class Dispatcher extends Actor{
2   schedule : JobSchedule
3   jobList : JobList
4
5   init(){
6    let topic = new this.libs.PubSubTag("jobData")
7    this.schedule = new JobSchedule()
8    this.jobList = new JobList()
9    this.libs.publish(topic,[this.jobList,this.schedule])
10  }
11 }
```

Listing 4.4: Defining the dispatcher actor.

an object is published under the subscribed topic. In our example the dispatcher publishes the *JobList* and *JobSchedule* replicas (line 9). The creation of new jobs or the rescheduling of existing ones is done by accessing the dispatcher's instance variables. We omit the user-interface code responsible for this as it does not contribute to this discussion.

Listing 4.5 provides the definition of the *Technician* actor. The technician starts by subscribing to the data published by the dispatcher (line 10). To subscribe, the actors provides a callback that is invoked with an eventual replica of the job list and a strong replica of the schedule. In other words, requesting the next scheduled job (line 17) is an asynchronous operation which resolves if strong consistency is guaranteed. In contrast, getting this next job from the job list is a synchronous operation (line 18). Programmers can install two kinds of callbacks on eventual replicas (such as jobs in our example). *onTentative* callbacks are invoked whenever the state of an eventual replica changes locally. In other words, the state of the replica has changed as the result of a local invocation of a mutating method. *onCommit* callbacks are invoked whenever the state of an eventual replica changes globally. In other words, the state of the replica has changed as the result of an invocation of a mutating method by another actor. As is the case for the dispatcher, we omit the technician's UI code which interacts with the current job (i.e. changing the description or adding pictures).

```
class Technician extends Actor{
 techName: string
 schedule : JobSchedule
 jobList : JobList
 current : Job

 init(techName){
  this.techName = techName
  let topic = new this.libs.PubSubTag("schedule")
  this.libs.subscribe(topic).once(([schedule,jobList])=>{
   this.schedule = schedule
   this.jobList = jobList
  })
 }

 jobDone(){
  schedule.nextJob.then(newJobId => {
   this.current = this.jobList.get(newJobId)
   this.current.onTentative(()=>{
    //Update UI
   })
   this.current.onCommit()=>{
    //Update UI
   })
  })
 }
}
```

Listing 4.5: Defining the technician actor.

4.2 The Entente Between Eventual and Strong Replicas

Triumvirate allows programmers to distribute remotely-active state without having to manually guarantee availability or consistency of this state. They do so using replicas, which come with a set of built-in guarantees (e.g. availability and eventual consistency for eventual replicas). To ensure these guarantees Triumvirate curtails the interactions between all three triumvirs (i.e. duplicates, replicas and derivations). In this section we focus on the interaction between duplicates, eventual and strong replicas. We discuss the interaction between replicas and derivations in the following chapter.

4.2.1 Keep them Separated

The CAP theorem states that a piece of partition tolerant state cannot be both available and consistent. As such, Triumvirate forces a clear divide between strong and eventual replicas. More concretely, three laws govern replicas.

Law 1: Preservation of Availability

Fields of an eventual replica cannot be assigned to a strong replica nor can a method of an eventual replica be invoked with a strong replica as argument.

Triumvirate enforces this law to guarantee the availability of eventual replicas. Assume that Triumvirate would not enforce the law and an eventual replica's field contains a reference to a strong replica. Operations on this strong replica are not available in the case of network partitions. Consequently, methods of the eventual replica that use this strong replica are not available under partitions. As such, this would break the eventual replica's availability guarantee (see Section 4.1.1).

Law 2: Preservation of Consistency

Fields of a strong replica cannot be assigned to an eventual replica nor can a method of a strong replica be invoked with an eventual replica as argument.

This law guarantees the consistency of strong replicas. Assume that Triumvirate would not enforce the law and a strong replica's field contains a reference to an eventual replica. Operations on this eventual replica might return diverging results across actors. Consequently, methods of the strong replica using this eventual replica might yield different results across the network. This would break the strong replica's consistency guarantee (see Section 4.1.1).

Triumvirate enforces both laws through run-time exceptions. Triumvirate checks that every assignment to a replica's field or each method invocation on a replica adheres to both laws. For example, Triumvirate is able to detect that an eventual replica is given as argument to a strong replica's method invocation. Triumvirate subsequently notifies the programmer of this error by raising an exception.

Law 3: Preservation of Serialisability

Methods of strong and eventual replicas are unable to lookup variables captured by their lexical scope.

This law guarantees the serialisability of strong and eventual replicas. The actors that make up a DRIA send replicas between themselves as a means to implement distributed shared state. As such, replicas are continuously serialised and deserialised. If a replica could lookup variables in its lexical scope the entire transitive closure of this scope would need to be serialised and deserialised as well. Triumvirate throws runtime exceptions when methods of replicas try to lookup variables in their lexical scope.

Constructors of strong and eventual replicas form an exception to Law 3. In other words, a replica is able to assign variables in its lexical scope to its fields during construction phase.

While this law might look very restricting at first sight it closely resembles approaches such as Scala's *spores* [MHO14]. In a nutshell, spores allow programmers to create closures which can be safely distributed (e.g. by enforcing that spores and the variables they capture are serialisable). Both approaches rely on the programmer to specify which variables in the replica's or spore's lexical scope are to be captured. However, the spores approach is more substantial as it includes a type system which can enforce safety properties at compile time.

4.2.2 From Consistency to Eventuality and Back

To allow a limited form of interaction, Triumvirate provides operators which convert eventual replicas into strong replicas and the other way around. On one hand *freeze* accepts an eventual replica as argument and creates a new strong replica which represents a snapshot of the eventual replica's state at freeze time. On the other hand, *thaw* accepts a strong replica as argument and returns a new eventual replica which represents a snapshot of the strong replica's state at thaw time. The replicas returned by freeze and thaw invocations do not share their identity with the argument replica. As such, there are no consistency guarantees between the replica returned from a freeze or thaw invocation and the argument replica.

```
1  class OfflineDispatcher extends Dispatcher{
2   schedule : JobSchedule
3   jobList : JobList
4
5   init(){
6    super.init()
7    let offlineTopic = new this.libs.PubSubTag("TechOffline")
8    let thawTopic = new this.libs.PubSubTag("ThawSchedule")
9    this.libs.subscribe(topic).each((techName)=>{
10    let thawSchedule = this.libs.thaw(this.schedule)
11    this.libs.publish(thawTopic,[techName,thawSchedule])
12   }
13  }
14 }
```

Listing 4.6: Extending the dispatcher actor for thawing.

```
1  class OfflineTechnician extends Technician{
2   init(){
3    super.init()
4    let thawTopic = new this.libs.PubSubTag("ThawSchedule")
5    this.libs.subscribe(thawTopic).each(([name,thawedSchedule]=>{
6     //continue using thawedSchedule
7    })
8   }
9
10  goOffline(){
11    let offlineTopic = new this.libs.PubSubTag("TechOffline")
12    this.libs.publish(offlineTopic,this.techName)
13  }
14 }
```

Listing 4.7: Extending the technician actor for thawing.

Assume that certain technicians in our fleet management application spend extended periods of time disconnected from the system. Moreover, these technicians are to complete multiple jobs during this period of disconnection. To accommodate these technicians, dispatchers allow them to access the schedule while being offline. In a nutshell, these technicians request a *thawed* version of the schedule before going offline. Listing 4.6 and Listing 4.7 show how to extend the *Dispatcher* and *Technician* actors to this end. Technicians signal that they go offline using the *goOffline* method (line 10 in Listing 4.7) that publishes an offline request under the technician's name. Dispatchers subscribe to these requests (line 9 in Listing 4.6) and respond by publishing thawed versions of the schedule. Once

a technician receives such a thawed schedule (line 5 in Listing 4.6) he is able to safely go offline for an extended period of time.

4.3 Managing Updates to Remotely-Active State

Triumvirate supports locally-active, remotely-active and reactive state updates through duplicates, replicas and derivations respectively. Supporting these different kinds of updates requires Triumvirate to implement various update strategies. For example, duplicates support locally-active updates through traditional method application and field assignment (i.e. as defined in most object-oriented languages). Replicas require different strategies depending on whether the replica is strong or eventual. In both cases Triumvirate relies on state-of-the-art algorithms to manage remotely-active updates. Finally, reactive updates to derivable state requires us to develop our own algorithm. We motivate this need and discuss the design of this algorithm in the following chapter.

In this section we detail the strategies implemented in Triumvirate to manage updates to strong and eventual replicas. More specifically, this section explains how we adapt state-of-the-art algorithms to the Triumvirate runtime. We discuss these implementations conceptually. For the source-code implementation of both update protocols we refer the reader to Triumvirate's implementation[1]

4.3.1 Strong Replicas as Far References

Triumvirate implements strong replicas using *far references* [CGS+14]. In other words, the actor that instantiates an object from a strong replicable class is said to be the *owner* of the replica. This replica can be sent from the owning actor to other actors in the application (e.g. as part of an asynchronous message, as the return value of a remote method invocation, etc.). Only the owning actor holds the actual replica, all other actors are only able to acquire a *far reference* to it. Each method invocation or field access on a far reference results in an asynchronous message sent to the owning actor and returns a promise. The owning actor performs the invocation or field access on the replica and resolves the promise with the

[1] `https://gitlab.soft.vub.ac.be/fmyter/triumvirate`

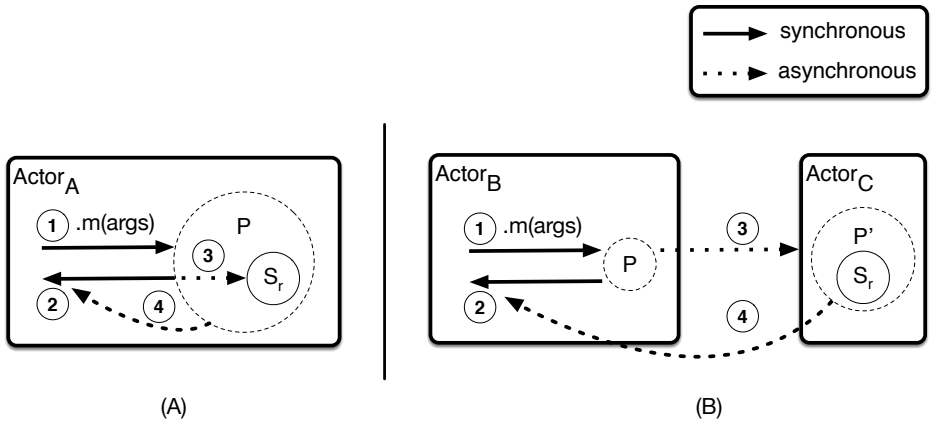


Figure 4.4: (A) Intra-actor method invocation on a strong replica. (B) inter-actor method invocation on a strong replica.

return value of the invocation or access. All messages are buffered by the far reference in case it loses connection with the owning actor.

Practically, Triumvirate uses JavaScript proxies [ecm19] to implement strong replicas. Proxies are wrappers around objects which allow one to override parts of JavaScript's built-in semantics. Programmers are able to install *traps*, or callbacks, which implement specific operations on objects. For example, the *set* trap allows programmers to override field assignment. Upon instantiating a replica from a strong replicable class Triumvirate returns such a proxy instead of the actual replica. This proxy essentially implements the update protocol for strong replicas.

Figure 4.4 show how Triumvirate uses proxies to implement strong replicas. More specifically, it depicts method invocations on strong replicas[2].

Figure 4.4(A) shows a method invocation on a strong replica from within the owning actor (i.e. $Actor_A$). In other words, an object within $Actor_A$ invokes a method m with arguments *args* on the proxy P wrapping strong replica S_r (1). P implements a trap which overrides method invocation as follows. First, P returns a promise (2) which will eventually resolve with the return value of the method invocation. Asychronously P

[2]We do not discuss the implementation of other operations (e.g. field accesses) as these follow a pattern identical to the one of method invocations.

then invokes m with arguments *args* on the replica S_r (3). Finally (4), P resolves the promise returned in (2) with the return value obtained in (3).

Figure 4.4(B) shows a method invocation on a strong replica from an actor (i.e. $Actor_B$) different than the owning actor (i.e. $Actor_C$). The protocol remains largely unchanged in comparison to Figure 4.4(A). The major difference is that step (3) and (4) now involve asynchronous messages. A proxy always has a reference to the single original replica to which it delegates method invocations, field access, etc. If the proxy resides within the same actor as the replica this reference is local, otherwise it is a far reference. Once P returns the promise in (2) it asynchronously invokes m using *args* on S_r and waits for the return value to resolve the promise (4).

Regardless of the number of actors invoking methods concurrently on their proxies, only a single actor sequentially executes these invocations on the strong replica. This mechanism guarantees sequential consistency: the operations issued by an actor are executed according to the actor's program execution. Moreover, the result of all operations on a strong replica is the same as if these operations were executed in some sequential order.

This concludes our discussion of the implementation of strong replicas using far references. In the following section we discuss the implementation of eventual replicas as global sequence data models.

4.3.2 Eventual Replicas as Global Sequence Data Models

Triumvirate relies on the global sequence protocol (GSP) [BLPF15] to govern updates to eventual replicas. GSP is able to ensure the availability of eventual replicas while maintaining a certain level of consistency (i.e. eventual consistency). We start by giving a broad overview of GSP before discussing its application to eventual replicas.

4.3.2.1 GSP in a Nutshell

We explain the global sequence protocol by means of a simple replicated counter example. Two nodes in a network own copies of this replicated counter and are both able to read and increment its value.

In a nutshell, GSP allows for concurrent and offline operations to be performed on replicated pieces of data, called *data models*. Each data

model defines an operation which can be applied over it (i.e. *Update*). Updates over a data model are aggregated in a log of operations. Furthermore, each data model is associated with a function which returns the value of an instance of the model given the update log and an initial value (i.e. *Read*). In our counter example the *Read* function returns the length of the update log, which contains increment operations.

Nodes in the network all have an instance of the data model. Nodes can perform updates on their local instances of this data model. GSP ensures that all nodes eventually read the same value for their instance of the data model. To do so it assumes that nodes communicate through a *reliable total order broadcast* (RTOB) [DSU04] communication medium (i.e. all messages are reliably received by all nodes in the same order). Triumvirate ensures RTOB through a client-server (i.e. worker and master replicas) architecture, where the server acts as a broadcaster.

Offline operations are supported by letting each instance of a data model maintain two logs of updates: *committed* and *tentative* updates. The former represents the last known global log of updates. The latter contains a log of update operations which are yet to be broadcasted. For instance, operations performed while the node lost connection with the network. Applying *Read* to the committed and tentative logs returns the current value for an instance of a data model.

Whenever a node performs an update on its instance of a data model the update is added to the tentative log. Furthermore, the update is broadcasted to all nodes. Upon receiving an update each node adds the update to the committed log. If a node receives its own update it removes said update from the tentative log. Figure 4.5 depicts how GSP ensures eventual consistency for our counter example. At *t=1*, *Node 1* and *Node 2* are in a consistent state. Both have a single update operation (i.e. an *inc*) in their committed log. Performing the *Read* operation with the committed log, tentative log and the initial value 0 therefore returns 1 for both nodes. At *t=2*, *Node 1* performs an update which is added to *Node 1*'s tentative log. This update is broadcasted by *Node 1* but not yet received by *Node 2*. The nodes are therefore in a temporarily inconsistent state because the *Read* function returns a different value for both nodes. At *t=3*, *Node 2* and *Node 1* have received the broadcast. Both nodes add the operation to their committed log and *Node 1* removes the update from

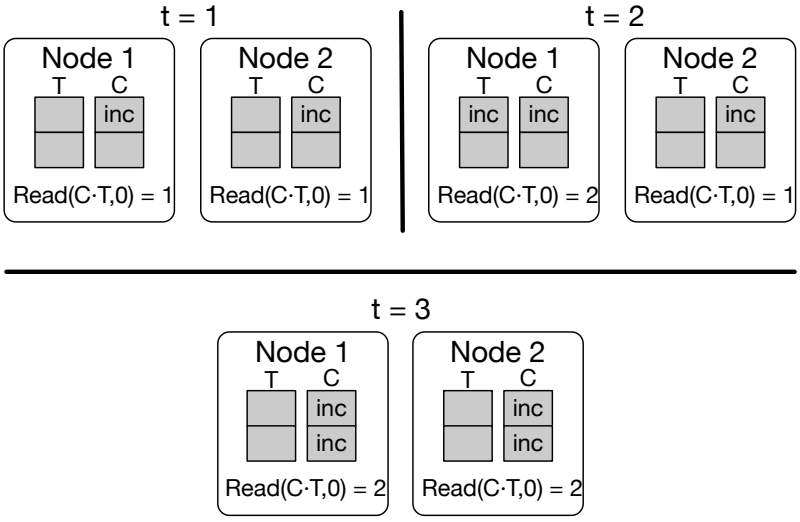


Figure 4.5: Example run of the GSP algorithm on the "counter" example.t=1, consistent starting state.t=2, *Node 1* performs an increment.t=3, consistent final state.

its tentative log. Applying the *Read* function returns a consistent value (i.e. 2) for both nodes.

We refer the reader to [BLPF15] for an in-depth explanation of optimisations for the algorithm as well as how it deals with message loss and disconnections. In the following section we detail how Triumvirate adapts GSP to implement eventual replicas.

4.3.2.2 From GSP to Eventual Replica

The global sequence protocol does not directly map onto the object-oriented programming paradigm (e.g. data models in GSP do not contain multiple fields and methods). As such, Triumvirate implements its own object-oriented variation in order to support updates to eventual replicas. We base this implementation on prior work [MCSDM16] and existing adaptations of GSP [CDMB16]. In this section we provide a conceptual overview of this implementation.

We discuss our implementation using the Triumvirate version of the counter example from the previous section. Listing 4.8 shows the imple-

```
export class Counter extends EventualReplicable{
 value

 constructor(){
  super()
  this.value = 0
 }

 @mutating
 inc(){
  this.value += 1
 }
}
```

Listing 4.8: An eventually consistent counter in Triumvirate.

mentation of an eventually consistent counter in Triumvirate. A single mutating method (i.e. *inc*) changes the counter's state that is read by reading a counter's *value* field.

Replicas and Proxies

As is the case for strong replicas, instantiating an eventual replicable class returns a replica wrapped by a JavaScript proxy (see Figure 4.6). This proxy serves as a bridge to our implementation of the global sequence protocol. It delegates the reading of fields and the invocation of mutating methods to GSP. In other words, the proxies wrapping eventual replicas implement GSP.

The owning actor of a strong replica has both the replica and its proxy while all other actors only acquire a copy of the proxy. This is not the case for eventual replicas where all actors acquire both a copy of the replica as well as a copy of the proxy. However, we distinguish between the master replica (and proxy) and worker replicas (and proxies). A master replica is created by instantiating an object from the *EventualReplicable* class (i.e. using the *new* operator). A worker replica is created whenever a master replica crosses the actor boundaries. The master replica determines the unique and total order in which operations on eventual replicas are applied. Worker replicas therefore maintain a far reference to their master replica.

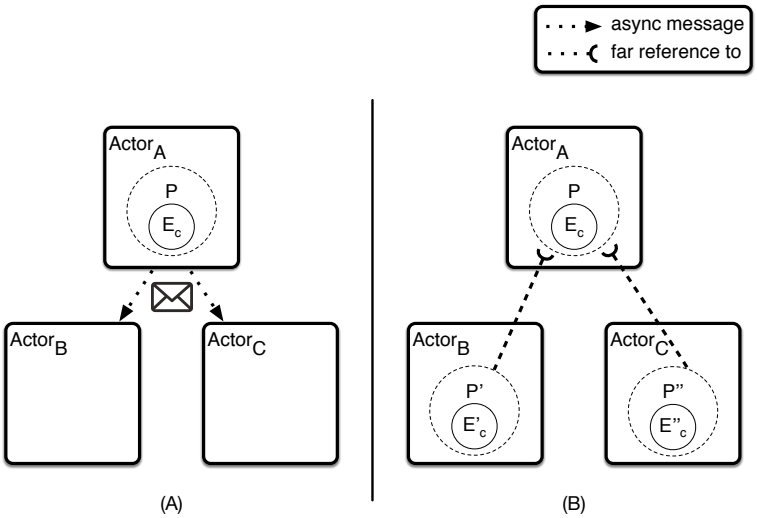


Figure 4.6: (A) System before dissemination of an eventual counter E_c. (B) System after dissemination of an eventual counter E_c.

Coordinating Replicas

A master replica serves as a central point of coordination for all its worker replicas. It is important to note that a Triumvirate application can contain multiple master replicas spread across multiple actors. As such, Triumvirate does not impose a single central point of coordination for all eventual replicable state in DRIAs.

Figure 4.6 shows three actors. Assume that $Actor_A$ instantiated an object from the *Counter* class and therefore owns a master replica E_c (which stands for ***e***ventually ***c***onsistent counter) wrapped by a proxy P. In Figure 4.6(A) $Actor_A$ sends an asynchronous message to both $Actor_B$ and $Actor_C$. Assume that this message contains a reference to E_c. As a result Triumvirate creates a deep copy of E_c and attaches it to the asynchronous message. Law 3 ensures that this does not require Triumvirate to deeply copy E_c's lexical scope.

In Figure 4.6(B) both actors have received the message. Upon receiving the message each actor deserialises the copy of E_c and creates a new

proxy wrapper (i.e. P' and P''). Triumvirate ensures that the copies of E_c hold far references to its original master replica residing in $Actor_A$.

Replicas as Data Models

Technically, eventual replicas are regular JavaScript objects. However, each replica maintains *two versions* for each of its fields: a tentative and a committed version. As such, each field in a replica is its own GSP data model. As is the case for vanilla GPS data models, the committed version of a field represents the last known global value across all replicas. Conversely, the tentative version of a field might diverge from the tentative versions of the same fields for other replicas. Mutating methods are the equivalent of *update* operations in GSP, although a single mutating method can update multiple fields simultaneously.

For example, a *counter* replica maintains a tentative and committed version of its *value* field. The *inc* method updates the tentative version of this field. The committed version of the field is only changed when the master replica confirms a new global operation to be applied.

Reading a Replica's Field

Our object-oriented adaptation adheres to the so-called *states and deltas* optimisation of GSP [BLPF15]. Data models, or fields in our case, do not keep a log of tentative and committed operations. Rather, each field of an eventual replica is a tuple containing the field's tentative and committed versions. These versions are obtained by incrementally reducing the operations sent by the master replica. In other words, operations are no longer added to a log of operations (i.e. as was the case for vanilla GSP) but are directly applied to a given initial state.

Reading a replica's field always returns the field's tentative version. This ensures that a read always reflects the latest operations locally applied to the replica. For example, the proxies wrapping counter replicas ensure that all accesses to the *value* field return its tentative version.

Invoking Mutating Methods

Assume that $Actor_B$ in Figure 4.6(B) invokes *inc* on its replica of the counter. This invocation not only impacts the replica on which the method

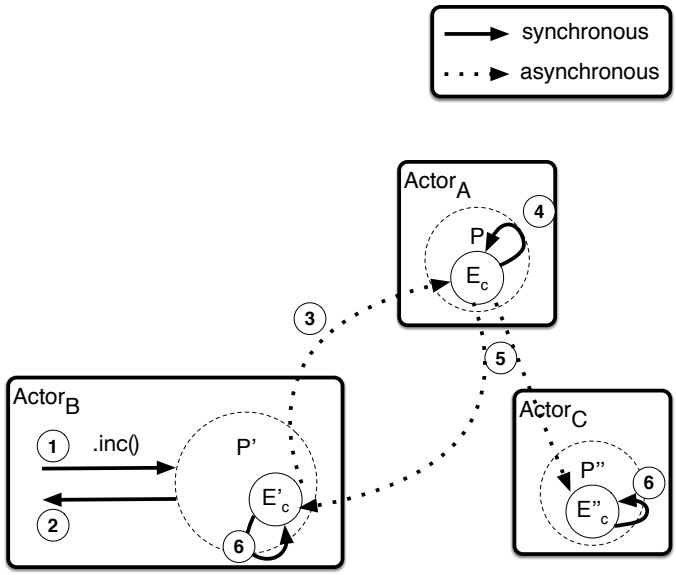


Figure 4.7: Invocation of a mutating method and subsequent events.

is invoked but it also impacts all other replicas. Figure 4.7 shows the protocol that Triumvirate initiates upon invocation of a mutating method.

This protocol operates according to the following steps:

Step 1. An object within $Actor_B$ invokes the *inc* method on proxy *P'* wrapping the eventual counter E'_c.

Step 2. The wrapper invokes the method on E'_c using the tentative version of its *value* field and immediately returns the return value to the object that invoked *inc*.

Step 3. Triumvirate encapsulates information about method invocations in *round* objects. A round contains the name of the invoked method as well as the invocation's arguments. This round object is sent to the master replica, which assigns it a time stamp.

Step 4. The master replica invokes the method specified by the round using the arguments contained in the round. The master replica's goal is to dictate the order in which rounds are to be executed. As such, the master replica always executes methods using the committed version of its fields. We say that the master *commits* the

received round and subsequently sends the round for commitment to all worker replicas.

Step 5. When a worker replica receives a round it first checks whether the new round directly follows the last committed round (i.e. using the time stamps contained by rounds). This allows worker replicas to determine whether they missed a number of rounds (e.g. due to network partitions, etc.). If rounds have been missed the worker replica requests these from the master replica.

Step 6. Worker replicas commit the round sent by the master replica. In other words, the method is invoked using the committed versions of the replica's fields. Subsequently the tentative versions of the replica's fields are set to the committed version. This leaves worker replicas in a consistent state with the master replica.

This concludes the implementation of eventual and strong replicas in Triumvirate. We refer the reader to Section 6.6.3 of Chapter 6 for two additional implementation of replicas using CRDTs [SPBZ11] and the two-phase commit protocol [LS76]. In the following section we discuss the performance differences between strong and eventual replicas.

4.4 Strong versus Eventual Replication: a Performance Choice

Programmers implement different parts of their distributed systems using eventual or strong replicas based on *functional* requirements (e.g. offline availability). However, these two kinds of replicable state also differ in their *performance* characteristics. To showcase these characteristics we perform micro-benchmarks using the online grocery list application presented in [MSDM18a] called Myosotis. The application allows members of a household to collaboratively edit a grocery list. Moreover, while shopping for the groceries the household members are able to mark certain items as "bought" (e.g. upon putting the item in their cart).

Figure 4.8 shows a screenshot of the application. The navigation bar at the top of the screen shows the grocery lists linked to the user's account. Moreover, it allows to create new grocery lists. The lower part of the screen shows all the items contained within one of these grocery lists. The

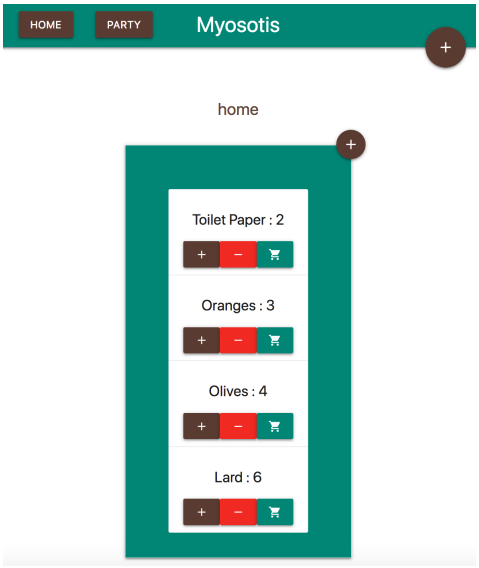


Figure 4.8: Screenshot of Myosotis.

quantity needed of each item can be incremented or decremented. Moreover, a user is able to mark an item as *bought* using the *"shopping cart"* button. Myosotis requires both available and consistent functionality:

Available Functionality Once a user logs in it receives a replica of all the grocery lists created for its account. A user is always able to create a new list, add an item to a list or change an item's quantity (i.e. increment or decrement it). Two users logged into the same account might therefore temporarily witness different values for the collection of lists, items in a list or the quantity of an item.

Consistent Functionality Marking an item as *bought* happens strongly consistent. In other words, an item can only be marked as *bought* once by a single user. Consequently, this functionality is only available to users which are connected to the Myosotis server.

Our micro-benchmarks compare two versions of the application. The first version (i.e. *MyosotisAP*) implements all list functionality(i.e. creating lists, adding items to a list, etc.) using two eventual replicas (i.e. an eventual replica that implements individual grocery items and an eventual replica that implements the list of grocery items). The only strong

replica in *MyosotisAP* is responsible for marking items as bought. The second version (i.e. *MyosotisCP*) solely uses strong replicas. In other words, *MyosotisCP* clients are unable to use any functionality while being offline. However, all changes to lists are always ensured to be strongly consistent across all clients.

We compare both versions using two performance characteristics. Assume that an operation o changes the state of a given eventual or strong replica r. We define these characteristics as follows:

Time to Consistency (TC) is the time required for the state change induced by o to be visible on all other replicas.

Time to Local Change (TLC) is the time required for the state change induced by o to be visible on r itself (i.e. within the actor in which o is applied).

The micro-benchmarks are conducted on an Ubuntu 14.04 server with two dual core Intel Xeon 2637 processors (2 physical threads per core) at 3.5 GHz with 265 GB of RAM using CAPtain.js[3] version 0.5.0. For each version of Myosotis we simulate 50 clients concurrently performing 10 operations on the shared state. The Myosotis server and all simulated clients are implemented as actors executing on the same physical machine. As such, our benchmarks do not account for network latencies across machines. Rather, our benchmarks are only affected by local network latencies (i.e. given that actors communicate through websockets). However, this does not impact our benchmarks given that these aim to showcase the relative performance differences between eventual and strong replicas.

Figure 4.9 shows the results of these benchmarks and highlights the fundamental difference between eventual and strong replicas. In *MyosotisAP* the TLC is roughly a factor 1000 faster than the TC. In comparison, the TLC and TC for *MyosotisCP* are almost identical. There are two reasons for this difference. First, eventual replicas are able to perform operations immediately which results in low TLC. However, maintaining all eventual replicas eventually consistent produces a significant performance overhead: Each operation must be sent to the server which conceptually replays the entire log of operations. Subsequently this log is sent to *all*

[3]CAPtain.js is Triumvirate's predecessor and is available at `https://github.com/myter/CAPtain`.

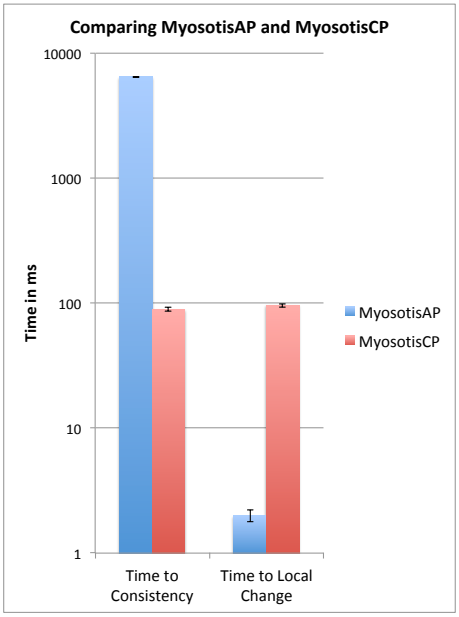


Figure 4.9: Comparing MyosotisAP and MyosotisCP. Error bars indicate the 95% confidence interval.

replicas which in turn replay all operations as well. Second, operations are only performed by a strong replica if it can guarantee strong consistency. In other words, such operations only require a single round trip message to the server. At the end of this round trip the strong replica is deemed consistent and the change has been applied locally.

4.5 A Live Experiment

The aim of this section is to showcase the practical usefulness of Triumvirate's replicas. To this end we developed an interactive online presentation. The source of inspiration behind this presentation are approaches such as reveal.js[4] that allow one to create slides and slide shows in HTML. We extend this idea by implementing our presentation as a full-fledged DRIA. More specifically, the presentation is made interactive by distributing logic and state between the server, the presenter and the audience.

[4] `https://github.com/hakimel/reveal.js/`

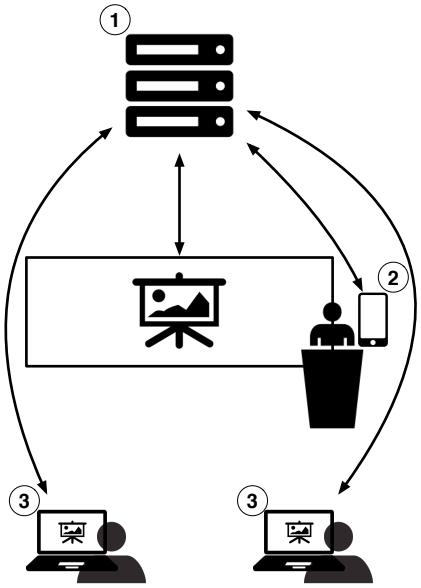


Figure 4.10: Overview of the presentation app.

The presentation serves to informally demonstrate how replicas significantly ease the development of DRIAs. We demonstrated this by using the presentation application at the Onward!2018 conference [MSDM18a] to present the very concept of replicas[5]. This section first provides a more in-depth explanation of our interactive presentation after which it highlights parts of the presentation's implementation. We invite the reader to browse through the video[6] of the actual Onward!2018 presentation.

4.5.1 Overview of the Application

Figure 4.10 provides a conceptual overview of the presentation set-up. We distinguish three key parts in this set-up:

Presentation (1) The presentation is a distributed rich internet application hosted by one of the Vrije Universiteit Brussel's servers. The presenter loads the application in his browser and projects the browser's window onto a screen in front of the audience.

[5] Do note that replicas were named consistents and availables at the time
[6] `https://www.youtube.com/watch?v=17cHhDDpJbg`

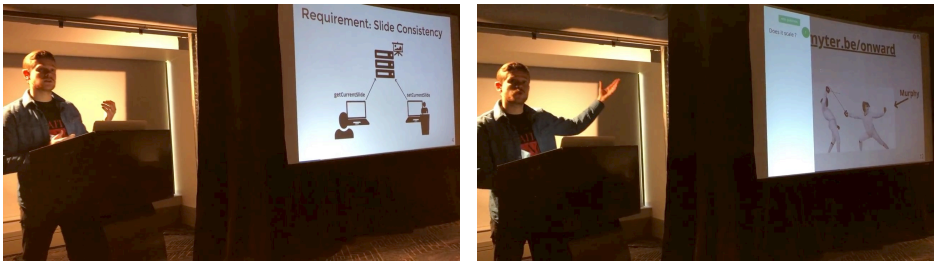


Figure 4.11: (Left) The presenter using the presentation application. (Right) The presenter opens the questions dialogue, shown on the left side of the screen, to explain its functionality to the audience.

Presenter (2) The application offers smartphone-specific functionalities for the presenter. In other words, the application offers an endpoint to which the presenter can connect by using his smartphone. This mobile application's functionality is twofold:

Offline Mode In case the presenter detects that the network is bad or unreliable he can go into offline mode. This allows the presenter to move through the slides even without being connected to the presentation's server.

Pointing functionality By dragging on his smartphone's screen the presenter projects a red dot on the current slide (i.e. the red dot is visible to all clients). Additionally the presenter is also able to go to the next or previous slide by swiping right or left respectively.

Audience (3) Members of the audience follow the presentation in their browsers by navigating to the server's public endpoint. A button overlaid on the slides allows members of the audience to submit questions. Moreover, members of the audience are able to upvote interesting questions.

Figure 4.11 shows two pictures taken during the live experiment at the Onward!2018 conference. The picture on the left shows the presenter using the application. The picture on the right shows the presenter explaining the application's question-related functionality to the audience. The left part of the screen shows the menu used by members of the audience to ask or upvote questions. Figure 4.12 shows a screenshot of the questions

Figure 4.12: Screenshot of the Onward!2018 presentation.

menu. The greyed-out part of the screenshot shows the current slide while the left part contains the list of questions asked by the audience together with the amount of votes for each question.

The presentation application, with its complex functionalities, qualifies as a DRIA:

Distributed Logic We distinguish two distributed functionalities. *First*, the presenter's laptop and smartphone are both able to change the current slide (i.e. move forwards or backwards in the slide deck). *Second*, all clients (i.e. members of the audience and presenters) are able to ask or upvote questions

Distributed State The application maintains two instances of distributed state, each with varying consistency requirements. *First*, the slide deck keeps track of the current slide's number (the actual HTML representing the slides is sent to the client upon connecting). This part of the application's state must be kept strongly consistent

to ensure that all clients always see the same current slide. As a consequence the slide deck is not available, in case of partitions one might not be able to view the current slide or increment the counter. A *second* part of the application's state is the list of questions and their votes. This part of the application's state can safely be kept available. Members of the audience are always (i.e. even under partitions) able to ask or upvote a question. Consequently, the state of the question list might temporarily diverge amongst members of the audience.

4.5.2 Implementation in Triumvirate

Overall the entire presentation is implemented in 1003 lines of Triumvirate code, excluding the HTML code required for the visual aspect of the slides. The complexity of the presentation's features (e.g. offline available question list, smartphone as click&point device, etc.) and the relatively small amount of code required to implement them showcases the strength of replicas. In what follows we highlight the most important aspects of the implementation. Parts of the code have been simplified or omitted for the sake of readability. However, these simplifications and omissions do not hide complexity on behalf of Triumvirate or the presentation's implementation. We refer the reader to the source code[7] for the complete implementation of the presentation.

[7] `https://github.com/myter/onward`

4.5.2.1 Implementing Questions and Slides

```
1  class Question extends EventualReplicable{
2      text     : string
3      votes    : number
4      id       : string
5
6      constructor(text){
7          super()
8          this.text   = text
9          this.votes  = 1
10         this.id     = "..."
11     }
12
13     @mutating
14     incDecVote(delta){
15       this.votes += delta
16     }
17 }
18 class QuestionList extends EventualReplicable{
19     questions : Map<string,Question>
20
21     constructor(){
22         super()
23         this.questions = new Map()
24     }
25
26     @mutating
27     newQuestion(question : Question){
28         this.questions.set(question.id,question)
29     }
30 }
```

Listing 4.9: The question-related state of the presentation.

The presentation's distributed state is implemented using three replicas. Two eventual replicas, given in Listing 4.9, implement the question-related state of the presentation. One eventual replica implements (line 1) individual questions. It keeps track of the actual question (i.e. the text) and the amount of votes the question has. A mutating method (i.e. *incDecVote*) change the question's total votes. This methods can be invoked even though the client is offline. Triumvirate ensures that eventually all clients read the same value for a question's *votes* field.

The second eventual replica (line 18) implements the list of questions. Internally it maintains a map of question identifiers to question objects. A single mutating method (i.e. newQuestion) mutates the state of the question list. All members of the audience share a single instance of *QuestionList*, which is kept eventually consistent by the Triumvirate runtime.

```
1  class SlideShow extends StrongReplicable{
2      currentSlide  : number
3      listeners : Array<FarRef<Client>>
4      constructor(){
5          super()
6          this.currentSlide = 0
7          this.listeners = []
8      }
9
10     incDecSlide(delta){
11       this.currentSlide += delta
12       this.listeners.forEach(listener => {
13        listener.slideChange()
14       })
15     }
16
17     onChange(listener : FarRef<Client>){
18       this.listeners.push(listener)
19     }
20 }
```

Listing 4.10: The slideshow-related state of the presentation.

The slideshow is implemented using a single strong replica, as shown in Listing 4.10. The slideshow essentially serves as a strongly consistent counter that indicates the current slide. A single method (i.e. *incDecSlide*) allows the presenter to update the slideshow's state. Triumvirate guarantees that all connected clients always read the same value for the slideshow's *currentSlide* field. Moreover, the *onChange* method accepts a far reference [CGS+14] (see Section 2.3.1.1) to a client and notifies this client whenever the slideshow's state changes.

4.5.2.2 Implementing the Presentation Server

The presentation server's goal is twofold. First, the server allows clients to connect and receive their code (i.e. actor definition), the application's state (i.e. three replicas) and the HTML needed to render the slides. Second, the server implements the presenter-specific functionality (i.e. overlaying a red "laser dot" on the slides and going in offline mode).

Listing 4.11 shows the implementation of the presentation server. The server's first role (i.e. registering connecting clients and sending them their initial data) is implemented on line 11 and line 12. These two lines create two endpoints (i.e. URLs) to which clients can connect to receive their source code and the slides' HTML source-code.

```
export class OnwardServer extends Application{
    clients
    slideShow
    questionList

    constructor(){
      super()
      this.clients = []
      this.slideShow = new SlideShow()
      this.questionList = new QuestionList()
      this.libs.serveApp("presenter.html","presenter.js",8002)
      this.libs.serveApp("audience.html","audience.js",8888)
    }

    registerPresenter(credentials){
      if(isPresenter(credentials){
       return generateToken()
      }
    }

    registerClient(clientRef : FarRef<Client>){
        this.clients.push(clientRef)
        return [this.slideShow,this.questionList]
    }

    goOffline(token){
      if(verified(token)){
        resolve(this.libs.thaw(this.slideShow as any))
      }
    }

    goOnline(token,availableSlides){
      if(verified(token)){
       this.slideShow = this.libs.freeze(availableSlides)
       client.slideReset(this.slideShow)
      }
    }

    moveDot(token,x,y){
      if(verified(token)){
       this.clients.forEach((client : FarRef<Client>)=>{
        client.dotPosition(x,y)
       })
      }
    }
}
```

Listing 4.11: Implementing the presentation server.

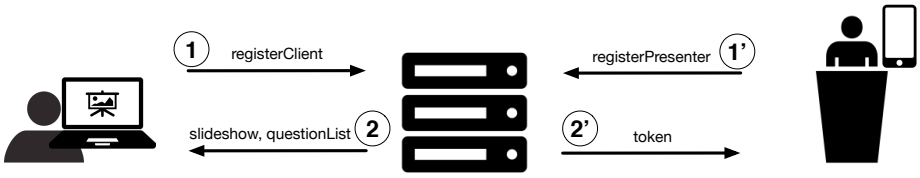


Figure 4.13: Interaction between clients and the presentation server during registration.

Figure 4.13 provides a diagrammatic overview of the interaction between the clients and server while registering. Audience members start by invoking the *registerClient* method (see Figure 4.13(1) and line 21) that returns the presentation's state (i.e. a replica of the slideshow and question list, see Figure 4.13(2)). Similarly, presenter devices invoke the *registerPresenter* method (see Figure 4.13(1') and line 15). This method accepts a set of credentials and returns a security token in case these credentials are correct (see Figure 4.13(2')). This token serves to ensure that only presenter devices are able to invoke some of the server's methods.

The presenter-specific functionality is implemented as follows:

Offline Mode The counter keeping track of the current slide dictates which slide is currently shown by connected clients. This counter is implemented as a strong replica and can therefore only be incremented or read by clients connected to the server. However, as we discuss in Section 4.5.1, the presenter has the ability to go into offline mode by pressing on a dedicated button. This button invokes the *goOffline* method (line 26) which returns a thawed version (i.e. an eventual replica) of the slideshow. Conversely, the *goOnline* method (line 32) accepts a thawed slideshow and freezes it.

Triumvirate's built-in *freeze* method returns a new strong replica of the provided eventual replica. *goOnline* therefore sends the new slideshow, resulting from the invocation of freeze, to all clients. Both methods first check whether the token passed as argument is valid (i.e. whether the presenter is invoking the method and not a malicious user).

Pointing functionality The presenter can project a red dot on the current slide by dragging his finger over his phone's screen (see Section 4.5.1). When the presenter's smartphone detects this dragging it invokes the server's *moveDot* method (line 39). This method instructs all clients to draw the red dot at the given position.

4.5.2.3 Implementing the Presentation Clients

The presentation application's clients mostly react to changes made to the presentation's state. Besides this, clients also implement a number of methods invoked directly by the application server. Listing 4.12 shows the implementation of these clients.

```
1  export class Client extends Application{
2
3      constructor(){
4        super()
5        let server = this.libs.buffRemote(serverAddress,serverPort)
6        server.registerClient(this)
7        .then(([slides,questions])=>{
8              slides.onChange(this)
9              questions.onCommit(_ => {
10               //Update UI
11             })
12             questions.onTentative(_ => {
13               //Update UI
14             })
15             newQuestionButton.onClick(questionText=>{
16               let question = new Question(questionText)
17               questions.newQuestion(question)
18             })
19         })
20     }
21
22     slideChange(){
23       //Update UI by switching to this.slideShow.currentSlide
24     }
25
26     dotPosition(x,y){
27       //Move dot on current slide
28     }
29
30     slideReset(newSlides){
31         newSlides.onChange(this)
32     }
33 }
```

Listing 4.12: Implementing the presentation server.

Clients start by acquiring a far reference to the presentation's server (line 5) using the built-in *buffRemote* method. This method takes the server's address and port as argument and returns a far reference to the specified server actor. Given this far reference clients register at the server by invoking its *registerClient* method, which returns a replica of the slideshow and question list (see Figure 4.13). Clients register two callbacks (i.e. *onTentative* and *onCommit*) which are invoked whenever the question list changes and which update the user interface. Moreover, clients register themselves as listeners for changes to the slideshow using its *onChange* method. In other words, as soon as the slideshow's state changes it invokes the client's *slideChange* method (line 22). Clients are also able to mutate the state of the question list. The callback on line 15 is invoked whenever the user completes the dialogue to ask a new question. It creates a new question and adds it to the question list. Triumvirate ensures that this addition eventually triggers the *onCommit* callbacks for all clients. Other methods of a client (i.e. *dotPosition* and *slideReset*) are invoked directly by the server and have been discussed in the previous section.

The presenter runs its own application actor which extends the *Client* class. The presenter's application class adds a number of user interface event handlers that directly invoke methods implemented by the server (e.g. the offline-mode button invokes the *goOffline* method on the server). We omit the presenter's code from this section as it does not significantly contribute to the implementation of the presentation.

4.5.3 Live Benchmarking

Besides the previously discussed functionality, our presentation at Onward!2018 also offers live benchmarking functionality. This functionality is limited to a single slide and allows the presenter to run micro-benchmarks on the audience's devices. In a nutshell, the specific slide contains a button only visible to the presenter that, once pressed, deploys benchmarking code on all connected devices. This benchmarking code then executes and the results are displayed in real-time in the slide.

```
1  export class OnwardServer extends Application{
2   //...
3   startBenchmark(token){
4    if(verified(token)){
5     let eventual = new EventualBench(this.clients.length)
6     let strong = new StrongBench(this.clients.length)
7     this.clients.forEach(client => {
8      return client.startBench(eventual,strong)
9     })
10    Promise.all([eventual.onFinish(),strong.onFinish()]).then(
       results)=>{
11     let result = aggregate(results)
12     this.clients.forEach(client => {
13      client.showResults(result)
14     })
15    }
16   }
17  }
18 }
19
20 export class Client extends Application{
21  //...
22  startBench(eventualBench, strongBench){
23   while(!benchmarkFinished){
24    eventualBench.perform()
25    strongBench.perform()
26   }
27  }
28 }
```

Listing 4.13: Extending the presentation server and client for live benchmarking functionality.

Listing 4.13 extends the server and client's code provided by Listing 4.11 and Listing 4.12. Concretely, the server defines a *startBenchmark* method (line 3) that is invoked when the presenter presses a button during the live-benchmark slide. This method creates a strong and eventual replica that is sent to all clients. The clients perform a number of operations on these replicas (line 22) after which the server aggregates the results. These

```
1  export class EventualBench extends EventualReplicable{
2   limit
3   performed
4   finished
5
6   constructor(limit){
7    super()
8    this.limit = limit
9    this.performed = 0
10  }
11
12  @mutating
13  perform(){
14   this.performed += 1
15   if(this.performed == limit){
16    this.finished()
17   }
18  }
19
20  onFinish(){
21   return new Promise(resolve=>{
22    this.finished = resolve
23   }
24  }
25 }
26
27 export class StrongBench extends StrongReplicable{
28  //same as BenchEventual definition
29 }
```

Listing 4.14: Implementing the live-benchmarking replicas.

results are then sent to all clients to be displayed in the presentation's benchmark slide[8].

Listing 4.14 contains the definitions of both live-benchmarking replicas. The replicas provide one mutating operation *perform* (line 13) that increments a counter. When this counter reaches a set limit the replica resolves its *onFinish* promise to notify listeners that it has finished benchmarking.

4.5.4 Experimental Results

The presentation application serves as anecdotal evidence that replicas significantly ease the development of DRIAs. With only 1003 lines of code

[8]The real implementation accumulates results incrementally and updates the results in real-time on the slide. We simplify this for the sake of readability.

(excluding the HTML source code that implements the slides' graphical content), a fully functional and robust DRIA was implemented. The presentation received general praise from the audience (± 20 simultaneous distributed users). Testimony to this is a mail received from an audience member after the presentation:

> First I must say that I am very impressed by the courage to use your own system while presenting! Last time I saw that was 1988 (or so) when a guy from [...] was giving a Keynote at OOPSLA (1500 or so attendees) and failed miserably. I feared the worst, but you did very well (Prof. Dr. Boris Magnusson, personal communication, November 9, 2018).

The same audience member points out a number of minor bugs (e.g. on some devices the red dot's placement was shifted compared to the dot on the projected screen). However, these bugs have to do with the implementation details of the presentation itself and not bugs within Triumvirate, replicas or our model. Although this live experiment does not formally prove the universality of our approach it does show that replicas allow one to easily implement DRIAs.

4.6 Related Work

We divide the work related to Triumvirate's replicables in two categories. The first category of related work offers language abstractions dedicated to distributed and shared state with various consistency guarantees. The second category of related work focuses on replicated data stores on which programmers can apply queries with various levels of consistency.

4.6.1 Consistency-oriented Languages and Language Abstractions

Replicables are heavily inspired by *Repliqs* [CDMB16]. A *repliq* is a first-class replicated object which is kept eventually consistent across clients through the *global sequence protocol* [BLPF15]. *Repliqs* allow programmers to implement the *AP* aspects of distributed systems. We extend the work presented in [CDMB16] with constructs to implement the *CP* aspects of distributed systems. Moreover, we introduce constructs to con-

vert the *AP* aspects of a distributed system into *CP* aspects and vice versa.

Cloud Types [BFLW12] are a programming model in which programmers are able to declare certain data types to be "cloud data". In other words, the programmer is able to declare that certain parts of the application's state are shared across the network. Cloud types come with a number of built-in cloud data types (e.g. numbers). Moreover, programmers are able to declare custom cloud data types. However, programmers need to provide merge functions for their custom data types in order to allow the cloud types runtime to handle conflicting updates. This contrasts with our approach that allows programmers to write custom replicated data types without requiring complicated merge functions.

Geo [BBB+17] is a geo-replicated actor system built atop the Orleans [BB16] programming language. The way Geo integrates various levels of data consistency into its programming model closely resembles Triumvirate. In a nutshell, Geo programmers are able to apply two kinds of operations on an actor's state: eventually consistent and linearisable operations. Moreover, Geo allows programmers to implement their own custom consistency protocols. Geo resides on the opposite side of the design spectrum with regards to Triumvirate. It allows programmers to implement both the *AP* and *CP* aspects of their application on a per-operation level. In contrast, Triumvirate allows programmers to do this on a per-data-type level.

Bayou [KKW19] is a replication protocol which mixes strongly and weakly consistent operations. Its workings closely resemble that of GSP. In a nutshell, replicas maintain a list of tentative and committed operations. The committed list represents the globally consistent order of operations performed by all replicas. Conversely, the tentative list might diverge amongst replicas and represents locally performed operations. Bayou resembles replicables in the sense that both aim to provide a single framework to reason about strongly and weakly consistent data. The difference between both approaches is the level of consistency granularity. Bayou allows for individual operations to be either strongly or weakly consistent. In contrast, a replicable either allows for only strong or weak method invocations.

Correctables [GPS16] are a language construct which allows programmers to perform operations on replicated objects using different levels of

consistency. Invoking an operation on a *correctable* will initially return a weakly consistent result after which it will progressively be refined with more consistent results (e.g. strongly consistent results). Additionally, correctables allow programmers to explicitly specify the level of consistency desired for a particular operation. Correctables differ from our approach with regards to the level of granularity on which programmers specify the desired level of consistency. Using correctables, programmers specify the desired level of consistency *per operation*. Both approaches also differ from a programmer's perspective. [GPS16] presents an API consisting of three methods: *invokeStrong*, *invokeWeak* and *invoke*. These methods allow the programmer to specify the desired level of consistency given an operation to be performed on a replicated object. Moreover, the correctables API allows programmers to implement their own consistency guarantees. In contrast, Triumvirate provides a full-fledged distribution model: it enables programmers to define the consistency levels for data types as well as how instances of these data types should be replicated amongst actors.

Conflict-free replicated data types (CRDTs) [SPBZ11] are a kind of abstract data type that guarantees availability and *strong eventual consistency*[9]. CRDTs are able to guarantee this by enforcing a number of properties on the operations they support (i.e. commutativity, idempotency and associativity). Consequently, CRDTs lack general applicability: only a limited amount of data types can be implemented as CRDTs. Although solutions have been proposed to alleviate this generality problem (e.g. JSON CRDTs [KB17]), CRDTs inherently only allow to implement the *AP* aspects of a distributed system.

Lasp [MVR15] is a distributed programming language whose sole data abstractions are CRDTs [SPBZ11]. Lasp is therefore able to model the *AP* aspects of a distributed system while guaranteeing strong eventual consistency. However, Lasp lacks the programming constructs to implement the *CP* aspects of a distributed system.

Dexter [TG11] is a Java framework which allows programmers to implement various distributed parameter passing semantics. Two of these semantics are of particular interest compared to the work presented in

[9] A variant of eventual consistency that guarantees *strong convergence*. In a nutshell, strong convergence states that replicas that receive the same updates must be in an equivalent state.

this dissertation. Pass by *remote reference* is essentially the same as AmbientTalk's *far* references or *E*'s *eventual* references. In other words, using pass by *remote reference* one is able to implement the *CP* aspects of a distributed system. Pass by *copy-restore* allows an object to be passed by copy between a server and a client. Changes made to the copy by the server are later restored on the client. To some extend this enables the implementation of the *AP* aspects of a distributed system in Dexter. To the best of our knowledge *copy-restore* does not provide any consistency guarantees. In other words, conflicts arising from concurrent modifications are not resolved. In contrast, eventual replicas allow for concurrent modifications while ensuring eventual consistency.

In the tuple space model [Gel85] processes conceptually access a globally shared memory comprised of data structures called *tuples*. Processes can write, read and remove tuples from this global memory. In the traditional tuple space model as defined by [Gel85] a centralised server maintains the state shared by clients. This model therefore only allows to implement the *CP* aspects of a distributed system. Other tuple space models [MZ04, GSMD14] replicate tuples across clients, allowing them to read or write tuples while being offline. However, these models do not account for conflicting updates to the conceptually shared tuple space.

E [MTS05] and AmbientTalk [CGS+14] both provide language constructs to implement the *CP* aspects of a distributed systems (i.e. *eventual* and *far references* respectively). Moreover, AmbientTalk provides *isolates* which are a kind of object that adhere to pass-by-copy semantics. Although *isolates* can therefore be used to implement the *AP* aspects of a distributed system they provide no consistency guarantees whatsoever. In other words, the states of two instances of the same *isolate* are never synchronised (i.e. as is the case for duplicates in Triumvirate).

Bloom [ACHM11] is a distributed programming language in which programs are expressed as a set of set manipulations over discrete timesteps. Moreover, these sets can be distributed amongst bloom programs running on a network. Simply put, Bloom offers built-in support for the set CRDT. Moreover, Bloom^{L} [CMA+12] extends Bloom with support for user-defined CRDTs. As such, programmers can use Bloom and Bloom^{L} to implement the *AP* aspects of their distributed systems.

4.6.2 Replicated Data Stores

A number of approaches have been proposed which allow programmers to perform operations (e.g. queries) on replicated data stores with various levels of consistency. As is the case for Triumvirate, these approaches allow programmers to implement the AP and CP aspects of their web applications. In contrast to replicated data stores, Triumvirate enables programmers to implement the AP and CP aspects of their web applications by providing replicated data as first-class values in a general purpose programming model.

Using Sieve [LLC+14] programmers specify application invariants to help static and dynamic analyses to determine optimal consistency levels for operations on the data store. Operations which can run under weak consistency are translated to *commutative shadow operations* (i.e. operations on CRDTs). In Quelea [SKJ15] programmers write contracts which specify the application-level consistency requirements of operations. The Quelea runtime statically verifies these contracts while a theorem prover maps these contracts to consistency properties which adhere to the contract's semantics. DCCT [ZN16] allows programmers to separate a data store's objects into *regions*. These regions are annotated with varying degrees of consistency which influences the semantics of the read and write operations one can perform on objects within a region. IPA [HBZ+16] programmers specify consistency policies by using an extensive annotation system (e.g. one can dynamically specify consistency policies based on the system's latency). Furthermore, IPA's type system allows it to enforce a number of properties at compile time (e.g. weakly consistent values never flow into strongly consistent operations). ConSysT [MS17] provides consistency specifications at the type level (i.e. values are typed with the desired consistency level). ConSysT's type system guards the programmer from erroneously combining values with different consistency levels (e.g. low consistency values flowing into high consistency computations).

4.7 Limitations

Eventual and strong replicas have a number of limitations which are the results of the synchronisation mechanisms used by Triumvirate. The global sequence protocol achieves eventual consistency by conceptually replaying the global log of operations for each replica. In our implementation

these operations amount to an eventual replica's methods. Programmers therefore need to be aware that side effects in methods can potentially be replayed multiple times, at different locations in the network.

The implementation of strong replicas uses far references: all strong replicas forward method invocations and field accesses to the original object. As such, strong replicas in Triumvirate offer a replication factor of 1. This makes strong replicas brittle with regards to failures. If the actor owning the original replica crashes, all other replicas will no longer be available.

In Section 6.6.3 of Chapter 6 we showcase two different implementations of eventual and strong replicas by using Triumvirate's meta-programming layer. These implementations are based on CRDTs and the two phase commit protocol [LS76] and mitigate the aforementioned limitations, though each at their own cost.

4.8 Replicas and Requirements for a DRIA-Oriented Programming Model

Replicas allow Triumvirate to partially fulfil requirements R_2 and R_3 for DRIA-oriented programming models as follows:

R_2: The model guarantees data consistency properties specified by the programmer

Triumvirate programmers use replicas to specify the consistency guarantees required for their application's state. More precisely, programmers can specify that a particular part of the state should be strongly or eventually consistent. Triumvirate ensures that these consistency properties are guaranteed throughout the application's lifetime, even in the face of concurrent operations.

However, the consistency guarantees provided by replicas are limited to remotely-active state. As such, replicas only partially fulfil R_2 as they do not support reactive state. In the following chapter we discuss how derivations complete replicas and allow Triumvirate to fulfil R_2 completely.

R_3: The model supports multiple parameter passing semantics

Replicas provide pass-by-replication parameter passing semantics.

In other words, replicas automatically maintain bookkeeping information as they are disseminated across the network. This bookkeeping information is then used by Triumvirate to ensure the consistency guarantees of the replicas.

As is the case for R_2, replicas only allow Triumvirate to *partially fulfil* R_3 as they do not support reactive state. In the following chapter we discuss how derivations provide pass-by-derivation parameter passing semantics.

4.9 Chapter Summary

Part of the state in DRIAs requires remotely-active update semantics: Different nodes in the network conceptually share a piece of state. When a node updates the state of the DRIA, this update becomes visible to all other nodes in the network. Triumvirate allows programmers to easily implement this kind of state by extending the *Replicable* class. These data structures provide built-in pass-by-replication parameter passing semantics and remotely-active update semantics.

As stated by the CAP theorem [Bre00, GL02], a piece of partition-tolerant distributed state cannot be both available and consistent. This has traditionally forced programmers to manually implement this trade-off in their web applications. To alleviate this burden from programmers Triumvirate provides two kinds of replicas: eventual and strong replicas. Instances of the former are available: actors are always able to read or update their state. However, eventual replicas only guarantee eventual consistency: the state of two replicas might temporarily diverge across actors. Instances of the latter guarantee consistency: their states never diverge across actors. However, an actor is not always able to read or update a consistent replica's state.

We discuss how Triumvirate's approach compares to the state of the art in programming language approaches to data replication and consistency. Most of these approaches offer constructs or abstractions which resemble either eventual or consistent replicas. However, Triumvirate is the first distributed programming language offering high-level support for both available and consistent remotely-active distributed state.

Chapter 5

Deriving Reactive State

In the previous chapter we discussed replicable state: state that is shared amongst actors and supports remotely-active updates. In this chapter we discuss derivable state: state that updates automatically as other parts of a system's state change.

At a first glance, the reactive programming (RP) paradigm [BCC+13] seems a perfect fit to implement derivable state. This paradigm is tailored towards event-driven applications and allows programmers to elegantly handle time-varying values in their applications. Moreover, the paradigm has recently [DSMM14, MBS+18] been harnessed to implement distributed systems. However, the centralised nature of current distributed reactive programming runtimes renders them inapt to handle the derivable state of DRIAs.

This chapter is divided in two main parts. In the first part we present *derivations* (i.e. instances of derivable state) as a means for programmers to implement reactive state in their DRIAs. Derivations provide an API which is heavily inspired by existing reactive programming approaches. However, existing reactive runtimes are ill-fit to support derivations in Triumvirate. The main problem is that these runtimes rely on centralised coordination to ensure that reactive updates happen in a correct order. In the second part of this chapter, which is based on [MSDM19], we therefore focus on Triumvirate's novel decentralised reactive runtime.

Figure 5.1 highlights the concepts discussed in this chapter as well as the requirements that derivations tackle. More precisely, in this chapter we detail how derivations partially fulfil requirements R_2 and R_3 for DRIA-

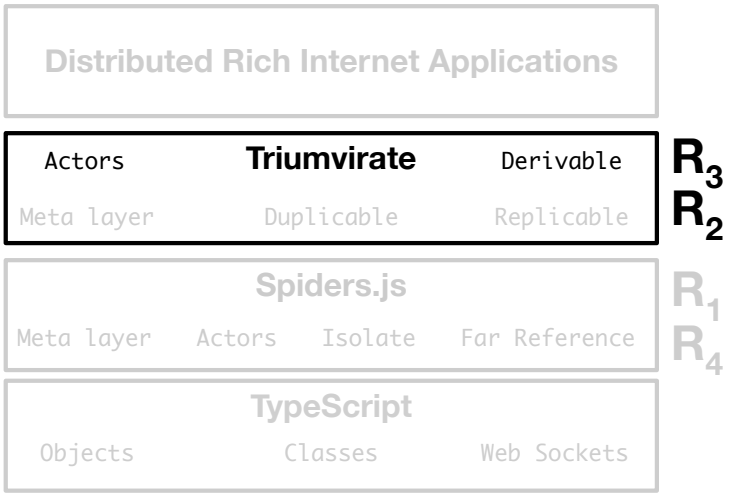


Figure 5.1: Concepts discussed in this chapter.

oriented programming models. In other words, we describe how derivations *guarantee data consistency properties specified by the programmer* and how they provide *support for multiple parameter passing semantics.*

5.1 Derivable State and Reactive Programming

Reactive programming is epitomised by three core concepts. First, *signals* represent time-varying values (e.g. the current mouse position). Second, programmers combine these signals using *lifted functions*. Third, the reactive language's runtime constructs a *dependency graph* from these combinations of signals. Moreover, the runtime ensures that changes to a source signal propagate through the dependency graph, thereby updating the application's state.

The fleet management application introduced in Section 2.1 of Chapter 2 is typically the kind of systems which one implements using reactive programming. Figure 5.2(A) provides an overview of the part of our fleet management application that is relevant to this chapter. Reactive programming allows developers to implement the services' internal logic. For example, the *fleet service* is implemented as follows. The serialised vehicle data is represented as a signal, each time this signal changes the data is deserialised and persisted. We therefore use two lifted functions:

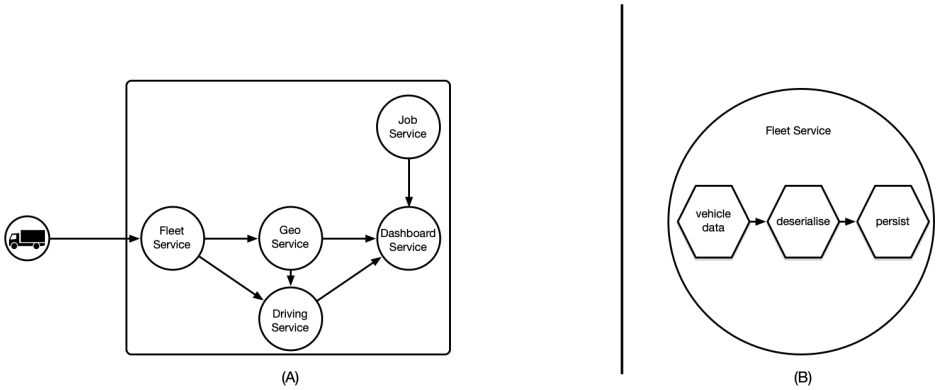


Figure 5.2: (A) Overview of the fleet management application. (B) The fleet service's internal dependency graph.

one which deserialises the vehicle's data and one which persists the deserialised data. Figure 5.2(B) depicts the dependency graph constructed by the reactive runtime for the fleet service. The runtime's propagation algorithm traverses the dependency graph in a topological order as soon as a source signal changes. The algorithm updates each signal using its predecessors' values during this traversal. For example, changes to the vehicle data signal trigger the algorithm to update the deserialise signal with the vehicle's new data. Subsequently, it updates the persist signal with the new deserialised data.

5.1.1 Distributed (un)Reactive Runtimes

Distributed reactive programming [CMVCDM10, MS14, RDP14] extends the concepts of reactive programming to the realm of distributed systems. In other words, programmers are able to apply lifted functions to signals which reside on physically distributed machines. For example, using DRP the geo service applies a lifted *reverse geo coding* function on the fleet service's deserialise signal. Whenever the deserialise signal changes (i.e. as a result of a change in a vehicle's data), the *reverse geo coding* function is invoked automatically (we say that the geo service *updates* whenever the deserialise signal changes).

As explained in the previous section each microservice contains a dependency graph which represents its internal logic. However, using DRP

the services themselves also form a *distributed* dependency graph. For example, the arrows in Figure 5.2(A) show the distributed dependency graph for our fleet management application. Whenever a vehicle's data changes, this updates the *fleet* service which propagates the deserialised data to the *geo* and *driving* services. Subsequently, both services update before they propagate their new values further downstream to the *dashboard* service.

As is the case for non-distributed dependency graphs, a propagation algorithm performs a traversal in a topological order of the distributed dependency graph to update all signals. To showcase the importance of this traversal order, consider the following hypothetical scenario. A vehicle sends its updated data to the *fleet* service, which subsequently deserialises and persists this data. Assume that the *geo* and the *dashboard* services update before the *driving* service. The dispatcher looking at the rendered dashboard might witness a faulty speed limit violation because the vehicle's position in the dashboard was updated before its driving statistics. This phenomenon is called a *glitch* [CK06]. A common strategy employed by non-distributed reactive programming languages to avoid glitches is to topologically sort the dependency graph before propagating values through it [CK06, MGB+09].

Problem Statement

The aforementioned non-distributed glitch-freedom strategy does not trivially scale towards distributed systems. A single central coordinator would be required to explicitly sort the distributed dependency graph and to determine when each node may update. As we detail in Section 2.1.1, the fleet management application requires updates to propagate from the *fleet* service to the *dashboard* service as fast as possible. A central coordinator would introduce a single point of failure as well as a significant performance bottleneck that would hamper the reactivity of the application.

Solution

Triumvirate's derivations provide an API similar to that of distributed reactive programming. They allow programmers to elegantly combine reactive states distributed across the network. However, Triumvirate implements a novel reactive runtime which guarantees glitch freedom without resorting to centralised coordination. As such, Triumvirate allows for the

```
class Vehicle extends Actor{
 init(){
  let topic = new this.libs.PubSubTag("Vehicle")
  let data = this.libs.derive(_=>{
   return readBeacon()
  },this.libs.seconds)
  this.libs.publish(topic,data)
 }
}
```

Listing 5.1: Defining the data uploaded by vehicles.

development of the vehicle-related functionality of our fleet management system while also meeting its reactivity requirement. The following section discusses the derivation API, after which we specify the reactive runtime.

5.2 Derivations in Practice

Figure 5.2(A) provides an overview of the reactive state in our fleet management application. More precisely, the figure shows how the fleet members' data flows through various microservices. In a nutshell, these services provide the following functionality. The *fleet service* serves as an entry point for fleet members to upload their data to the server. The service deserialises and persists the uploaded data. The *geo service* converts a fleet member's GPS coordinates into physical street addresses. The *driving service* calculates eco-driving scores and generates alerts based on a fleet member's data and physical location. The *job service* allows technicians to query and modify job-related data. Lastly, the *dashboard service* combines fleet member and technician data into a coherent whole.

In the following sections we detail how one uses Triumvirate's derivations to implement the flow of data within and across microservices.

5.2.1 Local Reactivity using Derivations

The vehicle-related functionality of our fleet management system is implemented using derivable state. Vehicles continuously upload their most recent information, after which a series of microservices transform this data before the dispatcher sees it on his dashboard. Listing 5.1 defines the actor which implements the vehicles. A vehicle's task is twofold: read the state of the hardware beacon at regular intervals and send said state

```
1  class Fleet extends Actor{
2
3   init(){
4    let vehicleTopic = new this.libs.PubSubTag("vehicle")
5    let fleetTopic = new this.libs.PubSubTag("fleet")
6    this.libs.subscribe(vehicleTopic).each((serialisedData)=>{
7     let deserialised = this.libs.derive(deserialise,serialisedData)
8     this.libs.derive(persist,deserialised)
9        this.libs.publish(fleetTopic,deserialised)
10   })
11  }
12 }
```

Listing 5.2: Defining the fleet service.

to the fleet service. To achieve the first task we use the *derive* function, which is part of the standard actor library. *Derive* takes a function and distributed input data (i.e. duplicates, replicas or derivations) as argument and outputs a new derivation. Every time the state of one of the input data changes, the function is reevaluated to calculate the new state of the output derivation.

Triumvirate's standard library contains the *seconds* derivation, which updates every second. On line 4 the vehicle actor derives *serialisedData* from seconds using a function which reads the state of the hardware beacon. For the sake of simplicity we assume the existence of a *readBeacon* method. Every second (i.e. when the state of the *seconds* derivation changes) all derivations that depend on *seconds* are recomputed. In other words, the vehicle actor reads the beacon information every second. In turn this updates the *serialisedData* derivation and all derivations that depend on it as well.

The vehicle's second task, sending its state to the *fleet* service, is done through Triumvirate's publish-subscribe mechanism (line 7). Note that the vehicle does not explicitly specify to which services its data must be sent. This loosens the coupling between the distributed components in our system.

5.2.2 Distributed Reactivity using Derivations

Triumvirate's *derive* function can be applied over distributed state regardless of the state's locality. In other words, an actor can apply the deriva-

tion function to state received from another actor. The output derivation's state is recomputed as soon as the state of one of the inputs changes.

Consider the fleet service, defined by the actor in Listing 5.2. It subscribes to the topic under which vehicles publish the *serialisedData* derivation (line 6). Whenever such a derivation is published, the *fleet* service reacts as follows. First, it derives the deserialised version of the serialised input data. Second, it persists this deserialised data. Third, it publishes the deserialised data under its own topic.

Deserialising and persisting the data happens via two derivation functions (i.e. *deserialise* and *persist*, of which we assume the existence). These derivations depend (respectively directly and indirectly) on the *serialisedData* derivation published by vehicles on line 7 in Listing 5.1. In other words, every second the serialised data is updated by the vehicle which in turn updates the *fleet* service's deserialised derivation that is automatically re-persisted upon updating. All these updates happen automatically and glitch free. This frees the programmer from manually ensuring reactive updates across actors.

5.3 Deriving, Replicating and Imperatively Mutating

In Chapter 4 we discuss how Triumvirate supports remotely-active state using replicas. These replicas, together with derivations and duplicates, form the distributed state of DRIAs. As such, derivations are bound to interact with duplicates and replicas. In this section we discuss the law that governs these interactions. We start the discussion with a number of examples.

5.3.1 Combining Reactivity and Activity

Triumvirate supports two kinds of updates on distributed state: active (i.e. local or remote) updates and reactive updates. Active updates are control-flow driven while reactive updates are data driven. To exemplify this difference, consider a traditional online spreadsheet application (e.g. Google Sheets). As shown in Figure 5.3, the client side of this application consists of two states: the document object model (DOM) tree (i.e. a tree that contains html elements) (shown on the left) and the spreadsheet's

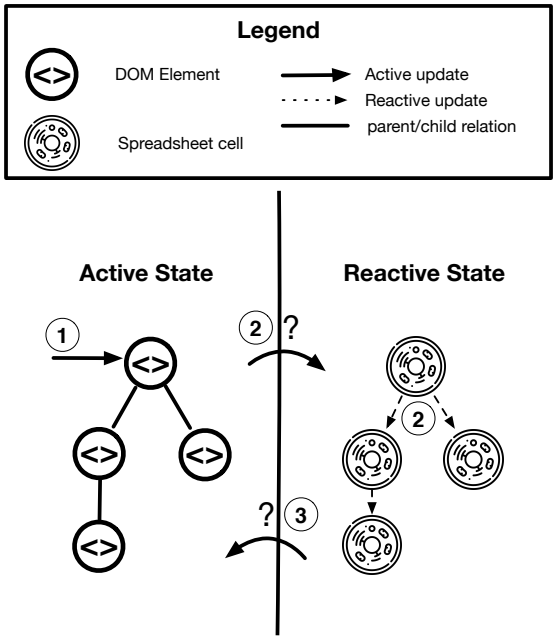


Figure 5.3: Active and reactive state in an online spreadsheet application.

cells (shown on the right). The DOM tree is imperatively (i.e. *actively*) updated by the application as the user interacts with the application interface (1). In contrast, cells of a spreadsheet update *reactively* as new data becomes available. The application's programmer must translate active updates to the DOM into reactive updates to spreadsheet cells (2). Conversely, reactive updates to the spreadsheet cells must be translated into active updates to the DOM tree (3). Related work discusses the combination of active and reactive state as well as translating updates from one kind to another at length [VdVDKMDM17, ICK06]. In the following sections we explain how Triumvirate deals with combinations of active and reactive state.

5.3.2 Reactivity and Activity in Triumvirate

Figure 5.4 provides a general overview of the interactions between active and reactive distributed state in Triumvirate. Duplicates and various kinds of replicas implement the *active* state in Triumvirate applications. As we discuss in Chapter 4, active updates to replicas involve various

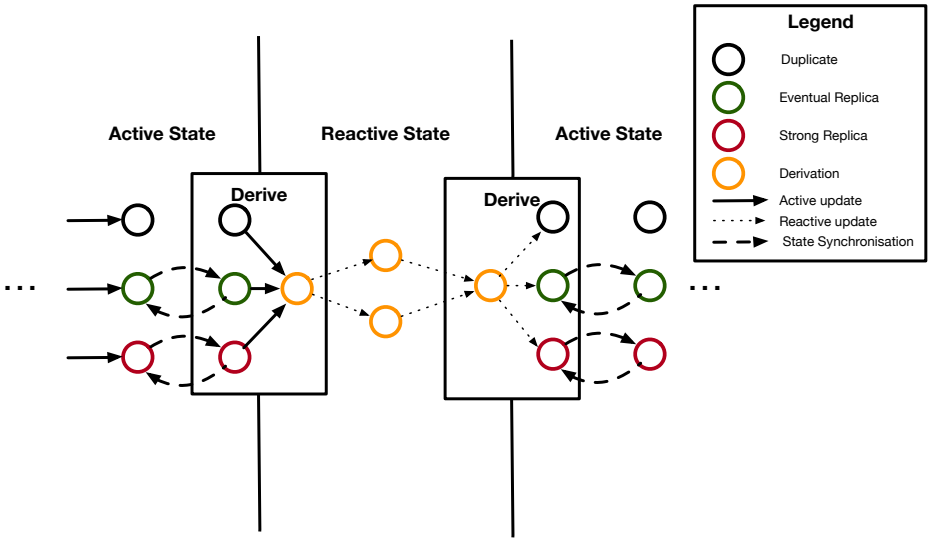


Figure 5.4: Overview of state interactions within Triumvirate.

state synchronisation mechanisms. Derivations implement the *reactive* state in Triumvirate applications. It is important to note that each actor in Triumvirate can contain *both* active and reactive state. In other words, the state depicted in Figure 5.4 could be distributed amongst any number of actors.

The *derive* function available in the standard library of all Triumvirate actors allows programmers to translate control-flow-driven active updates from duplicates or replicas to data-driven reactive updates on derivations and vice versa. As an example of this translation consider the implementation of Triumvirate's built-in *seconds* derivation given by Listing 5.3.

The listing comprises two parts. A first part implements the active clock state using the *Clock* duplicable (line 1 to line 13). A clock provides a single method *tick* that increments the clock value.

A second part concerns the implementation of Triumvirate actors (line 15 to line 31). We omit all code from this class which does not directly relate to the seconds derivation. Actors maintain a single *clock* instance (line 16). Actors increment the clock's value every second by recursively invoking the *tickLoop* method (line 25). The *seconds* derivation is created on line 20 by deriving from the clock duplicate using the provided function. Changes to the clock's state (i.e. as a result of an invocation of the

```
1  class Clock extends Duplicable{
2   value
3
4   constructor(){
5    super()
6    this.value = 0
7   }
8
9   @mutating
10  tick(){
11   this.value += 1
12  }
13 }
14
15 class Actor{
16  clock : Clock
17
18  init(){
19   this.tickLoop()
20   this.libs.seconds = this.libs.derive((c : Clock)=>{
21    return c.value
22   },this.clock)
23  }
24
25  tickLoop(){
26   setTimeout(()=>{
27     this.clock.tick()
28     this.tickLoop()
29   },1000)
30  }
31 }
```

Listing 5.3: Defining the seconds derivation.

mutating *tick* method) trigger re-evaluations of this function that updates *seconds*' state.

This example demonstrates how derivation functions translate active updates into reactive update: an active clock tick triggers a reactive update for *seconds* and all its derivations. Conversely, the body of the derivation function (i.e. line 21) has access to mutable state (i.e. the clock) and is able to translate reactive updates to active updates if need be.

5.3.3 Derivation and Replication

In general, *derive* takes a derivation function as argument and any number of distributed states (i.e. duplicates, replicas or derivations). Derive returns a new derivation that updates, according to the provided derivation

function, as soon as a mutating method is invoked on one of the input arguments. We discern *two* types of derivations.

Source derivations solely have replicas or duplicates as input arguments. For example, the derivation published by vehicles in our fleet management system are source derivations.

Intermediate derivations solely have other derivations as input arguments. For example, the derivation published by the fleet service is an intermediate derivation.

A third type of derivation (i.e. one that combine derivations, replicas and duplicates) is forbidden by Triumvirate in accordance to the following law:

Law 4: Activity-Reactivity Isolation

The *derive* function either accepts only active state (i.e. duplicates or replicas) as arguments or only reactive state (i.e. derivations) as arguments.

The reason for this law is the following. Source derivations serve as a bridge between active and reactive state: they translate active update to reactive updates. Intermediate derivations allow programmers to declaratively construct data flows across actors. An intermediate derivation updates if one of its predecessors in the graph has updated and if this update does not cause a glitch in the application. Adding a replica or duplicate dependency to an intermediate derivation goes against these update semantics. Intermediate derivations would then become subject to changes occurring outside of the reactive dependency graph. This goes against the declarative nature of derivations and breaks Triumvirate's glitch freedom guarantees. Triumvirate ensures that programmers cannot create such derivations by throwing run-time exceptions.

This concludes our overview of the programmatic aspects of derivations in Triumvirate. In the following section we discuss distributed glitch freedom and how Triumvirate tackles it.

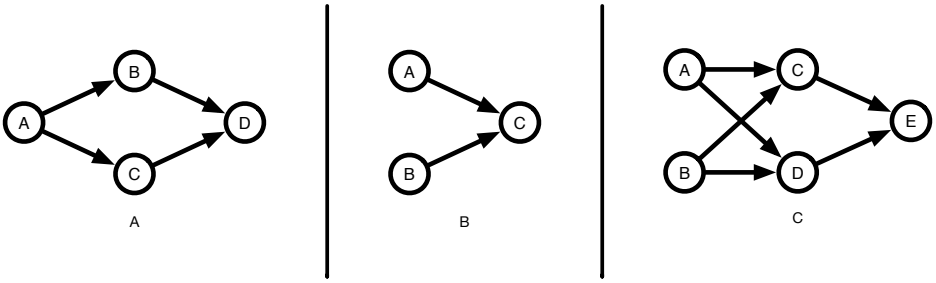


Figure 5.5: A) A dependency graph susceptible to glitches. B) A dependency graph not susceptible to glitches. C) A dependency graph susceptible to concurrent glitches.

5.4 Distributed Glitch Freedom

Informally, a reactive application glitches if a node in the dependency graph updates before its predecessors. For example, in our fleet management application (see Figure 5.2) this happens if the *dashboard* updates using new data from the *driving* service and old data from the *geo* service. Non-distributed reactive applications trivially guarantee the absence of such glitches. It suffices for the propagation algorithm to update nodes according to the topology of the dependency graph. Similarly, in a centralised distributed context a single coordinator can ensure that distributed nodes in the graph update in the correct order. Guaranteeing glitch freedom in a decentralised distributed context, such as our fleet management system, is significantly more complex.

Glitches in Distributed Reactive Systems

The key intuition behind glitches is that they can only occur for certain topologies of dependency graphs. Consider Figure 5.5(A), whenever A propagates a value both B and C need to update before they propagate values to D. We say that D is susceptible to glitches with regards to A.

Our fleet management system is a concrete example of such an update ordering. In contrast, in Figure 5.5(B) no node is susceptible to glitches: C can update as soon as it receives a new value from either A or B (which are both trivially free from glitches). For example, if C receives a new

propagation value from A it uses the previously received value from B to update itself.

In order to detect glitches, assume A in Figure 5.5(A) possesses a logical clock which it increments each time it propagates a value to its successors. Moreover, A tags each value it propagates with the current clock time. $U_D(val_B, val_C)$ denotes the updating of D using a value val_B propagated by B and a value val_C propagated by C. D must adhere to the following constraint to avoid glitches, where $Time_A(v)$ denotes A's clock time tagged to value v : $U_D(val_B, val_C) \iff Time_A(val_B) == Time_A(val_C)$. Furthermore, values can propagate concurrently through the distributed dependency graph. Consider Figure 5.5(C), A and B might concurrently propagate new values. Given network delays these values can be received by C, D and E at arbitrarily different points in time. E's constraint to avoid glitches is therefore the following:

$$U_E(val_C, val_D) \iff Time_A(val_C) == Time_A(val_D) \wedge Time_B(val_C) == Time_B(val_D)$$

To summarise, the concurrent and asynchronous nature of DRIAs can lead to glitches that would not occur in non-distributed or centralised systems.

5.5 Glitch Freedom using Queued Propagation

The rest of this chapter discusses the propagation algorithm that Triumvirate uses to guarantee glitch freedom. This algorithm, called queued propagation (QPROP) [MSDM19], governs the updates of derivations across Triumvirate actors. It ensures that each derivation updates glitch-freely without resorting to centralised coordination.

QPROP serves as a general solution to the problem of distributed glitch freedom. Although we apply it specifically to derivations in Triumvirate, it can be used as a foundation for other distributed reactive approaches. We therefore make abstraction of derivations in the discussion of QPROP presented below. Instead, we represent a distributed reactive program by means of its dependency graph.

QPROP is divided into **three phases**. First, during the *exploration* phase each node uses its neighbours to explore its position in the graph. Second, the *barrier* phase ensures that the exploration phase has successfully finished before nodes start to propagate values. Third, the *propagation* phase does the actual propagation of values.

QPROP assumes the following:

- As is commonly the case in reactive programming, dependency graphs are finite and acyclic [DSMM14, MGB+09, CC13]. Moreover, there are no intentional topological changes to the graph after the exploration phase. In other words, no new nodes or edges are added to the dependency graph after the exploration phase.
- The nodes in the dependency graph are aligned with the unit of distribution (e.g. a microservice, a process running on a server, a derivation etc.).
- Propagation of values *within* a single node (i.e. through a non-distributed dependency graph) is abstracted as an update function. We assume that this update function provides glitch free propagation of values *within* nodes.
- Nodes communicate with one another through an asynchronous communication medium which ensures exactly once, in-order delivery of messages.
- At the start of the application (i.e. before the exploration phase) each node has references to its direct predecessors and successors in the graph. References uniquely identify nodes (e.g. references are IP addresses).

Before we start the presentation of QPROP, we introduce our notation.

5.5.1 Notation

We represent a node n as a 9-tuple:
$n = (DP, DS, I, S, U, initVal, lastProp, self, clock)$.
Each element in the tuple contains the following information:

DP is the set of n's direct predecessors.

DS is the set of n's direct successors.

I is a dictionary of input sets which stores values propagated by n's direct predecessors (i.e. I's keys are references to predecessors).

S is a dictionary where the keys are references to source nodes and the values are sets of references to direct predecessor which are included in paths from the key source node to n.

U is n's update function, its arity equals DP's cardinality. Once called with the values of n's predecessors this function returns a single value to be propagated downstream by n.

initVal is n's initial value (i.e. the value before its first update).

lastProp is n's last propagated value.

self is a reference to n.

clock is a logical clock which n uses to timestamp propagation values.

We describe QPROP in pseudocode notation. Each node in the distributed dependency graph runs the three phases of the algorithm in sequence (i.e. first the exploration phase, then the barrier phase and finally the propagation phase). During these phases nodes communicate only using asynchronous messages. We provide pseudocode notation of message handlers to describe a node's behaviour upon receiving a particular message. We denote sending an asynchronous message m with arguments $(arg_1, ..., arg_n)$ to a node n using: $n \leftarrow m(arg_1, ..., arg_n)$. Moreover, using *await* we specify that the execution of the pseudocode only continues once n's message handler returns (e.g. $value = await\ n \leftarrow m(arg_1, ..., arg_n)$). To denote the elements within a dictionary we use the following notation: $[k, v] \in D$ where $[k, v]$ is a key-value pair and D is a dictionary. To read the value bound to key k in dictionary D we write: $D.k$. To add a key-value pair $[k, v]$ to a dictionary D we write: $D = D \cup \{[k, v]\}$. Finally, we use the equality symbol (i.e. $=$) to assign values to variables and use the following symbol for equality: $==$.

5.5.2 Exploration Phase

ALGORITHM 1: Exploration

```
1  sourcesReceived = 0
2  foreach pred ∈ DP do
3  |   I = I ∪ {[pred, {}]}
4  end
5  if |DP| == 0 then                                   /* I am a source node */
6  |   lastProp = (self, initVal, {[self], 0}, 0)
7  |   for succ ∈ DS do
8  |   |   succ ← sources({self}, lastProp)
9  |   end
10 end
```

Handler sources(sources,initProp)

```
1  from = initProp.from
2  I.from = I.from ∪ {initProp}
3  sourcesReceived += 1
4  foreach s ∈ sources do
5  |   if [s, _] ∉ S then
6  |   |   S = S ∪ {[s, {}]}
7  |   end
8  |   S.s = S.s ∪ {from}
9  end
10 if sourcesReceived == |DP| then
11 |   allSources = {s|[s, _] ∈ S}
12 |   sourceClocks = {[s, 0]|s ∈ allSources}
13 |   lastProp = (self, initVal, sourceClocks, 0)
14 |   for succ ∈ DS do
15 |   |   succ ← sources(allSources, lastProp)
16 |   end
17 end
```

Algorithm 1 provides the specification of a node's behaviour during the exploration phase. The algorithm is executed for each node at the start of the reactive program. Only the node's direct predecessors, direct successors, initial value, self reference and logical clock are known at this point (i.e. *DP*, *DS*, *initVal*, *self* and *clock* contain this information, all other node elements are empty).

Informally the purpose of QPROP is twofold. First, each node computes the paths from source nodes which lead to that node. Second, nodes populate their I dictionaries with their predecessors' initial values.

At the start of the exploration phase each node creates a new dictionary per predecessor to store that predecessor's input set (line 2 to line 4). Moreover, source nodes send the *sources* message (line 5 to line 9) which contains a singleton set with their self reference and their

initial propagation value. We represent propagation values as 4-tuples $propVal = (from, value, sClocks, fClock)$: *from* is a reference to the node propagating the *value*, *sClocks* is a dictionary of clock times for all source nodes which are direct or indirect predecessors of *from* and *fClock* is *from*'s clock time.

The *sources* Handler defines how nodes handle the *sources* message. As soon as a node receives a set of source references from each of its direct predecessors it relays these references together with its initial propagation value to all its direct successors (line 10 to line 16 in the *sources* Handler). At the end of this process each node in the graph knows which source nodes are able to reach it and through which direct predecessor. For example, in Figure 5.5(C) E's S dictionary contains $[A, \{C, D\}]$ and $[B, \{C, D\}]$

5.5.3 Barrier Phase

ALGORITHM 2: Barrier

```
1  startsReceived = 0
2  if
     |DS| == 0 ∧ sourcesReceived == |DP|
     then
3      foreach pred ∈ DP do
4          pred ← start()
5      end
6  end
```

Handler start()

```
1  startsReceived += 1
2  if startsReceived == |DS| then
3      foreach pred ∈ DP do
4          pred ← start()
5      end
6  end
```

Glitches could occur if values were to propagate before all nodes were able to construct their input queues. The barrier phase, see Algorithm 2, allows source nodes to determine when it is safe to start propagating values.

As soon as a sink node (i.e. a node without successors) is done exploring (i.e. it has received a *sources* message from all its direct predecessors (line 2)) it sends a *start* message to all its predecessors to indicate that they can start producing values. Any non-sink node relays this message upstream as soon as it has received a *start* message from *all* its direct successors (line 2 in the *start* Handler). A source node starts propagating values once it receives a *start* message from all its direct successors. At this point the source node knows by induction that all downstream successors are done exploring.

Handler change($v_{new} = (from, value, sClocks, fClock)$)

```
1  I.from = I.from ∪ {v_new}
2  allArgs = ×({{v_new}} ∪ {i|i ∈ I \ {I.from}})
   matches = {args ∈ allArgs|∀arg_dp1, arg_dp2 ∈ args : [s, {dp1, ..., dp2}] ∈ S →
3                                  arg_dp1.sClocks.s == arg_dp2.sClocks.s}
4  if matches ≠ ∅ then
5  |  lastMatch = max(matches)
6  |  clock += 1
7  |  lastProp = (self, U(lastMatch.values), lastMatch.sClocks, clock)
8  |  foreach succ ∈ DS do
9  |  |  succ ← change(lastProp)
10 |  end
11 |  foreach arg = (f, v, sc, fc) ∈ lastMatch do
12 |  |  I.f = I.f \ {vals ∈ I.f|vals.fClock < fc}
13 |  end
14 end
```

5.5.4 Propagation Phase

Each key-value pair $[s, preds] \in S$ informs a node that it can only update using values received from predecessors in *pred* if these values have equal clock times for source node s. In other words, the following must hold:
$U(arg_{pred_1}, ..., arg_{pred_n}) \iff \forall [s, \{pred_i, ..., pred_j\}] \in S : arg_{pred_i}.sClocks.s == arg_{pred_j}.sClocks.s$

The *change* Handler defines the heart of QPROP (i.e. how values propagate through the distributed dependency graph in a glitch free way). It starts as soon as a node n receives a new propagation value $v_{new} = (from, value, sClocks, fClock)$ which is immediately stored in n's I set for *from* (line 1). We assume that each input set in I is totally ordered based on its values' *fClocks*. Subsequently, n computes a nested set of all possible arguments to its update function (line 2). This partially ordered set is obtained by taking the cross product (marked by the $\times$ operator on line 2) of v_{new} with each of n's predecessors' I sets.

n filters this set of arguments to only contain glitch-free sets of arguments (line 3). Furthermore, n takes the lexicographic maximum of these sets of arguments (line 5). This set contains the last value propagated by each predecessor which can be used to update n in a glitch free way. n uses these values to invoke its update function after which it propagates its updated value to all direct successors (line 7 to line 10). We assume that *lastMatch.values* and *lastMatch.sClocks* respectively return a set contain-

ing the value of each argument in *lastMatch* and the union of the *sClocks* of each argument in *lastMatch*.

Finally, n removes all stale values from its input sets. We consider a value to be stale if a newer value propagated by the same predecessor has previously been used as argument to U. All values older than the ones in *lastMatch* are stale and can therefore be removed (line 11).

5.6 QPROP by Example

This section serves as an example of QPROP's workings. Consider an application comprised of five microservices as shown in Figure 5.6. The update functions of these microservices go as follows:

C adds the values it receives from A and B (i.e. $C = A + B$).

D subtracts the values it receives from A and B (i.e. $D = A - B$).

E adds the values it receives from C and D
(i.e. $E = C + D = (A + B) + (A - B = 2 * A)$.

Figure 5.6 provides an overview of the application's state as values propagate through the microservices. We discuss QPROP's behaviour at each time step.

t=0 The barrier phase is completed. We assume that A, B, C, D and E respectively have 5, 3, 8, 2 and 10 as initial values. Each node stores its predecessors' initial propagation values in its I set. For example, both C and D store A's initial value (i.e. $(A, 5, \{[A, 0]\}, 0)$) in their $I.A$ sets (see Section 5.5.2 for an overview of the data contained in propagation values). Moreover, the figure also shows each node's S set. For example, E's S set contains two entries: one which specifies that C and D propagate values originating from A and one which specifies that C and D propagate values originating from B. In other words, E can only update using values from C and D if the following holds:

$$U(val_C, val_D) \iff val_C.sClocks.A == val_D.sClocks.A \wedge val_C.sClocks.B == val_D.sClocks.B$$

t=1 A updates to 7 and propagates a new value $(A, 7, \{[A, 1]\}, 1)$ to its direct successors. At this point in time, only C receives this new value and stores it in its $I.A$ set.

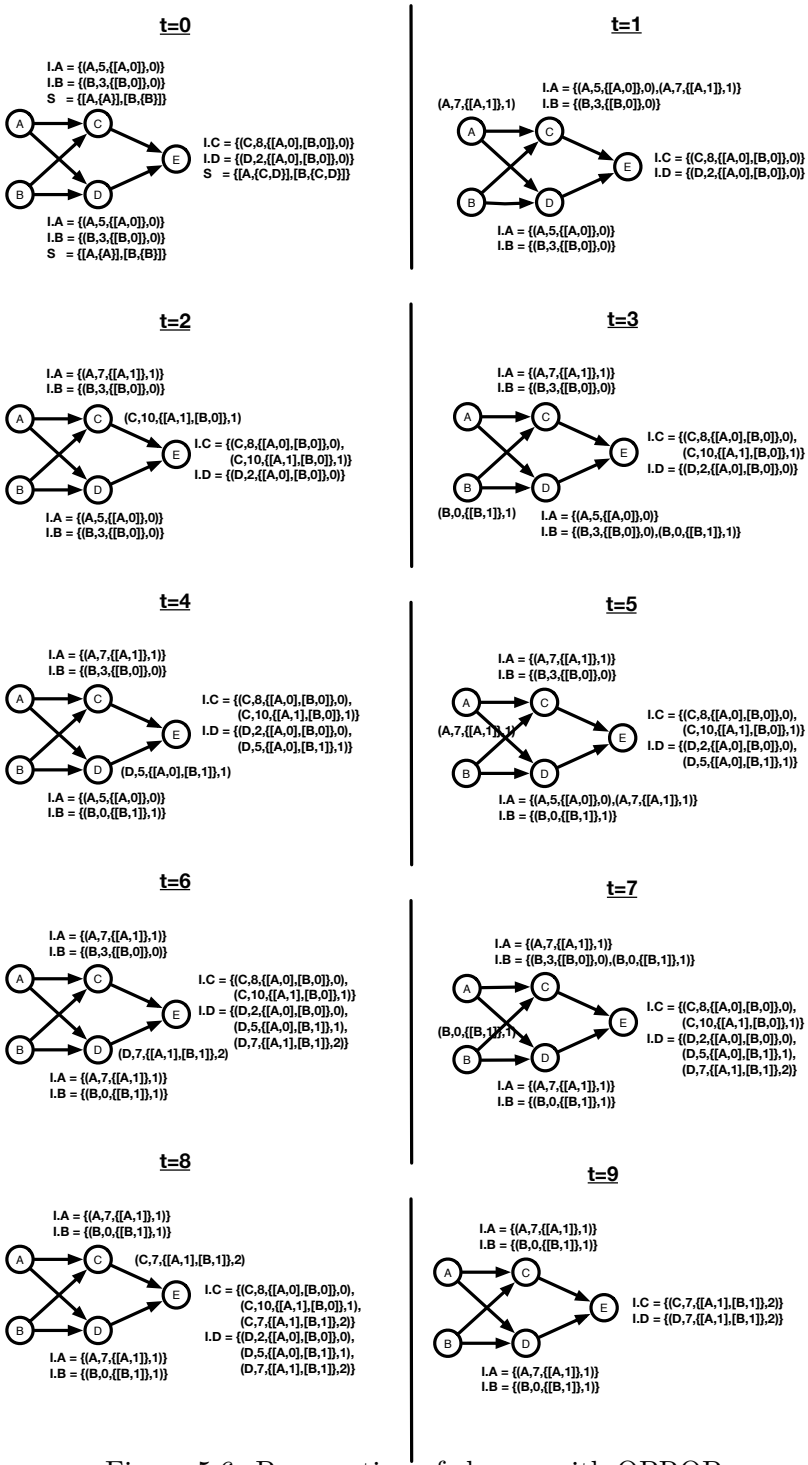


Figure 5.6: Propagation of change with QPROP.

t=2 Given that C just received a new value it calculates the cross product between $\{(A, 7, \{[A, 1]\}, 1)\}$ and $I.B$. This results in a single set of arguments namely:

- $\{(A, 7, \{[A, 1]\}, 1), (B, 3, \{[B, 0]\}, 0)\}$

This set of arguments is trivially glitch free given that $\nexists[s, preds] \in S : A \in preds \wedge B \in preds$ (i.e. the values received from A and B do not originate from a common source). C invokes its update lambda with 7 and 3 as arguments and propagates the resulting value (i.e. $(C, 10, \{[A, 1], [B, 0]\}, 1)$) to E which stores it in its $I.C$ set. C removes all values from its $I.A$ set which have an *fClock* value smaller than 1 and all values from its $I.B$ set which have an *fClock* value smaller than 0.

t=3 E calculates the cross product between $\{(C, 10, \{[A, 1], [B, 0]\}, 1)\}$ and $I.D$. This results in a single set of possible arguments for E: $\{(C, 10, \{[A, 1], [B, 0]\}, 1), (D, 2, \{[A, 0], [B, 0]\}, 0)\}$. However, this set is not glitch free given that both arguments do not have equal clock values for A. E therefore refrains from invoking its update function.

Meanwhile, B updates to 0 and propagates this new value $(B, 0, \{[B, 1]\}, 1)$ to its direct successors. At this point in time, only D receives this new value and stores it in its $I.B$ set.

t=4 D calculates the cross product between $\{(B, 0, \{[B, 1]\}, 1)\}$ and $I.A$. This results in a single set of glitch-free arguments namely:

- $\{(A, 5, \{[A, 0]\}, 0), (B, 0, \{[B, 1]\}, 1)\}$

D invokes its update function with 5 and 0 as arguments and propagates the resulting value (i.e. $(D, 5, \{[A, 0], [B, 1]\}, 1)$) to E which stores it in its $I.D$ set. D removes all values from $I.A$ which have an *fClock* value smaller than 0 and all values from $I.B$ which have an *fClock* value smaller than 1.

t=5 E calculates the cross product between $\{(D, 5, \{[A, 0], [B, 1]\}, 1)\}$ and $I.C$. This results in two sets of possible arguments for E:

- $\{(C, 8, \{[A, 0], [B, 0]\}, 0), (D, 5, \{[A, 0], [B, 1]\}, 1)\}$

- $\{(C, 10, \{[A, 1], [B, 0]\}, 1), (D, 5, \{[A, 0], [B, 1]\}, 1)\}$

This first set is not glitch free given that both arguments do not have equal clock values for B. The second set is not glitch free either given that both arguments do not have equal clock values for A nor B. E therefore refrains from invoking its update function.

Meanwhile, D receives the value propagated by A at time *t=1* and stores it in its *I.A* set.

t=6 D calculates the cross product between $\{(A, 7, \{[A, 1]\}, 1)\}$ and *I.B*. This results in a single set of possible arguments: $\{(A, 7, \{[A, 1]\}, 1), (B, 0, \{[B, 1]\}, 1)\}$. D invokes its update function with 7 and 0 as arguments and propagates the resulting value (i.e. $(D, 7, \{[A, 1], [B, 1]\}, 2)$) to E which stores it in its *I.D* set. D removes all values from its *I.A* and *I.B* sets which have an *fClock* value smaller than 1.

t=7 E calculates the cross product between $\{(D, 7, \{[A, 1], [B, 1]\}, 2)\}$ and *I.C*. This results in two sets of possible arguments for E:

- $\{(C, 8, \{[A, 0], [B, 0]\}, 0), (D, 7, \{[A, 1], [B, 1]\}, 2)\}$
- $\{(C, 10, \{[A, 1], [B, 0]\}, 1), (D, 7, \{[A, 1], [B, 1]\}, 2)\}$

This first set is not glitch free given that both arguments do not have equal clock values for A nor B. The second set is not glitch free either given that both arguments do not have equal clock values for B. E therefore refrains from invoking its update function.

Meanwhile, C receives the value propagated by B at time *t=3* and stores it in its *I.B* set.

t=8 C calculates the cross product between $\{(B, 0, \{[B, 1]\}, 1)\}$ and *I.A*. This results in a single set of glitch-free arguments namely: $\{(A, 7, \{[A, 1]\}, 1), (B, 0, \{[B, 1]\}, 1)\}$. C invokes its update function with 7 and 0 as arguments and propagates the resulting value (i.e. $(C, 7, \{[A, 1], [B, 1]\}, 2)$) to E which stores it in its *I.C* set. C removes all values from its *I.A* and *I.B* sets which have an *fClock* value smaller than 1.

t=9 E calculates the cross product between $\{(C, 7, \{[A, 1], [B, 1]\}, 2)\}$ and *I.D*. This results in three possible sets of arguments:

- $\{(C, 7, \{[A, 1], [B, 1]\}, 2), (D, 2, \{[A, 0], [B, 0]\}, 0)\}$
- $\{(C, 7, \{[A, 1], [B, 1]\}, 2), (D, 5, \{[A, 0], [B, 1]\}, 1)\}$
- $\{(C, 7, \{[A, 1], [B, 1]\}, 2), (D, 7, \{[A, 1], [B, 1]\}, 2)\}$

The first set is not glitch free given that both arguments do not have equal clock values for A nor B. The second set is not glitch free either given that both arguments to do not have equal clock values for A. However, the last set of arguments fulfils E's glitch freedom constraint. E therefore invokes its update function with 7 and 7 resulting in 14. Note that E therefore updates with twice the value of A's update at time $t=1$, as prescribed by our example. E removes all values from its $I.C$ and $I.D$ sets which have *fClock* values smaller than 2.

5.7 Supporting Dynamicity of the DAG

QPROP assumes that the dependency graph's topology does not intentionally change during a reactive application's lifetime. This assumption severely restricts the kind of application QPROP can support. For example, this prohibits the dynamic addition of new microservices to a running application.

To support such intentional topological changes we extend QPROP to QPROP$^{\text{d}}$ (i.e. dynamic QPROP). We support four dynamic operations: adding a node to and removing a node from the dependency graph and adding a dependency to and removing a dependency from the graph.

5.7.1 Dynamic Graph Changes: An Intuition

Applying dependency changes to the graph's topology requires all directly or indirectly affected nodes to update their S and I dictionaries while values are flowing through the graph. Figure 5.7(A) and (B) respectively show the state of the dependency graph before and after the addition of a new dependency between A and D. It is QPROP$^{\text{d}}$'s task to extend E's S dictionary entry for A from $[A, \{C\}]$ to $[A, \{C, D\}]$, to add A to D's S dictionary and to ensure that E now receives values originating from A through D.

QPROP$^{\text{d}}$ must guarantee glitch freedom during the addition of dependencies. Assume that source node A propagates value a_1 before the

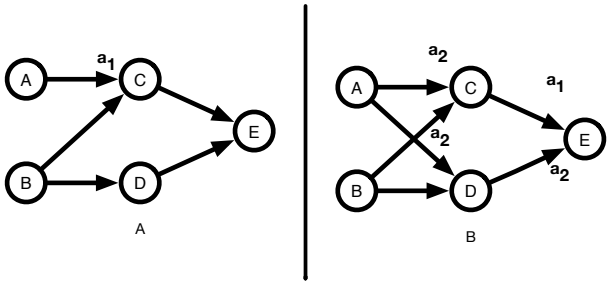


Figure 5.7: A) Dependency graph before dynamic addition of a dependency between A and D. Source A propagates value a_1. B) Dependency graph after dynamic addition of a dependency between A and D. Source A propagates value a_2.

addition of the dependency (as shown in Figure 5.7(A)). Throughout this example we omit the *sClocks* and *fClock* parts of propagation values for the sake of brevity. We also assume that B does not change throughout the example (e.g. C updates itself using a_1 and B's initial value as arguments). Due to network congestion, node E is yet to receive a_1 from C.

A propagates value a_2 after the dependency between A and D is added and all S dictionaries have been updated. a_1 finally arrives to E via C and a_2 arrives to E via D. However, dynamically adding a dependency changed E's S dictionary. As a result, E can now only update if:
$\exists arg_C \in I.C, \exists arg_D \in I.D : arg_C.sClocks.A == arg_D.sClocks.A$.
D will never propagate a_1, given that it was not a successor of A at the time. This leaves E with two options. First, E continues behaving according to the definition of QPROP. In this case E can only satisfy the aforementioned condition using a_2 values. E will never update using C's a_1 value and will remove it from its I sets instead (see the *change* Handler). In other words, the dynamic addition of a dependency resulted in the loss of a message.

A second option is for E to temporarily ignore values propagated by D. As soon as E has updated itself with C's a_1 and D's last valid propagated value (i.e. not D's a_2 value) it can resume regular propagation using values from C and D. E can safely use this value for D given that it was propagated before the addition of the dependency between A and D. We say that D is *brittle* for E. Concretely, the problem occurs when a node

must extend an already existing entry in its S dictionary. In our example, node E adds D to the $[A, \{C\}]$ entry in S. To avoid the loss of messages we choose for this latter option in QPROP$^{\mathrm{d}}$.

5.7.2 Adding Dependencies in QPROP$^{\mathrm{d}}$

ALGORITHM 3: Dynamic dependency addition

```
   arguments: A new predecessor pred
1  (predLastProp, sources) =
2        await pred ← newSucc(self)
3  DP = DP ∪ {pred}
4  if ∄source ∈ sources : [source, _] ∈ S then
5  |     I = I ∪ {[pred, {predLastProp}]}
6  end
7  await self ← addSources(pred, sources)
```

Handler newSucc(succ)

```
1  DS = DS ∪ {succ}
2  if |DP| == 0 then
3  |     return (lastProp, {self})
4  else
5  |     allSources = {s|[s, _] ∈ S}
6  |     return (lastProp, allSources)
7  end
```

Handler addSources(from,sources)

```
1  foreach source ∈ sources do
2  |     if [source, _] ∈ S then
3  |     |     S.source = S.source ∪ {from}
4  |     |     B_r = B_r ∪ {[from, {}]}
5  |     else
6  |     |     S = S ∪ {[source, {from}]}
7  |     end
8  end
9  foreach succ ∈ DS do
10 |     await succ ← addSources(self, sources)
11 end
```

For each dynamic operation in QPROP$^{\mathrm{d}}$ we provide an algorithm which is run by the node initiating the operation (e.g. a node dynamically adding a dependency to a new predecessor). Moreover, we define a number of new message handlers which extend the set of message handlers defined in Section 5.5.

A node n dynamically adding a dependency to a new predecessor *pred* runs Algorithm 3, which performs three main tasks. First, n requests the last propagated value from *pred* together with a set of all sources able to reach *pred* (line 2). By requesting this information, *pred* adds n to its list of successors (line 1 in the *newSucc* Handler). Second, if n is not brittle for *pred* (i.e. there is no overlap between the sources that can reach *pred* and those that can reach n) it creates an entry in I for *pred* (line 4). Third, n updates its own topological information (i.e. the S dictionary) and that of its direct and downstream successors. To

do so, n sends itself the *addSources* message (line 7) with *pred* and the sources which can reach *pred* as arguments. The *addSources* Handler uses these arguments to update the receiving node's S dictionary (line 2 to line 6). Moreover, the handler tracks which predecessors are brittle for n in a dictionary B_r which contains entries $[brittlePred, vals] \in B_r$, where *brittlePred* is a brittle predecessor for n and *vals* are values propagated by said predecessor. This *addSources* message is recursively sent to n's direct and downstream successors (line 10).

n immediately continues with QPROP$^{\mathrm{d}}$'s pre-propagation phase once the dependency addition operation has completed (i.e. n and all its downstream successors have updated their topological information). A barrier phase is not required here, given that values are already propagating through the system.

5.7.3 Pre-propagation

ALGORITHM 4: Pre-propagation

```
   arguments: A new value v_new = (from, value, sClocks, fClock)
 1 case ¬isBrittle(from) ∧ ¬hasBrittleSibling(from) do
 2 |   self ← change(v_new) /* Proceed with QPROP's propagation phase */
 3 case ¬isBrittle(from) ∧ hasBrittleSibling(from) do
 4 |   I.from = I.from ∪ {v_new}
 5 |   if ∀pred ∈ DP : isBrittleSibling(pred, from) ⟹ B_r.pred ≠ ∅ then
 6 |   |   self ← change(v_new) /* Proceed with QPROP's propagation phase */
 7 |   |   foreach pred ∈ DP : isBrittleSibling(pred, from) ∧ synchronised(pred) do
 8 |   |   |   MoveToI(pred)
 9 |   |   end
10 |   end
11 case isBrittle(from) do
12 |   B_r.from = B_r.from ∪ {v_new}
13 |   if |B_r.from| == 1 then
14 |   |   if synchronised(from) then
15 |   |   |   MoveToI(from)
16 |   |   |   self ← change(v_new) /* Proceed with QPROP's propagation phase */
17 |   |   else
18 |   |   |   foreach pred ∈ DP : isBrittleSibling(from, pred) do
19 |   |   |   |   foreach val ∈ I.pred \ {I.pred.first()} do
20 |   |   |   |   |   /* Run Algorithm4 with val as input */
21 |   |   |   |   end
22 |   |   |   end
23 |   |   end
24 |   end
25 end
```

QPROP^{d} introduces a **pre-propagation** phase for all nodes. The goal of this phase is to determine whether certain predecessors cease to be brittle as the result of a node receiving a new propagation value. Consider the example depicted in Figure 5.7(B). D ceases to be brittle for E as soon as E updates using C's a_1 value as arguments. In other words, when the value with the smallest clock time for A in $B_r.D$ is one clock time bigger than the value with the smallest clock time for A in $I.C$ (we assume a_1 and a_2 to have clock times for A of 1 and 2 respectively). We say that D is a brittle sibling of C and that D has synchronised with C if the aforementioned condition holds.

Algorithm 4 defines a node n's behaviour when it receives a *change* message from a predecessor *from*. It relies on a number of predicates (i.e. *isBrittle, hasBrittleSibling, isBrittleSibling* and *synchronised*) and a function (i.e. *MoveToI*). These predicates are defined separately in Algorithm 5 below to enhance the readability of Algorithm 4. n executes Algorithm 4 before the *change* Handler. The algorithm discriminates based on the predecessor *from* propagating a value v_{new} to n:

1. *from* is not a brittle predecessor and it does not have any brittle siblings (line 1). In this case n can safely continue with QPROP's propagation phase.

2. *from* is not a brittle predecessor but it has at least one brittle sibling (line 3). In other words, n has another predecessor *pred* which is brittle and which shares a common predecessor with *from*. The algorithm first checks whether all brittle siblings of *from* have at least propagated one value. Assume a brittle predecessor $pred_{brittle}$ has not yet propagated a value (i.e. $B_r.pred_{brittle}$ is empty). It is impossible for n to assess whether $pred_{brittle}$ is already synchronised or whether it is still brittle and n can therefore not update itself safely. If all brittle siblings have at least propagated one value, n is able to try and update itself with v_{new}. Subsequently the algorithm checks whether certain predecessors have ceased to be brittle as a result of this possible update. If a predecessor has ceased to be brittle (indicated by the *synchronised* predicate) the *MoveToI* function is invoked which copies the predecessor's propagation values from B_r to I and removes it from B_r.

3. *from* is a brittle predecessor (line 11). The algorithm starts by checking whether v_{new} is the first message received from *from* (i.e. $|B_r.from| == 1$). If this is the case it can be that *from* is already synchronised (line 14) in which case the *MoveToI* function is invoked and n is able to safely update. This would be the case in our example if D would propagate a_1 due to A only propagating a_1 after the dynamic dependency addition. If *from* is not synchronised but v_{new} is the first message received from *from* (line 18) the algorithm needs ensure that *from*'s non-brittle siblings have no unprocessed values. Concretely, when $B_r.from$ is empty all values propagated by its non-brittle siblings are stored in I without being processed (line 4). Therefore, when n receives the first value from *from* the algorithm needs to process these unused values.

ALGORITHM 5: Pre-propagation Predicates and MoveToI

```
1  Predicate isBrittle(pred) = [pred, _] ∈ B_r
2  Predicate hasBrittleSibling(pred) = ∃[s, dps] ∈ S, ∃pred_brittle ∈ dps :
3            pred ∈ dps ∧ isBrittle(pred_brittle)
4  Predicate isBrittleSibling(pred_brittle, pred) = isBrittle(pred_brittle) ∧ ∃[s, dps] ∈ S :
5            pred_brittle ∈ dps ∧ pred ∈ dps
6  Predicate synchronised(pred_brittle) = ∀pred ∈ DP, ∀[s, dps] ∈ S :
7            isBritle(pred_brittle) ∧ pred ∈ dps ∧ pred_brittle ∈ dps ⟹
8            B_r.pred_brittle.first().sClocks.s − I.pred.first().sClocks.s ≤ 1
9  Function MoveToI(pred_brittle):
10   if [pred_brittle, _] ∈ I then
11     I.pred_brittle = I.pred_brittle ∪ B_r.pred_brittle
12   else
13     I = I ∪ {[pred_brittle, B_r.pred_brittle]}
14   end
15   B_r = B_r \ {[pred_brittle, _]}
16   return
17
```

Algorithm 5 contains the predicates and the *MoveToI* function omitted from Algorithm 4 for the sake of brevity. It is important to note that the predicates are to be considered as macros (i.e. they expand when the node runs Algorithm 4). In other words, the free variables in the predicates (e.g. B_r in *isBrittle*) are bound to the corresponding node elements upon expansion.

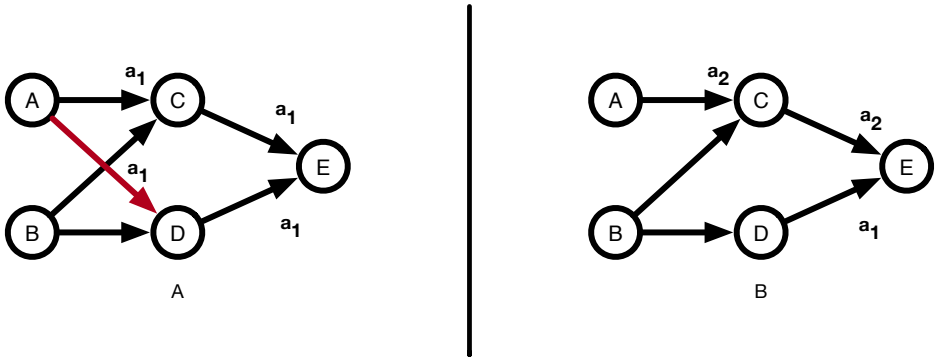


Figure 5.8: (A) Dependency graph before dynamic removal of a dependency between A and D. Source A propagates value a_1.(B) Dependency graph after dynamic removal of a dependency between A and D. Source A propagates value a_2.

5.7.4 Removing Dependencies in QPROP$^{\mathbf{d}}$

Figure 5.8(A) and (B) respectively show the state of a dependency graph before and after removing the dependency between A and D. As is the case for the addition of a new dependency (see Section 5.7.1), QPROP$^{\mathrm{d}}$'s task is to guarantee glitch freedom during the removal operation while values are propagating through the dependency graph.

Assume A propagates a_1 before the dependency between A and D is removed. Furthermore, assume E is able to update itself using these two values (see Figure 5.8(A)). At this point E's S dictionary contains two entries: $[A, \{C, D\}]$ and $[B, \{C, D\}]$ In Figure 5.8(B) the dependency between A and D has been removed. This entails that the A entry in E's S dictionary now looks as follows: $[A, \{C\}]$. Assume A propagates a_2 to C (given that D is no longer a successor of A). According to the *change* Handler (see Section 5.5.4), E now only needs to ensure that it uses arguments for which the B clock times are equal. This would lead E to update itself using a_1 from D and a_2 from C, which constitutes a glitch.

This problem arises whenever a node must modify an existing entry in its S dictionary. In our example, E removes D from $[A, \{C, D\}]$. In other words, all values stored by E in $I.D$ have become stale given that D no longer propagates values which originate from A. To avoid this issue, E must therefore remove all values received from D in $I.D$.

ALGORITHM 6: Dynamic dependency removal

```
   arguments: A predecessor to remove pred
1  sources = await pred ← remSucc(self)
2  I = I \ {I.pred}
3  DP = DP \ {pred}
4  await self ← remSources(pred, sources)
5  if |DP| == 0 then
6  |   foreach succ ∈ DS do
7  |   |   await
   |   |     succ ← addSource(self, self)
8  |   end
9  end
```

Handler addSource(from,source)

```
1  if [source, _] ∈ S then
2  |   S.source = S.source ∪ {from}
3  else
4  |   S = S ∪ {[source, {from}]}
5  end
6  foreach succ ∈ DS do
7  |   await succ ← addSource(self, source)
8  end
```

Handler remSources(from,sources)

```
1  removed = {}
2  foreach source ∈ sources do
3  |   S.source = S.source \ {from}
4  |   if S.source == ∅ then
5  |   |   S = S \ {[source, {}]}
6  |   |   removed = removed ∪ {source}
7  |   else
8  |   |   I.from = {}
9  |   end
10 end
11 foreach succ ∈ DS do
12 |   await succ ←
13 |         remSources(self, removed)
14 end
```

Handler remSucc(succ)

```
1  DS = DS \ {succ}
2  if |DP| == 0 then
3  |   return {self}
4  else
5  |   allSources = {s|[s, _] ∈ S}
6  |   return allSources
7  end
```

A node n dynamically removing a dependency to a predecessor *pred* runs Algorithm 6. In a first step, n informs *pred* that it is removing the dependency by sending the *remSucc* message. Furthermore, n removes *pred*'s input set from I and updates its direct predecessors DP. Upon receiving the *remSucc* message (see the *remSucc* Handler) *pred* removes n from its set of direct successors (i.e. DS) and returns a set with all sources which are able to reach it. n uses this set as an argument to the *remSources* message which it sends to itself. The *remSources* Handler recursively updates the topological information held by n and all its direct and downstream successors.

For our example depicted in Figure 5.8 D sends itself the *remSources* message with A as arguments for both *from* and *sources*. As a result, D removes A from the $[A, \{A\}]$ entry in S (line 3). Given that $S.A$ is now empty, D removes the entire entry from S (lines 5 and 6). In other words, D no longer propagates values originating from A and notifies E of this fact by recursively sending the *remSources* message to E using D and A as arguments. Upon receiving the message, E removes D from $[A, \{C, D\}]$ in S. Given that E can still receive values originating from A through C

it must empty D's input set (line 8) in order to avoid the aforementioned glitch.

Once all of n's direct and downstream successors have finished updating, n needs to ensure that it didn't become a source node by removing the dependency with *pred* (i.e. *pred* was its only direct predecessor). If n did become a source, it notifies its direct and downstream successors of this fact by sending the *addSource* message. In a nutshell, the *addSource* Handler updates a node's S set based on the newly reachable source.

5.7.5 Adding and Removing Nodes in QPROP^{d}

ALGORITHM 7: Dynamic node addition

```
1 foreach pred ∈ DP do
2 |  /*Proceed with Algorithm 3*/
3 end
4 foreach succ ∈ DS do
5 |  /*succ proceeds with Algorithm 3*/
6 end
```

ALGORITHM 8: Dynamic node removal

```
1 foreach pred ∈ DP do
2 |  /*Proceed with Algorithm 6*/
3 end
4 foreach succ ∈ DS do
5 |  /*succ proceeds with Algo. 6*/
6 end
```

Dynamically Adding a node n to a dependency graph is equivalent to letting n sequentially run Algorithm 3 with each of its direct predecessors as arguments. Moreover, each of n's direct successors runs Algorithm 3 with n as argument. Similarly, removing a node dynamically from a dependency graph follows the same pattern using Algorithm 6.

In summary, QPROP^{d} relaxes one of the assumptions made by QPROP. Namely, that the dependency graph does not intentionally change once the system has passed the exploration phase. However, this relaxation comes at the price of temporarily ignoring propagation values. This is a necessary evil in order to guarantee glitch freedom in this dynamic context.

5.8 Evaluation of QPROP and QPROP^{d}

Our evaluation of QPROP and QPROP^{d} is twofold.

1. We compare the runtime **performance** of our approach to that of SID-UP [DSMM14]: the state of the art in terms of centralised reactive propagation algorithms.

2. We prove that QPROP and QPROP^{d} guarantee **glitch freedom**, eventual consistency, monotonicity and absence of progress.

We start by discussing QPROP and QPROP^{d}'s performance.

5.8.1 Performance

In order to compare our approach to SID-UP we systematically compare the runtime performance of two distributed reactive systems implemented in Spiders.js: one built atop QPROP or QPROP^{d} and one built atop SID-UP. The SID-UP algorithm as presented in [DSMM14] is implemented in Scala, we therefore conduct all benchmarks using our own implementation [1] of SID-UP in Spiders.js.

We first compare a QPROP and SID-UP implementation of our running example discussed at the start of this chapter. Subsequently we compare a larger, synthetic, application built atop QPROP, QPROP^{d} and SID-UP. We compare the approaches using the following three metrics:

Load is the amount of requests per second the system receives. Each request results in the propagation of a value through the distributed dependency graph.

Latency is the average time it takes for a single value to propagate from a given source node to a sink node.

Throughput is the amount of values which propagate from source node to sink node for a given period of time.

Processing time is the time it takes for a request to propagate to a sink node. The difference with latency is that we start measuring processing time as soon as a request has been made. In contrast, we measure latency only as soon as the request is first propagated by a source node.

Memory usage we define the *heap memory usage* as the memory used by a particular service in its allocated heap. Moreover, we define the *RSS (Resident Set Size) memory usage* as the memory used by a particular service in its allocated heap, stack and code segment. Both heap memory usage and RSS memory usage are measured through node.js' *process.memoryUsage()*[2].

[1] `https://github.com/myter/ReactiveSpiders/tree/master/src/SID-UP`
[2] `https://nodejs.org/api/process.html#process_process_memoryusage`

It is important to note that due to QPROP and QPROP^{d}'s eventually consistent nature they might perform less updates than SID-UP for a same load. In SID-UP each update of a source node is guaranteed to cause a single update for all its successors. In QPROP and QPROP^{d} concurrent updates to multiple source nodes might only cause common successors to update once. To ensure the fairness of our comparison we therefore only consider a benchmark to have finished when the given system has completely processed the given amount of load. For example, assume a benchmark which simulates a load of 100 requests per second for a total of 30 seconds. Moreover, the system used for the benchmark contains a single sink node. For both SID-UP as well as QPROP and QPROP^{d} we consider the benchmark to have finished if the system's sink node has updated 3000 times. In Section 5.8.1.3 we measure the amount of these *concurrent interactions*.

In general our benchmarks show that QPROP significantly outperforms SID-UP with regards to throughput, processing time and memory usage. This is in spite of QPROP's substantial computational complexity which is $\mathcal{O}(M^N)$ for QPROP's core (i.e. line 2 in the *change* Handler) where M is the worst-case amount of messages stored for N direct predecessors. Although it slightly under performs with regards to latency, QPROP is able to respond to client requests in a timely fashion regardless of load.

5.8.1.1 Use Case Comparison

The goal of our first benchmark is to compare both approaches in a real-world setting. We therefore compare runtime performance for our running example, which is a prototypical implementation of the production system deployed by Emixis (see Section 2.1 of Chapter 2). We measure latency, throughput, processing time and memory usage under varying loads actually measured by Emixis' production version of the fleet management application. In a nutshell the production system receives on average 45 requests per second during the weekend, 75 requests per second during the evening and 300 requests per second during daytime. We conduct the benchmarks using a setup similar to Emixis', namely an Ubuntu 14.04 server with two dual core Intel Xeon 2637 processors (2 physical threads per core) with 265 GB of RAM.

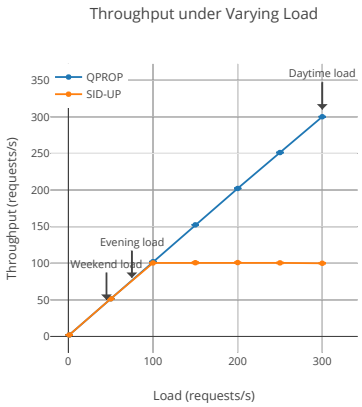


Figure 5.9: Throughput under varying loads. Error bars indicate the 95% confidence interval.

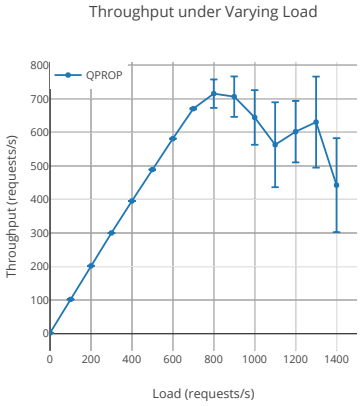


Figure 5.10: QPROP's maximum throughput for the fleet management application. Error bars indicate the 95% confidence interval.

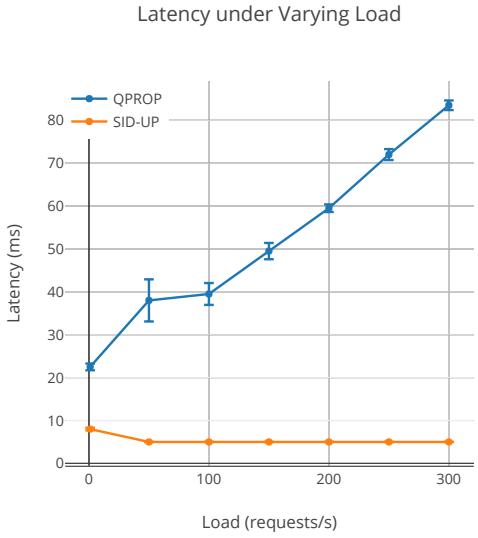


Figure 5.11: Latency under varying loads. Error bars indicate the 95% confidence interval.

Throughput

Figure 5.9 shows how both algorithms compare with regard to throughput under varying load. QPROP clearly outperforms SID-UP. SID-UP achieves its maximal throughput at 100 requests per second. If SID-UP receives more than 100 requests per second all additional requests are stored in a central buffer for delayed processing. In other words, SID-UP is unable to efficiently handle the daytime load of our fleet management application.

In order to measure QPROP's maximum throughput for our use case we vary the request load to 1400 requests per second. Figure 5.10 shows the results for this experiment. QPROP is roughly able to handle 700 requests per second, after which throughput becomes negatively affected by the increasing load.

Latency

Figure 5.11 shows how both algorithms compare with regard to latency under varying load. QPROP performs worse with regard to latency. In SID-UP a value can only propagate through the distributed dependency graph when the previous value has completed its propagation. In other words, all nodes in the distributed dependency graph are always ready to accept new values. Latency is therefore unaffected by the load. In QPROP this is not the case, values propagate through the graph concurrently. Upon receiving a new value, a node could still be processing the previous one which negatively impacts latency.

Processing

Figure 5.12 shows the request processing times for both approaches under varying load. Although QPROP suffers from a latency overhead, one clearly sees that SID-UP suffers from a much larger processing time overhead. The reason for this overhead is that a request can only be handled by SID-UP once the previous request has been handled. In contrast, QPROP allows our fleet management application to handle requests in parallel.

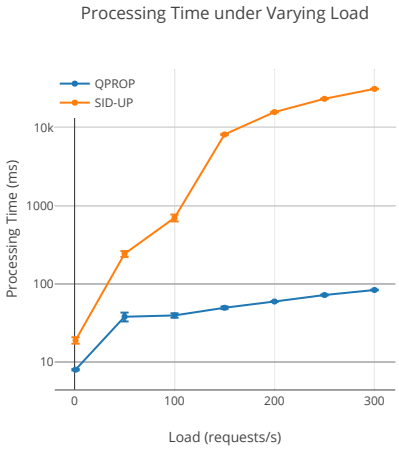


Figure 5.12: Request processing time under varying load. Error bars indicate the 95% confidence interval.

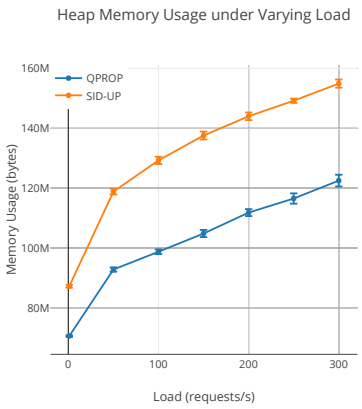


Figure 5.13: Heap memory usage across services under varying loads. Error bars indicate the 95% confidence interval.

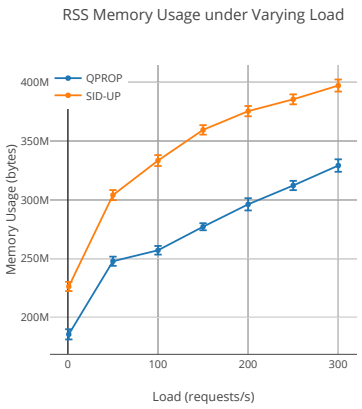


Figure 5.14: Rss memory usage across services under varying loads. Error bars indicate the 95% confidence interval.

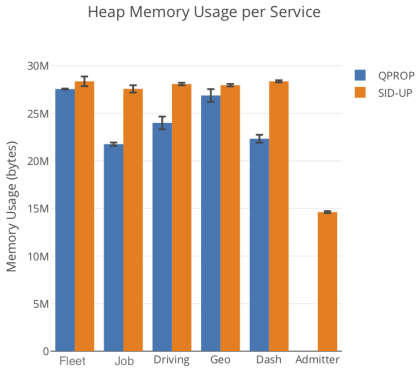


Figure 5.15: Heap memory usage per service for a load of 300 requests per second. Error bars indicate the 95% confidence interval.

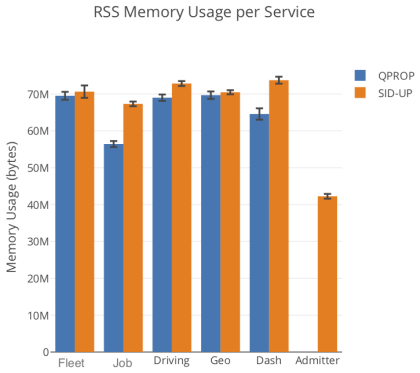


Figure 5.16: RSS memory usage per service for a load of 300 requests per second. Error bars indicate the 95% confidence interval.

Memory Consumption

Figure 5.13 and Figure 5.14 show the results for the memory measurements. SID-UP suffers from an overhead for both heap memory usage as well as RSS memory usage. To understand this overhead, consider Figure 5.15 and Figure 5.16 which detail the heap memory usage and RSS memory usage per service for a load of 300 requests per second. The services running QPROP consistently use less heap and RSS memory, although this could be attributed to implementation differences. The major difference between both approaches comes from the fact that SID-UP requires an additional *admitter* service to coordinate updates to the application.

5.8.1.2 General Comparison

The fleet management application only consists of five microservices. To further investigate the performance properties of QPROP, QPROP^{d} and SID-UP we compare the approaches using a larger example. Concretely, we implement a system comprised of 60 microservices where each service

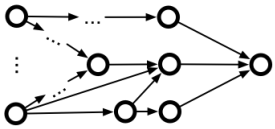


Figure 5.17: Dependency graph of the larger microservice system.

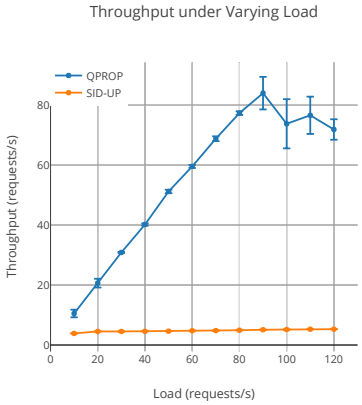


Figure 5.18: Throughput under varying loads. Error bars indicate the 95% confidence interval.

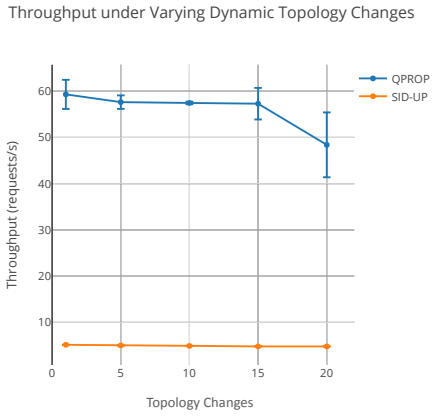


Figure 5.19: Throughput under varying dynamic topology changes. Error bars indicate the 95% confidence interval.

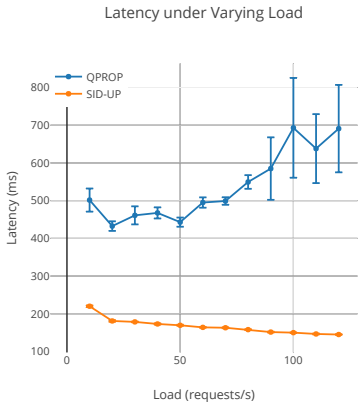


Figure 5.20: Latency under varying loads. Error bars indicate the 95% confidence interval.

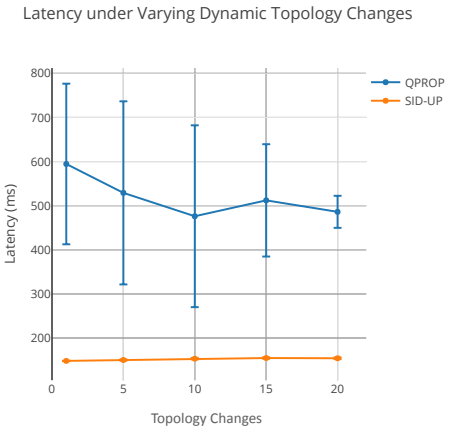


Figure 5.21: Latency under varying dynamic topology changes. Error bars indicate the 95% confidence interval.

relays the requests it receives to other services. The application's dependency graph is exemplified in Figure 5.17.

We conduct the benchmarks on a cluster of 60 Raspberry Pi 3 devices (Quad Core 1.2GHz Broadcom 64bit CPU and 1GB of RAM). Each Raspberry Pi has a 100 Mbit network port and hosts a single microservice. Figure 5.22 shows a picture of the cluster. All Raspberry Pi casings as well as the cluster's rack are hand-built using Lego bricks.

Throughput

Figure 5.18 compares the throughput of the QPROP and SID-UP implementations of the microservice systems. As is the case for the fleet management application, SID-UP is able to handle considerably less load than QPROP. SID-UP's maximum throughput is roughly 5 requests per second, while QPROP reaches its throughput peak at 85 requests per second. Although QPROP's throughput decreases after this peak, it is still roughly able to handle an order of magnitude more requests per second.

In order to compare QPROP^{d} and SID-UP we measure the impact of dynamic graph changes on their throughput. For a static load of 100 requests per second we vary the number of dynamic dependencies added

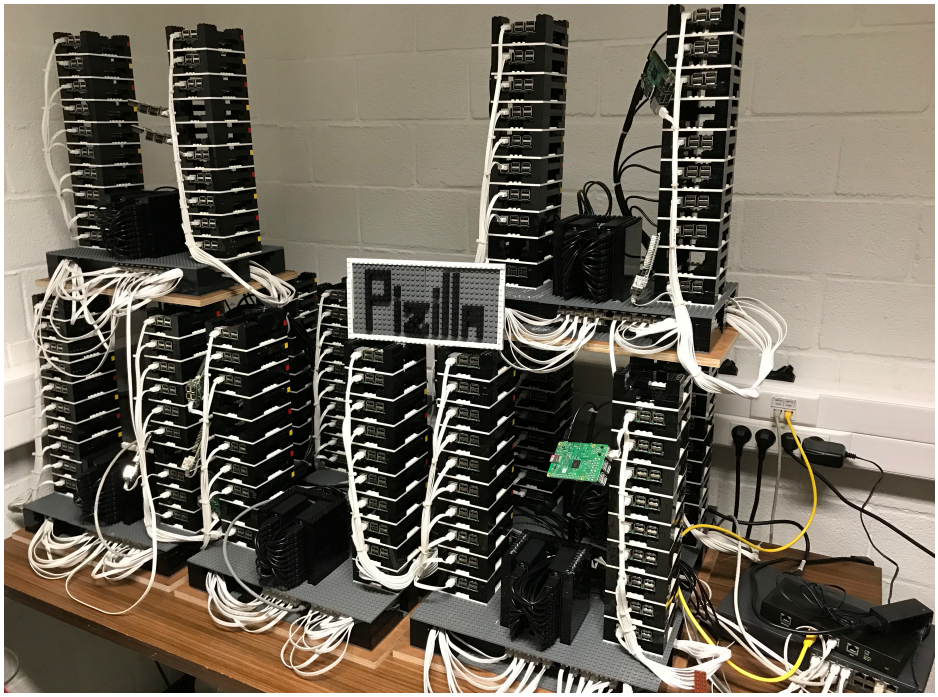


Figure 5.22: Picture of *"pizilla"*, the Raspberry Pi cluster.

to the dependency graph. Figure 5.19 shows the results of these experiments. Dynamic topology changes affect the throughput of both algorithms: QPROP's throughput roughly decreases with 20% when 20 operations are performed while SID-UP's throughput decreases with roughly 10%.

Latency

Figure 5.20 shows the latency results comparing QPROP and SID-UP. As is the case for the fleet management application, QPROP introduces an overhead with regards to latency.

Figure 5.21 shows the latency results comparing QPROP^{d} and SID-UP under a load of 100 requests per second. Dynamic topology changes only seem to impact QPROP^{d}'s latency periodically. More precisely, in QPROP^{d} a topology change will render the part of the dependency graph affected by the change unresponsive until the change completes. As a result, our benchmarks show an increase in outliers while the average latency remains roughly similar across the benchmarks. In contrast, SID-UP's latency is unaffected by load or dynamic topology changes. The reason for this phenomenon is explained in Section 5.8.1.1. SID-UP's latency is unaffected by dynamic topology changes for essentially the same reason. A change is only performed on the dependency graph whenever the previous change has completed or the previous propagation value has traversed the graph.

Processing

Figure 5.23 shows the average processing time per request. As is the case for the fleet management application, SID-UP suffers from a significant overhead as load increases. Figure 5.24 shows how processing times are affected by dynamic topology changes, both algorithms are put under a static load of 100 requests per second. The results show that topology changes hardly affect processing times.

5.8.1.3 Concurrent Interactions

QPROP and QPROP^{d} allow updates to source nodes to concurrently traverse the dependency graph. As a result it can happen that a node depending on two source nodes only updates once as a result of both source

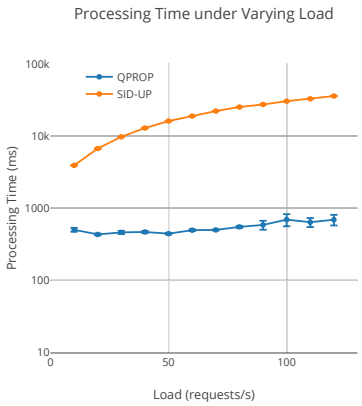


Figure 5.23: Request processing time under varying load. Error bars indicate the 95% confidence interval.

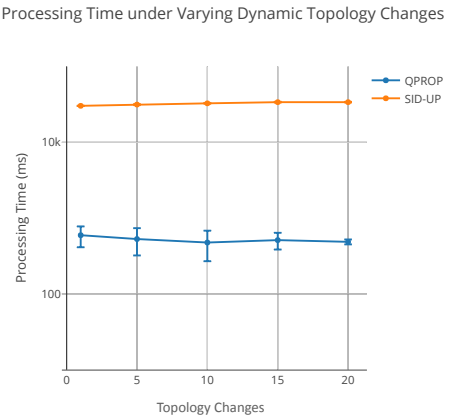


Figure 5.24: Request processing time under varying dynamic topology changes. Error bars indicate the 95% confidence interval.

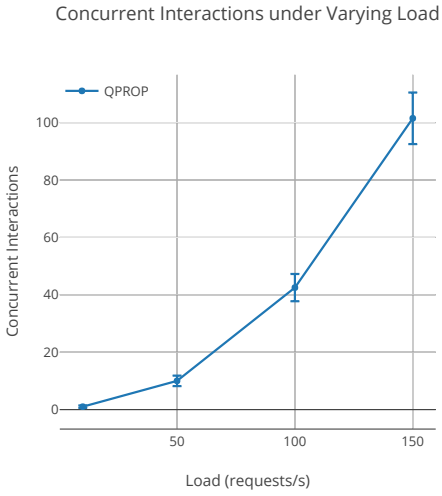


Figure 5.25: Concurrent interactions under varying loads. Error bars indicate the 95% confidence interval.

nodes updating. We call this phenomenon *concurrent interactions*. We refer the reader to Section 5.6 for an example of such concurrent interactions.

We measure the amount of concurrent interactions for the larger, synthetic, microservice system. Regular benchmarks are stopped whenever the sink nodes processed the given amount of load. For the *concurrent interactions* benchmarks we stop the benchmark as soon as the source nodes have produced the given amount of load. The difference between the generated amount of load by the source nodes and processed amount of load (i.e. updates) by the sink nodes allow us to measure the amount of concurrent interactions.

Figure 5.25 shows the amount of concurrent interactions as load increases. As the load increases the opportunities for multiple source nodes to update concurrently and for nodes in the dependency graph to receive partial updates increases. Hence, the amount of concurrent interactions increase as well.

5.8.2 Proving Glitch Freedom and Other Properties

To prove QPROP's correctness we start by proving that it guarantees glitch freedom for the applications it supports before proving its other properties. Informally, an application is glitched if it has only *partially* been updated as the result of a change in one of the event sources (i.e. one of the source nodes in the underlying dependency graph). Such a partial update is the result of an incorrect traversal of the dependency graph by the propagation algorithm. In other words, the algorithm has updated a certain node before it updated all of its predecessors in the dependency graph.
To aid us in proving[3] that QPROP is glitch free we introduce the following definitions:

[3]Our proof technique was inspired by the one employed in [DSMM14]

Definition 5: Dependency Graph

A **dependency graph** is a pair (N, E) where N is the set of nodes and E is a set of pairs denoting directed edges between nodes in N. This graph consists of three types of nodes:

- *Source* nodes :
  $\{n_{so} \in N | \nexists n \in N : (n, n_{so}) \in E\}$.
- *Intermediate* nodes :
  $\{n_i \in N | \exists n_1, n_2 \in N : (n_1, n_i) \in E \wedge (n_i, n_2) \in E\}$.
- *Sink* nodes :
  $\{n_s \in N | \nexists n \in N : (n_s, n) \in E\}$

Definition 6: Path

$P(x, y)$ denotes the existence of a **path** in a dependency graph (N, E) between nodes x and y. In other words : $P(x, y) \iff (x, y) \in E \vee \exists z \in N : P(x, z) \wedge P(z, y)$.

Definition 7: Propagation Path

The set of reachable nodes starting from a source node n_{so} is a partially ordered set: $PP_{n_{so}} = \{n_{so}\} \cup \{n \in N | P(n_{so}, n)\}$. The order of the nodes in this set is defined as follow: $\forall n_1, n_2 \in PP_{n_{so}} : n_1 > n_2 \iff P(n_1, n_2)$. We say that $PP_{n_{so}}$ is $n'_{so}s$ **propagation path**.

Definition 8: Precedence

We define that a node x **directly precedes** a node y in a dependency graph (N, E) as follows: $x >> y \iff \exists (x, y) \in E$. Similarly, we define that a node x directly succeeds a node y in a dependency graph (N, E) as follows: $x << y \iff \exists (y, x) \in E$ Note that both relations are non-transitive.

5.8.2.1 Exploration Correctness

QPROP's exploration phase guarantees that a node n's S_n dictionary contains entries which map each source node n_{so} able to reach n onto n's direct predecessors $DP_n = \{dp \in N | dp >> n\}$. More precisely:

Theorem 1: Exploration Correctness

$$\forall n \in N, \forall dp_i \in DP_n : \exists P(n_{so}, dp_i) \iff \exists [n_{so}, \{..., dp_i, ...\}] \in S_n$$

We prove this by contradiction: a node n is reachable by a source node n_{so} through a direct predecessor dp_i. However S_n lacks an entry which reflects this fact. Formally:
$\exists n \in N, \exists dp_i \in DP_n, \exists P(n_{so}, dp_i) : \nexists [n_{so}, \{..., dp_i, ...\}] \in S_n$.
This means that dp_i did not include n_{so} as an argument to the *sources* message (see Section 5.5.2). A node only sends the *sources* message if it has received the *sources* message from all its direct predecessors (see the *sources* Handler on line 10). Therefore, at least one of dp_i's direct predecessors should have included n_{so} as an argument to its *sources* message but failed to do so. Iteratively applying this reasoning would entail that n_{so} did not include itself in the *sources* message to each of it direct successors. As shown in Algorithm 1 on line 8 this cannot be the case.

5.8.2.2 Glitch Freedom

Definition 9: Glitch

Assume a node n with update lambda U_n and a set of direct predecessors $DP_n = \{dp \in N | dp >> n\}$. If v_{dp_i} denotes a value propagated to n by dp_i and $Time_{n_{so}}(v_{dp_i})$ denotes a source node n_{so}'s timestamp attached to v_{dp_i} then $U(v_{dp_1}, ..., v_{dp_{|DP_n|}})$ produces a **glitch** if:
$\exists P(n_{so}, dp_i) \wedge \exists P(n_{so}, dp_j) \wedge Time_{n_{so}}(val_{dp_i}) \neq Time_{n_{so}}(val_{dp_j})$

We refer the reader to Section 5.4 for an intuitive explanation of glitches and glitch freedom.

Theorem 2: Glitch Freedom

$\forall n \in N : U_n(v_{dp_1}, ..., v_{dp_{|DP_n|}}) \iff$
$\forall [n_{so}, \{dp_i, ..., dp_j\}] \in S_n : Time_{n_{so}}(v_{dp_i}) == Time_{n_{so}}(v_{dp_j})$

Assume a set of source nodes $N_{so} = \{n_{so_1}, ..., n_{so_n}\}$ update and propagate their new values to all their direct successors concurrently. For a glitch to occur, at least one node n in $PP_{N_{so}} = \bigcup_{i=1}^{n} PP_{n_{so_i}}$ needs to update itself with a set of values *vals* such that:
$\exists n_{so} \in N_{so}, \exists v_{dp_i}, v_{dp_j} \in vals : Time_{n_{so}} v_{dp_i} \neq Time_{n_{so}} v_{dp_j}$. A node is only able to invoke its update lambda if line 3 in the *change* Handler (see Section 5.5.4) returns a set of glitch free arguments. We discern two cases. First, n was unable to find such a set of glitch free arguments. In this case n could not have invoked its update lambda and could therefore not have caused a glitch which contradicts our assumption. Second, line 3 in the *change* Handler returned *vals* as set of glitch free arguments. In this case n can safely invoke its update lambda using *vals* as arguments without causing a glitch which contradicts our assumption.

5.8.2.3 Monotonicity

Definition 10: Monotonic update

Assume a node n with update lambda U_n, a set of direct predecessors $DP_n = \{dp \in N | dp >> n\}$. n invokes its update lambda using the following set of arguments $Args_t = \{v_{dp_1}, ..., v_{dp_{|DP_n|}}\}$. Moreover, n's clock value is t before this update happens. Later, at clock time $t + n$, n invokes its update lambda using the following set of arguments : $Args_{t+n} = \{v'_{dp_1}, ..., v'_{dp_{|DP_n|}}\}$. We say that n updates **monotonically** if and only if: $\nexists dp_i \in DP_n, v_{dp_i} \in Args_t, v'_{dp_i} \in Args_{t+n} : Time_{n_{so}}(v_{dp_i}) > Time_{n_{so}}(v'_{dp_i})$.

In other words, once n updates with a value v_{dp_i} originating from a source n_{so} it will never update with an older value originating from the same source.

Theorem 3: Monotonicity

Nodes always update monotonically.

We prove this by contradiction. Assume n updates twice: first using $Args_t$ as set of arguments to U_n and later using $Args_{t+n}$ as a set of arguments to U_n. We assume that n updated non-monotonically, in other words the following holds:
$\exists dp_i \in DP_n, v_{dp_i} \in Args_t, v'_{dp_i} \in Args_{t+n} : Time_{n_{so}}(v_{dp_i}) > Time_{n_{so}}(v'_{dp_i})$.
Given that v_{dp_i} and v'_{dp_i} are the results of dp_i's updates this means that dp_i updated non-monotonically. Applying this reasoning iteratively results in a direct successor of n_{so} updating non-monotonically. In other words, n_{so} first propagated a value $v_{n_{so}}$ and then a value $v'_{n_{so}}$ for which the following holds:
$Time_{n_{so}}(v_{n_{so}}) > Time_{n_{so}}(v'_{n_{so}})$ However, this cannot happen given that clock times only monotonically increase (line 6 in the *change* handler). Consequently, this means that none of n_{so}'s direct or indirect successors could have updated non-monotonically which contradicts our original assumption.

5.8.2.4 Eventual Consistency

Theorem 4: DAG Construction

Any DAG can be constructed by recursively adding sink nodes (i.e. nodes with an out degree of 0) starting from the empty DAG.

Every DAG has at least one topological ordering. Hence, one can construct any DAG by recursively adding nodes in the order for which they appear in the DAG's topological ordering. By the definition of topological ordering this entails that each node is added to the DAG before any of its successors. In other words, in each recursive step a node is added with out degree 0.

Definition 11: Graph Consistency

A distributed dependency graph is **consistent** if the following holds: $\forall n_{so} \in N, n \in PP_{n_{so}} : Time_{n_{so}}(lastProp(n)) = clock(n_{so})$.We denote the value of source node n_{so}'s *clock* with $clock(n_{so})$ and we denote the last propagated value by n with *lastProp(n)*.

In other words, all nodes n in a source node n_{so}'s propagation path must have witnessed its last update.

Theorem 5: Eventual Consistency

If all source nodes stop propagating new values, eventually the dependency graph reaches consistency.

We prove this by structural induction over the dependency graph (i.e. a DAG constructed by recursively adding sink nodes starting from the empty DAG). The *induction base* considers a dependency graph containing a single node. This node is trivially consistent with itself. The *induction hypothesis* is that a given dependency graph n reaches consistency when all of its source nodes stop propagating new values. The *induction step* extends this graph n with a new *sink* node n_{new}. We do this by adding an arbitrary amount of edges from an arbitrary amount of nodes in n to the new *sink* node. We prove by contradiction that this new graph is eventually consistent. Assume that all source nodes have stopped propagating updates and that our graph is inconsistent. Given our hypothesis this can only mean that n_{new} causes the inconsistency. In other words: $\exists n_{so} \in N : n_{new} \in PP_{n_{so}} \wedge Time_{n_{so}}(lastProp(n_{new})) \neq clock(n_{so})$. There are two possible reasons for this:

First, n_{new} was unable to update using its direct predecessors' last propagated value without causing a glitch. In other words, at least two of these values have a different *sClock* timestamp for n_{so}. However, the induction hypothesis ensures that n_{new}'s direct predecessors have witnessed n_{so}'s last update. Therefore, it is impossible for at least two of n_{new}'s direct predecessor to propagate values with a different *sClock* timestamp for n_{so}.

Second, n_{new} was able to update itself using its direct predecessors' last propagated value, yet it still causes the graph's inconsistency. Through the *change* Handler (see Section 5.5.4) we know that n_{new}'s last propagated value's *sClocks* is a union of the *sClocks* of all values received from n_{new}'s direct predecessors. By definition this means that the following holds:

$$\forall dp \in \{dp \in N | dp >> n\} : Time_{n_{so}}(lastProp(dp)) = Time_{n_{so}}(lastProp(n_{new}))$$

Moreover, the induction hypothesis ensures that the following holds:
$\forall dp \in \{dp \in N | dp >> n\} : Time_{n_{so}}(lastProp(dp)) = clock(n_{so})$.
Therefore, $Time_{n_{so}}(lastProp(n_{new})) = clock(n_{so})$ which contradicts our original assumption.

5.8.2.5 Progress

Informally, a distributed system makes progress if it performs useful computations towards termination [HSH05]. In our case we define this termination as follows:

Definition 12: Update Completion

Assume a source node n_{so} which updates and propagates a new value $v_{n_{so}}$. The update which caused n_{so} to propagate $v_{n_{so}}$ is said to **complete** if eventually:
$\forall n \in PP_{n_{so}} : Time_{n_{so}}(lastProp(n)) = Time_{n_{so}}(v_{n_{so}})$

In other words, we say that a distributed reactive system provides progress if (concurrent) updates are guaranteed to finish in a finite amount of time. QPROP and QPROP^{d} are therefore unable to guarantee progress given that both suffer from livelocks (we discuss this in Section 5.9). In a nutshell, infinite concurrent updates to at least two source nodes can cause livelocks for common successors to said source nodes. However, assuming that concurrent updates stop, both algorithms are able to guarantee the completion of at least the last update to each source node. This trivially follows from the proof on eventual consistency (see Section 5.8.2.4).

5.8.2.6 Dynamic Graph Changes

Essentially, QPROP^{d} provides two dynamic operations: adding and removing a dependency between nodes. Adding and removing a node from the dependency graph are sequences of dependency additions and removals. We therefore prove the correctness of dependency addition and removal.

Dynamic Dependency Addition

We refer the reader to Section 5.7 for a detailed explanation on the issue which can arise upon dynamically adding a dependency between two nodes in the dependency graph. Assume a node n_1 part of a source node's n_{so}'s propagation path $PP_{n_{so}}$. Moreover, assume n_{so} propagates values $V_{before} = \{v_1, ..., v_n\}$ to its successors. A dependency is dynamically added from a node n_2 to n_1, which therefore adds n_2 to $PP_{n_{so}}$. After this, n_{so} propagates values $V_{after} = \{v_{n+1}, ..., v_{n+n}\}$ to its successors However, n_2

will never receive values from V_{before} given that these were propagated before it joined n_{so}'s propagation path. QPROP$^{\mathrm{d}}$ must therefore ensure that all nodes in $PP_{n_{so}}$ update using values from V_{before} before updating using values from V_{after}.

We prove this by contradiction. Assume a node n_3 in $PP_{n_{so}}$ updates using values from V_{after} without first updating using values from V_{before}. Given that nodes propagate values in order, this can only mean that $\exists P(n_2, n_3)$ (i.e. all other nodes in $PP_{n_{so}}$ will first propagate values from V_{before}). According to Algorithm 3 n_3 must have received the *addSources* message from at least one of its predecessors $n_{3_{pred}}$. Moreover, $n_{3_{pred}}$ must be a direct successor of n_2 and must have added n_2 to its B_r dictionary. Line 11 in Algorithm 4 ensures that $n_{3_{pred}}$ will only update itself using the first value of V_{after} (i.e. v_{n+1}) if it previously updated itself with the last value of V_{before} (i.e. v_n). Hence, n_3 can impossibly receive values from V_{after} before receiving values from V_{before}.

Dynamic Dependency Removal

We refer the reader to Section 5.7.4 for a detailed explanation on the issue which can arise upon dynamically removing a dependency between two nodes in the dependency graph. Dynamically removing a dependency between a node n_2 and its predecessor n_1 changes n_2's S dictionary as well as the S dictionaries of all its successors. QPROP$^{\mathrm{d}}$ must therefore ensure glitch freedom while nodes update their topological information.

A node n only updates its topological information as a result of receiving the *remSources* message (see the *remSources* Handler). We discern two cases. First, n removes an entry for a source node n_{so} from S (line 5 in the *remSources* Handler). In this case, n only had a single predecessor propagating values which originate from n_{so} and can therefore not produces glitches. Second, n removes a predecessor *pred* from an entry for a source node n_{so} which still contains other predecessors *preds*. However, in this case n removes all values for *pred* from I. All subsequent values n will receive from *pred* will no longer originate from n_{so} and can therefore not cause glitches.

5.9 Limitations of QPROP

As we prove in the previous section, QPROP guarantees eventual consistency. Assuming that source nodes stop propagating new values, all nodes in the graph eventually update using their predecessors' last values. However, nodes in QPROP can become subject to livelocks.

For example, reconsider Figure 5.5(C). Node E's glitch freedom constraint goes as follows:

$$U_E(val_C, val_D) \iff Time_A(val_C) == Time_A(val_D) \wedge Time_B(val_C) == Time_B(val_D)$$

However, it is possible that E never receives a pair (val_C, val_D) for which this holds. Assume that A and B update concurrently. Due to interleaving of messages it can be that C invokes its update function using the new value for A and B's old value as arguments. Furthermore, D invokes its update function using the new value for B and the old value for A as arguments. Upon receiving these values, E will not be able to meet its glitch freedom constraint. Assume that A and B infinitely update concurrently and this exact interleaving of messages continues. In this case E is never able to update without causing glitches and is therefore stuck in a livelock. However, E resolves this livelock as soon as A or B stop updating. In general, nodes in QPROP can livelock for graph topologies where two or more source nodes (e.g. A and B) all propagate values to a single node in the graph (e.g. D) via two or more overlapping paths.

QPROP's livelocks resemble the *"duelling proposers"* [Lam98] scenario for Paxos (i.e. two nodes alternately increase proposal numbers). Future work will focus on assessing whether randomisation [Lam98, OO14] (e.g. letting nodes await a random sleep timeout before handling a *change* message) could alleviate the livelock issue in practice.

5.10 Related Work

Derivations are directly inspired from abstractions traditionally found in reactive programming languages. As a concept derivations do not specifically contribute to the state of the art in reactive programming. However, QPROP is the first decentralised propagation algorithm to guarantee glitch freedom. This section discusses the most prominent approaches in glitch prevention for reactive applications.

In Elm [CC13] each node in the dependency graph runs under its own thread of control. Moreover, each node has a number of queues which hold values propagated by predecessors. However, Elm relies on a *global event dispatcher* to provide new values to source nodes and is therefore not suited to handle derivations in DRIAs.

A number of non-distributed reactive programming languages have recently been researched (e.g. FrTime [CK06] and Flapjax [MGB+09]). These languages topologically sort the dependency graph underlying reactive applications to guarantee glitch freedom. A similar approach is taken by synchronous reactive programming languages, such as Esterel [BG92], which rely on a scheduler to determine the order in which values propagate through the dependency graph. This approach does not fit the decentralised distributed systems we target.

Globally asynchronous locally synchronous(GALS) [BCLG00] systems provide a distributed approach to synchronous reactive programming. GALS discriminate between two kinds of systems: *exochronous* and *endo/isochronous* [LGTLL03]. Endo/isochronous systems are defined by the fact that the system only relies on the values of signals, never on the presence or absence of the value of a signal. Concretely, this constraints a system to having a number of input signals for which one can infer at what pace they produce values. Using this classification, QPROP explicitly targets exochronous systems where multiple source signals might produce values at different and varying rates. To our knowledge current GALS systems solely target endo/isochronous systems [GG10, PBCB06]. Although exochronous systems can be *endochronised*, this entails the addition of a centralised monitor or master clock to obtain the presence or absence of a signal value [BCLG00].

Quality-aware reactive programming (QUARP) [PB17] abstracts away the notion of glitches to a more general notion of *propagation quality*. This allows QUARP to implement decentralised glitch freedom as well as other propagation criteria (e.g. geographical location of nodes). QUARP and QPROP fundamentally differ in the glitch freedom guarantees they provide. In a nutshell, QUARP nodes only maintain the last propagated value for each of their predecessors. This value is overwritten each time the node receives a value from its associated predecessor. Nodes in QUARP can therefore livelock for *any* graph topology able to cause glitches (see Figure 5.5(A)). In contrast, QPROP nodes can never livelock for the topology

shown in Figure 5.5(A). Moreover, QUARP does not guard against the issues which arise from dynamic topology changes (see Section 5.7).

The work presented in [SSR17] extends the synchronous reactive programming language Céu [SRI+13] with support for GALS systems. However, as stated in [SSR17] it does not guarantee glitch freedom.

SID-UP [DSMM14] is a propagation algorithm specifically designed towards the development of decentralised distributed systems. However, SID-UP requires a central coordinator to support systems where source nodes update concurrently. This feature of SID-UP makes it unfit to deal with the reactive systems we envision.

Burckhardt et al. present a reactive caching algorithm in [BC18]. In a nutshell, this algorithm caches the return values of requests across (micro)services in a distributed application. These caches are reactively updated whenever updates to the system's state render them invalid. The novelty of this approach is that it leverages an intuitive pull-based API with push-based update semantics. However, the approach does not guarantee glitch freedom.

ScalaLoci [WKS18] is a multitier distributed reactive programming language. It allows programmers to explicitly specify the placement of pieces of the application's distributed state (i.e. on which nodes in the network the data should be kept). Moreover, ScalaLoci provides two forms of derivable data (i.e. signals and events) which can span across nodes in the network. ScalaLoci's reactive abstractions therefore closely resemble Triumvirate's derivations. However, ScalaLoci does not guard applications against glitches.

Spreadsheet applications such as Microsoft's Excel essentially allow one to write asynchronous reactive code. Each cell in the spreadsheet can be seen as a signal which can depend on the values contained in other cells. Moreover, given Excel's multithreaded capabilities [Cap14] these updates can happen concurrently. A generalisation of this model to stream processing was introduced by [VTR+14]. However, glitches are avoided in this model by analysing the dependencies between cells. Such an analysis cannot be employed in a distributed context without resorting to a centralised entity. Moreover, this analysis would need to be re-run upon dynamic changes to the dependency graph incurring a substantial overhead.

AmbientTalk/R [CMVCDM10] is a reactive extension to the AmbientTalk [CGS+14] language. However, AmbientTalk/R only guards against local glitches (i.e. only signals residing on the same physical device are updated glitch freely).

5.11 Derivables and Requirements for a DRIA-Oriented Programming Model

Derivations allow Triumvirate to fulfil requirements R_2 and R_3 for DRIA-oriented programming models as follows:

R_2: The model guarantees data consistency properties specified by the programmer
Triumvirate programmers use derivations to implement reactive state. Our QPROP algorithm ensures that this reactive states updates without causing glitches. Moreover, QPROP guarantees the eventual consistency of derivations.

R_3: The model supports multiple parameter passing semantics
Derivations provide pass-by-derivation parameter passing semantics. In other words, a distributed dependency graph is created as derivations are sent across the network. Updates to the sources of this dependency graph automatically trigger updates for all derivations in the graph.

Together with replicas and duplicates, derivations enable Triumvirate to completely meet requirements R_2 and R_3 for DRIA-oriented programming models.

5.12 Chapter Summary

Part of the state in DRIAs is distributed across the network *by derivation*. These derivation provide reactive update semantics. Whenever a part of the application's state changes all its derivations automatically update as well.

In Triumvirate programmers implement this kind of distributed state using *derivations*. Derivations are heavily inspired by the reactive programming paradigm. In a nutshell, programmers create derivations by

applying a derivation function to a piece of distributed state (i.e. a duplicates, replicas or other derivations). The return value of this derivation function is a new derivation. Whenever one of the input arguments updates the derivation is recomputed in order to update the state of the output derivation. These semantics apply regardless of the input arguments locality (i.e. the input arguments can reside on different actors).

The Triumvirate runtime ensures that updates propagate through the dependency graph of derivations. More concretely a propagation algorithm traverses this graph in topological order to perform these updates. This ordering of updates is fundamental to the correctness of Triumvirate applications using derivations. State of the art reactive propagation algorithms are unable to guarantee this correctness property without resorting to centralised coordination, which goes against the nature of Triumvirate applications.

In this chapter we introduce a novel propagation algorithm QPROP, and its dynamic extension QPROP$^{\mathrm{d}}$. This propagation algorithm is able to guarantee glitch freedom in distributed reactive applications without resorting to centralised coordination.

Chapter 6

A Platform for Distributed Web-oriented Programming Languages

The previous chapters discuss Triumvirate in general and replicable and derivable state in detail. Triumvirate is implemented as a DSL atop TypeScript, a typed superset of JavaScript. More precisely, Triumvirate is built atop a TypeScript library that we developed for distributed and parallel web-programming called *Spiders.js* [MSDM16, MSDM18b]. Triumvirate's central idea, the classification of distributed state into three categories, only fulfils requirements R_2 and R_3 for a DRIA-oriented programming model: duplicates, replicas and derivations allow programmers to express complex parameter passing and data update semantics for their distributed state. Triumvirate relies on Spiders.js' actor model and meta-programming abstractions to fulfil the remaining two requirements (i.e. R_1 and R_4).

In a nutshell, Spiders.js provides a unit of distributed logic (i.e. actors) and allows programmers to extend its built-in parameter passing and update semantics. Triumvirate can be thought of as an organisational layer on top of Spiders.js. Figure 6.1 highlights the concepts discussed in this chapter.

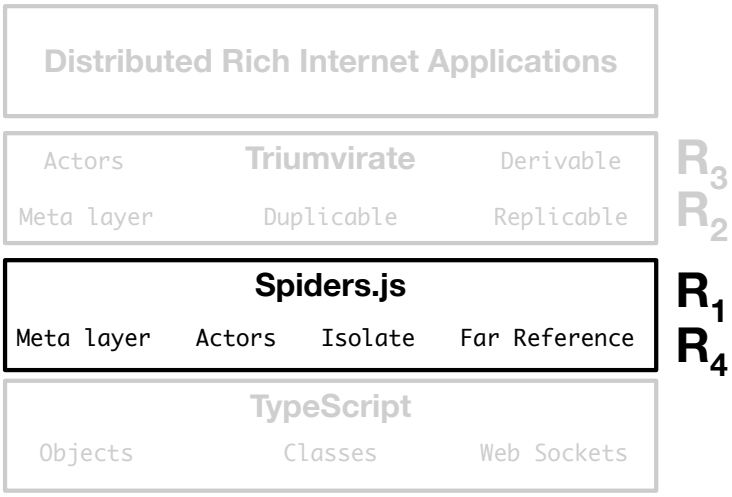


Figure 6.1: Concepts discussed in this chapter.

6.1 The Lack of a Distributed Programming Model for the Web

The actor model, and more specifically communicating event-loops (CEL, see Section 2.3.1.1 of Chapter 2), inherently fits the web given that JavaScript engines are essentially implemented as CEL actors [ecm19]. In other words, a JavaScript engine maintains a heap of objects and a message queue containing events. These events are sequentially removed from the queue and drive the application logic. As such, web applications are internally implemented as actor systems.

From a programmer's perspective JavaScript provides two actor-like constructs: *web workers* and *child processes*. The former are only available within browser environments (i.e. the client-side of web applications) while the later is available for server-side applications (i.e. applications written in Node.js, which is the most prominent server-side JavaScript implementation). Unfortunately, these two constructs exhibit three major shortcomings:

Location Transparency One of the strengths of the actor model is that actors are location agnostic. The semantics of an actor (e.g. parameter passing semantics) do not depend on an actor's location (i.e. whether they reside on the same machine or not). Actors in

JavaScript break this transparency property on two levels. First, the API used by actors is different per tier (i.e. server or client side). Client-side actors (i.e. web workers) employ *HTML5* message channels, while communication between server-side actors (i.e. child processes) is traditionally implemented using web sockets. Second, JavaScript actors lack the native constructs to communicate across single machine boundaries. This burdens the programmer with the task of providing communication between server and client-side actors.

Coarse-grained parameter passing semantics Programmers are unable to specify how values are sent from one actor to another. A value is first copied before it is passed between actors. However, this is only guaranteed for primitive data types (e.g. numerals, strings). All other data types (i.e. functions, object methods) must be serialised manually by the programmer which quickly leads to bugs. For example, manual serialisation of objects forces the programmer to take care of possible scoping issues (e.g. a method which refers to variables defined in its lexical scope).

Client-side actors support additional parameter passing semantics which *transfers* objects between actors. Transferring an object between actors can be compared to pass-by-reference semantics where the sender loses its reference to the transferred object. However, this feature is primarily used for performance reasons, as it only applies to a limited number of objects (i.e. *ArrayBuffer*, *MessagePort* or *ImageBitmap*).

Second class actor references An actor is able to spawn other actors. We say that the spawning actor is the *parent* of its spawned *children* actors. Message sends between actors are natively supported only between such parent and child actors. A parent actor obtains a reference to its children upon spawning them. This reference allows the parent to send messages to its children and vice versa. However, JavaScript disallows such references to be copied between actors (e.g. as part of a message send).

This burdens the programmer with the task of implementing arbitrary actor-to-actor communication. For client-side actors this typically entails the use of *HTML5* message channels (i.e. a tuple

of ports), which creates a communication channel between two web workers. In a nutshell, both actors must obtain a port of the same message channel to be able to send messages. For server-side actors one traditionally achieves communication between arbitrary actors through the use of web sockets.

In this chapter we present our *solution* to the aforementioned shortcomings of actors in JavaScript: Spiders.js. Spiders.js is a framework which solves the problems of built-in JavaScript actors as follows:

- Spiders.js exposes a **uniform API and semantics** regardless of the tier (i.e. client or server) in which it is used. Moreover, Spiders.js' underlying parameter passing system can handle both *vertical* as well as *horizontal* distribution. The former allows for traditional client-to-server communication while the latter allows for server-to-server and client-to-client communication.
- Programmers are freed from the burden of manually serialising objects. **Objects are 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.
- Actor references are **first class values**. This entails that all actors are able to exchange references between and send messages to each other.

6.2 A Collaborative Code Editor

To discuss Spiders.js' constructs we make use of an example application called *CoCode*[1]: a collaborative code editor. In a nutshell, the application is a browser-based code editor that offers syntax highlighting and that synchronises the code amongst collaborators.

Figure 6.2 shows a screenshot of CoCode in use. CoCode offers two functionalities. First, programmers enter their code in the text area at the top of the screen. Code is highlighted and synchronised across clients by pressing on the *"commit"* button. Second, programmers are able to send

[1] `https://gitlab.soft.vub.ac.be/fmyter/cocode`

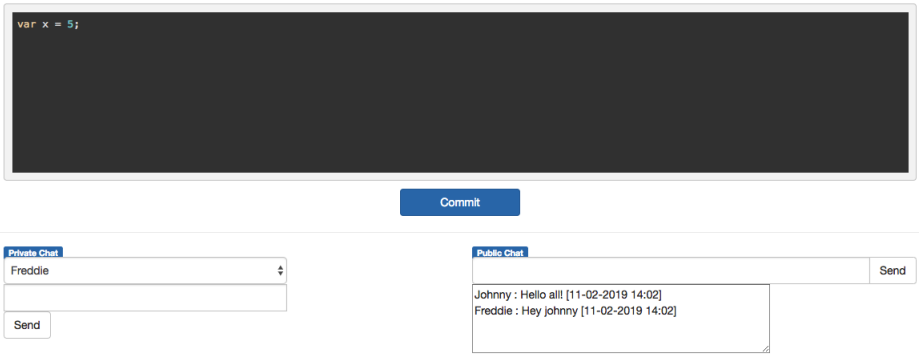


Figure 6.2: Screenshot of CoCode in use.

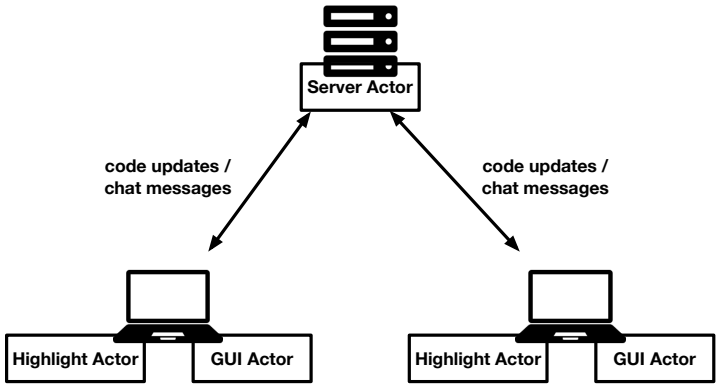


Figure 6.3: Architecture of CoCode.

private messages to each other (using the input field and button on the left side of the screen) or chat publicly (using the text area and button on the right side of the screen).

Figure 6.3 gives an overview of CoCode's architecture. CoCode clients are implemented using two actors: A *GUI Actor* and a *Highlight Actor*. The former is responsible for updating the user interface and communicating with the CoCode server. The latter is responsible for syntax highlighting the code provided by the user. Highlighting the code in a separate actor allows CoCode's user interface to remain responsive while the user is coding.

Class	**Functionality**
Application	Runs on the main thread
	Has access to the window object (client-side)
	Serves as an actor factory
Actor	Runs under its own thread of control
Isolate	Adheres to pass by copy semantics
Object	Adheres to pass by far reference semantics
Promise	Represent asynchronous computations

Table 6.1: Spiders.js' Base-level Constructs.

The CoCode server allows clients to initially download their actors' and user interface's source code. Moreover, the server is responsible for both synchronising the code base that is collaboratively edited by CoCode users as well as for delivering chat messages between users.

We start by giving a brief overview of the constructs provided by Spiders.js before explaining the implementation of CoCode using said constructs.

6.3 An overview of Spiders.js

Table 6.1 provides an overview of the constructs provided by Spiders.js. In a nutshell, Spider.js provides two abstractions to represent distributed logic (i.e. *actor* and *application*), two to represent distributed state (i.e. *isolate* and *far reference*) and one to represent asynchronous computation (i.e. *promise).*

Distributed Logic Spiders.js programmers implement distributed logic using CEL actors (see Section 2.3.1.1 of Chapter 2). These are defined by extending one of two classes: *Application* or *Actor*. The application actor is spawned by instantiating an object from its class definition (i.e. using JavaScript's *new* operator), which can only be done once per web page or server instance. All other actors are spawned through the application actor's *spawnActor* method which takes an extension of the *Actor* class as argument, spawns a new actor with an instance of this class and returns a far reference of the newly spawned actor.

Client-side actors are built atop web workers. We differentiate between the application actor which runs on the client's main thread (i.e. the user-interface 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.

For example, in CoCode the *GUI Actor* extends the *Application* class while the *Highlight Actor* extends the *Actor* class.

Distributed State Spiders.js provides two kinds of distributed state. The first kind **crosses actor boundaries by copy** and is implemented by extending the ***Isolate*** class. Isolate objects provide a synchronous API: using JavaScript's dot operator programmers synchronously access an isolate object's instance variables or invoke one of its methods. All primitive JavaScript data types (e.g. strings, booleans, numbers, etc.) are isolates. For example, in CoCode chat messages are implemented as isolates given that these are basically time stamped strings.

The second kind of distributed state **crosses actor boundaries by far reference** and is implemented by extending the ***Object*** class. The actor which instantiates an object from the *Object* class is said to *own* the object. The owning actor is able to synchronously access the object's fields and methods. All other actors are able to acquire a far reference to the object. These actors are only able to asynchronously access the object's fields or methods. In other words, accessing a field or invoking a method on a far reference returns a promise which might resolve with the field's value or method return value. If an exception occurs during the access or invocation the returned promise is rejected with said exception.

For example, client-side actors in CoCode distribute references to themselves using far references.

Asynchronous Computation JavaScript already natively provides a construct to represent asynchronous computations: *promises*. These objects represent the asynchronous or delayed execution of a piece of code. Promises offer a dual API. On the one side a promise's

Function	Signature
remote	string → number → Promise<FarRef>
buffRemote	string → number → FarRef
reflectOnActor	ActorMirror
reflectOnObject	Object → ObjectMirror
reflectOnObject	Isolate → IsolateMirror
serveApp	string → string → number → Object → Promise<void>

Table 6.2: Standard library available to all Spiders.js actors.

constructor accepts two callbacks which allow programmers to resolve or reject it. On the other side programmers are able to install callbacks, through a promise's *then* and *catch* methods, that are triggered upon resolution or rejection of a promise.

Spiders.js extends JavaScript promises' functionally (i.e. the promise API is the same in Spiders.js and JavaScript). In Spiders.js promises are first-class distributed objects: a promise can be sent around between actors. Moreover, one is able to resolve or reject and listen to the resolution or rejection of a remote promise.

6.3.1 Standard Library

Table 6.2 provides an overview of the functions provided by Spiders.js' standard library. This library is available to all actors as an instance variable called *libs*. Concretely, the library provides the following functionality:

remote connects to an actor running on the specified address and port and returns a promise which resolves with a far reference to said actor. This actor must be server-side, given that only server-side actors are able to specify the address and port on which they are reachable.

buffRemote connects to a server-side actor running on the specified address and port and immediately returns a far reference to said actor. This far reference buffers all invocations and accesses until the con-

nection with the server is established. In contrast, *remote*'s return promise only resolves once the connection is established.

For example, CoCode's client-side actors obtain an initial far reference to the server by using *buffRemote*.

reflectOnActor returns the actor mirror of the actor invoking the function. We provide more detail concerning Spiders.js' reflection capabilities in Section 6.5.

reflectOnObject returns the mirror on the provided object.

serveApp allows a server-side actor to function as an HTTP server for web applications. Concretely, *serveApp* takes three mandatory arguments: the path to the application's main HTML file, the path to the application's actor definitions and a port number on which the actor listens to HTTP requests. When a browser connects to the specified port it receives the HTML and actor definitions. Subsequently the HTML is rendered and the actors spawned. Optionally, *serveApp* accepts an options object which allows programmers to specify HTTP specific attributes (e.g. the path to publicly available folders on the server).

For example, the CoCode server uses the *serveApp* function upon booting after which clients can receive their actors' and user interface's source code.

This concludes our overview of Spiders.js' most important constructs. In the following section we use these constructs to implement CoCode.

6.4 Implementing CoCode

We start our implementation of CoCode by first focusing on the client-side functionality (i.e. syntax highlighting code). Subsequently we discuss the implementation of the distributed aspects of CoCode.

6.4.1 Client-side Implementation

Listing 6.1 implements the standalone functionality of each CoCode client (i.e. highlighting code in the web page as a user types). Each client uses

```
1  class CoCodeClient extends Application{
2
3   newCode(code : string){
4     highlighter.highlight(code).then((highlightedCode : string) =>
       {
5      //Update user interface
6     })
7   }
8  }
9
10 class HighlightActor extends Actor{
11
12  init(){
13   this.libs.importScripts('./highlightLib.js')
14  }
15
16  highlight(code : string) : string{
17    return highlightLib.highlight(code)
18  }
19 }
20
21 let CoCode  = new CoCodeClient()
22 let highlighter  = CoCode.spawnActor(HighlightActor)
```

Listing 6.1: Spawning client-side actors in Spiders.js.

two actors: one to update the user interface (i.e. the application actor) and one to highlight code (i.e. the *HighlightActor*).

The application actor's *newCode* method (line 3) is invoked for each of the user's key strokes (the code responsible for this is omitted for the sake of brevity). Subsequently, the application actor invokes the *highlight* method on the far reference to the highlighting actor with the new code as argument. Given that the code is represented by a string, which is a primitive data type, it is sent to the highlighting actor by copy. Method invocations on far references are asynchronous: they return promises and are translated to asynchronous messages sent to the owner of the referenced object. Such a promise either resolves with the method's return value or is rejected with an error. In our case the application actor registers a callback (line 4) on a promise that resolves with the return value of the highlighting actor's *highlight* method. Once the promise resolves the user interface is updated with the highlighted code.

6.4.2 Distributed Implementation

Until now the CoCode application is standalone: it only responds to new code being produced by the local user. Turning CoCode into a true web application requires a server and communication between client-side instances.

6.4.2.1 Implementing the Server

The CoCode server's tasks are twofold. First, it provides an initial entry point to the application. A client connects to the server and receives the definition of the actors discussed in Section 6.4.1 and all of CoCode's HTML definitions. Second, it maintains the state of the application (i.e. the shared code base) and notifies clients of updates to this state.

```
1  class CoCodeServer extends Application{
2      clients          : Map<string,FarRef<ClientInterface>>
3      currentCode      : String
4
5      constructor(){
6          this.clients = new Map()
7          this.libs.serveApp("./index.html","./client.js",8888)
8      }
9
10     register(newClient: FarRef<ClientInterface>,newName: string){
11         this.clients.forEach((client,name)=>{
12             client.newCoder(newClient,newName)
13             newClient.newCoder(client,name)
14         })
15         this.clients.set(newName,newClient)
16         newClient.globalCodeUpdate(this.currentCode)
17     }
18
19     updateCode(rawCode : String){
20         this.currentCode = rawCode
21         this.clients.forEach((client : FarRef<CoCodeClient>)=>{
22             client.globalCodeUpdate(rawCode)
23         })
24     }
25 }
26 new CoCodeServer()
```

Listing 6.2: Implementing the CoCode server in Spiders.js.

Listing 6.2 implements the CoCode server. The server actor starts by setting up an HTTP server (line 7) on port 8888 (i.e. by invoking the *serveApp* method). Browsers connecting to this server receive the speci-

fied *HTML* files (i.e. *index.html* in our example) and the specified actor definitions (i.e. *client.js* in our example).

The server acts as a discovery service, allowing clients to obtain references to each other. This functionality can safely run on a single thread, which is why the server is implemented solely by an application actor. Concretely, the *register* method is invoked by a new client upon starting the application. The arguments to this method are a self-reference to the client and the client's name (line 10). The method then forwards this name and reference to all connected clients by invoking their *newCoder* methods.

The server also maintains the application's global state (i.e. the shared code base). A clients invokes the server's *updateCode* method whenever its user presses the *commit* button (line 19). In turn, the server notifies all other clients that the code base has changed.

6.4.2.2 Adapting the Clients for Distribution

The addition of the server requires us to update the implementation of our client-side CoCode actors. Listing 6.3 shows the additions to the original client-side code of Listing 6.1 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.

```
1  class CoCodeClient extends Application {
2   server : FarRef<CoCodeServer>
3
4   constructor(clientName : string){
5    super()
6    this.server = this.libs.buffRemote(localhost,8888)
7    this.server.register(new ClientInterface(clientName),clientName)
8   }
9
10  newCode(code : string){
11    this.server.updateCode(code)
12  }
13 }
```

Listing 6.3: Making CoCode clients distributed.

There are two additions to the original code. First, the *newCode* method (line10) is called by the UI whenever a user types in code and invokes the *updateCode* method on the server. Second, each client first acquires a reference to the server actor through the *buffRemote* method

comprised in the standard actor library (line 6). This function takes the IP address and port number of a given server actor and returns a far reference to said actor. In our case this reference is used by each client to register themselves as a coder. This is done by invoking the *register* method on the server reference. The arguments to this invocation are a reference to a *ClientInterface* object and the client's name. The former, which we discuss below, is an object and is therefore passed by far reference while the latter is a string and is passed to the server by copy.

```
1  class ClientInterface{
2   clientName : string
3   coCoders : Map<string,FarRef<ClientInterface>>
4
5   constructor(clientName : string){
6    super()
7    this.clientName = clientName
8   }
9
10  newCoder(coder : FarRef<ClientInterface>,coderName : string){
11   this.coCoders.set(coderName,coder)
12  }
13
14  globalCodeUpdate(newCode){
15   //Highlight code and update user interface
16  }
17
18  newMessage(msg : Message){
19   //Update user interface
20  }
21
22  sendPublicMessage(text : string){
23   this.coCoders.forEach((coder : FarRef<ClientInterface>)=>{
24    coder.newMessage(new Message(this.clientName,text))
25   })
26  }
27
28  sendPrivateMessage(to : string,text : string){
29   this.coCoders.get(to).newMessage(new Message(this.clientName,
       text))
30  }
31 }
```

Listing 6.4: The distributed interface of CoCode clients.

Listing 6.4 provides the definition of the interface used by clients to communicate with each other. *ClientInterface* implements two pieces of functionality: reacting to code updates and messaging across clients. Code updates are broadcasted by the server to all clients by invoking the *globalCodeUpdate* method, which highlights the code and updates the user

interface. The server broadcasts an update as soon as a client invokes the *updateCode* method (line 10 in Listing 6.3).

Users 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).

```
1  class Message extends SpiderIsolate{
2   from : string
3   text  : string
4   date : Date
5
6   constructor(sender : string,text : string){
7    super()
8    this.from  = sender
9    this.text   = text
10   this.date  = new Date()
11  }
12 }
```

Listing 6.5: Making CoCode clients distributed.

In both cases an *isolated* object is created (see Listing 6.5 for the object's class definition) which contains the name of the sender, the date and the actual text of the message. Remember from Section 6.3 that isolated objects are sent by copy rather than by far reference. As such, when a client receives a message through the *newMessage* method, the message's data can directly be read from the copied object. If messages were implemented as regular objects they would be sent by far reference. Each access to a message (e.g. getting the message's date) would then return a promise, which would be impractical for this use case.

CoCode showcases how Spiders.js tackles the problems of JavaScript actors. More precisely, Spiders.js provides the following features:

Location Transparency Programmers extend the same *Actor* classes regardless of whether the actor is going to be instantiated server or client-side. Moreover, actor semantics do not depend on their location. For example, both client and server-side actors communicate by asynchronously invoking methods on each other's objects.

Fine-grained parameter passing semantics Spiders.js provides two built-in parameter passing semantics. Isolate objects and primitive data types (i.e. numbers, booleans, strings) cross actor boundaries using pass-by-copy semantics. All other objects cross actor boundaries using pass-by-far-reference semantics.

First-class actor references Spiders.js allows arbitrary communication patterns between actors. It achieves this by allowing actor references to be parameters to asynchronous method invocations.

6.5 Metaprogramming in Spiders.js

Spiders.js fulfils requirement R_1 for DRIA-oriented programming models: Its actors serve as a unit of distribution for both horizontal as well as vertical distribution. Through a meta-layer, Spiders.js also fulfils requirement R_4 for DRIA-oriented programming models: it allows programmers to extend built-in semantics. More precisely, Spiders.js offers a *mirror-based* meta-level architecture [BU04]. In a nutshell, **mirror-based** reflection mechanisms are based on three core principles [BU04]:

Encapsulation Meta-level entities encapsulate their implementation details. This principle promotes reuse of meta-level code by letting programmers develop their meta programs against an interface rather than an implementation. In other words, different implementations of the same meta-level entity can be used interchangeably as long as they respect the provided interface.

Stratification Meta-level entities are completely separated from base-level entities. Simply put, base-level entities should only provide base-level functionality while meta-level entities should only offer meta-level functionality. As a result, base-level entities do not have direct references to their meta-level counterparts. Instead, accessing meta-level entities is achieved through dedicated linguistic constructs.

Ontological Correspondence Meta-level entities are governed according to the same principles and rules that apply to base-level entities. There are two forms of ontological correspondence: *structural* and *temporal* ontological correspondence. Structural correspondence means that each base-level entity has a corresponding meta-level entity. Temporal correspondence means that the meta-level architecture distinguishes between code (i.e. the textual representation of a process) and computation (the actual running process).

The following sections describe how we apply mirrors and their core principles in Spiders.js.

6.5.1 Mirrors in Practice

Spiders.js' mirror architecture is heavily inspired by AmbientTalk's mirror architecture [MVCT+09]. As such, it provides two levels of reflection. First, reflection on objects through object mirrors. Second, reflection on actors through actor mirrors. We describe each of these levels in details.

6.5.1.1 Mirrors on Objects

Reconsider our CoCode application. Assume that CoCode requires to keep track of the exact moment a chat message is sent or received by a client. Implementing this bookkeeping functionality at the base level (i.e. within the *sendPublicMessage*, *sendPrivateMessage* and *newMessage* methods) poses two problems. First, this pollutes the base-level logic responsible for sending and receiving messages with message bookkeeping logic. Second, a base-level programmer is unable to accurately determine when a chat message is actually sent or received. The programmer is only able to determine when the *sendPublicMessage*, *sendPrivateMessage* or *newMessage* methods are executed.

To implement the message bookkeeping functionality we make use of Spiders.js' mirrors on objects. Amongst others, mirrors on objects allow programmers to perform *intercession* [KDRB91]. In other words, programmers are able to redefine parts of Spiders.js' standard semantics. Programmers do this by extending one of the two default mirrors provided by Spiders.js: *SpiderIsolateMirror* or *SpiderObjectMirror*. The former is used to redefine the behaviour of isolates while the later is used to redefine the behaviour of all non-isolate objects in Spiders.js.

Listing 6.6 implements CoCode's message bookkeeping functionality. We start by extending the *SpiderIsolateMirror*, given that chat messages are isolates. This default isolate mirror provides two methods which we override: *pass* and *resolve*. The former is called right before an isolate is sent from one actor to another. The latter is called right after an isolate is received by an actor. In both cases the mirror records the timestamp of the interceded event.

Our chat messages are yet to be connected to instances of *ChatMessageMirror*. In order to connect an isolate with a specific mirror it suffices to pass an instance of the mirror class to the *Isolate* constructor (i.e. to the *super()* call on line 7 of Listing 6.5). Actors are able to access an isolate's

```
1  class ChatMessageMirror extends SpiderIsolateMirror{
2      sentTimeStamp
3      receiveTimeStamp
4
5      pass(hostActorMirror: SpiderActorMirror) {
6        this.sentTimeStamp = Date.now()
7        return super.pass(hostActorMirror)
8      }
9
10     resolve(hostActorMirror: SpiderActorMirror) {
11       this.receiveTimeStamp = Date.now()
12       return super.resolve(hostActorMirror)
13     }
14 }
```

Listing 6.6: Implementing message bookkeeping through mirrors on objects.

mirror through the *reflectOnObject* method available in their standard library.

6.5.1.2 Mirrors on Actors

Assume we want to extend CoCode's messaging functionality even further. More specifically, we want to implement a delivery notification system. In other words, clients receive delivery notifications for the private messages they send. We implement this functionality by extending the default mirror for actors, as shown in Listing 6.7.

We extend the default actor mirror to override its *receiveInvocation* method. This method is invoked each time an actor receives an asynchronous method invocation on one of the object it owns. The method's arguments are a far reference to the actor sending the message, the object on which the method is invoked, the method's name and arguments (we leave out the remaining optional arguments for the sake of brevity). The *ReceptionMirror* extends the default actor semantics for two specific messages. First, if the invoked method is *newPrivateMessage* the receiving actor notifies the sending actor that the chat message is delivered. To do so the receiving actor invokes the *privateMessageReceived* method on the sending actor (line 5). Second, if the invoked method is *privateMessageReceived* the receiving actor notifies the user that its chat message

```
1  class ReceptionMirror extends SpiderActorMirror{
2
3      receiveInvocation(sender: FarRef<any>, targetObject: Object,
       methodName: string, args: Array<any>) {
4          if(methodName == "newPrivateMessage"){
5              sender.privateMessageReceived()
6              return super.receiveInvocation(...arguments)
7          }
8          else if(methodName == "privateMessageReceived"){
9              //Show notification in the user interface
10         }
11         else{
12             return super.receiveInvocation(...arguments)
13         }
14     }
15 }
```

Listing 6.7: Implementing notification through mirrors on actors.

is delivered. All other method invocations are delegated to the default mirror (line 12).

As is the case for isolates, actors are connected with their mirrors by passing an instance of an actor mirror to the actor's constructor (i.e. to the *super()* call on line 5 of Listing 6.3. Actors are able to access their mirror by invoking the *reflectOnActor* method which is part of the actor standard library. This nullary method returns the mirror of the actor in which it is invoked.

6.5.2 Spiders.js' Meta-object and Meta-actor Protocol

A metaobject protocol [KDRB91] governs the interactions between all objects and actors. For each Spiders.js feature this protocol dictates the sequence of meta messages sent between objects, mirrors and actors. To explain the protocol we use one of the most important Spiders.js features: method invocations on far references.

We reuse CoCode's private chat message functionality detailed in Section 6.4.2.2. More precisely, assume that a coder *c1* sends a private message *m* to another client *c2*. Figure 6.4 provides an overview of *c1*'s application actor. This actor contains two base-level objects: a far reference to *c2*'s client interface and a message *m*. Both these objects are causally connected to two meta-level mirrors. Moreover, the meta-level

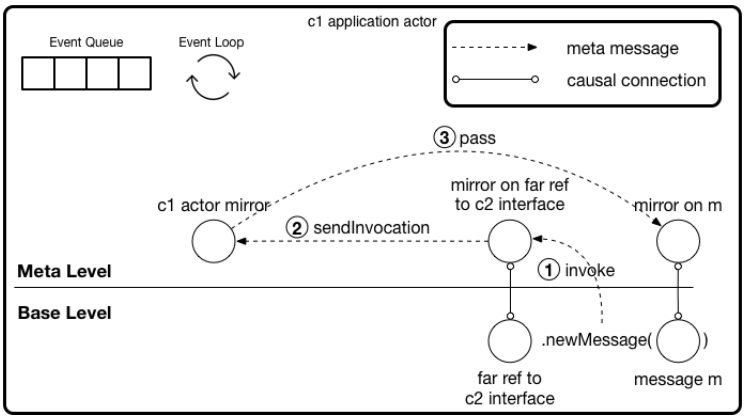


Figure 6.4: Method invocation protocol at sender side.

of *c1*'s application actor also contains the actor mirror, event queue and event loop.

The labelled arrows in Figure 6.4 showcase the first half of the protocol responsible for method invocations on far references. This half of the protocol is responsible for the sequence of operations at the sender side. The protocol starts when, on the base level, a method is invoked on a far reference. In our example the *newMessage* method is invoked on a far reference to *c2*'s client interface with the message *m* as argument. The following steps in the protocol succeed this method invocation:

Step 1. The *invoke* method is called on the mirror attached to the far reference to *c2*'s client interface. Arguments to this meta-method are the name of the invoked base method (i.e. *newMessage*) and the provided arguments. The mirror on the far reference to *c2*'s client interface implements the semantics of this method invocation.

Step 2. Translating method invocations on far references to asynchronous messages is the responsibility of actor mirrors. In our example the mirror on *c1*'s application actor sends an asynchronous message to *c2*'s application actor. These semantics are implemented in an actor mirror's *sendInvocation* method which takes as arguments a far reference to the receiver, the method name and the provided arguments. In our example these arguments are a reference to *c2*'s application actor, *newMessage* and the message *m*.

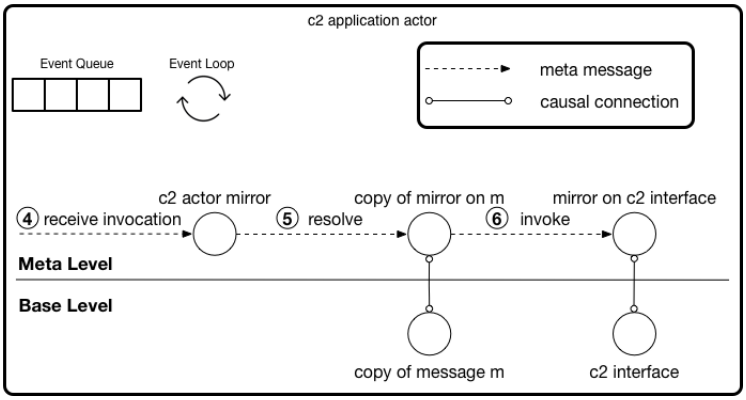


Figure 6.5: Method invocation protocol at receiver side.

Step 3. Each argument from the previous step must first be serialised before sending the message. To this end the *pass* method is invoked on each argument's mirror. This method allows objects to determine what information should be serialised and sent between actors. For example, the default semantics for isolates is to return a copy of themselves. This is the case for our example when *pass* is invoked on *m*'s mirror.

Figure 6.5 shows the part of the method invocation protocol responsible for operations at the receiver side. This part of the protocol is subdivided into three steps:

Step 4. *c2*'s application actor receives the message containing the invocation sent by *c1*'s application actor. This message is handled by invoking the *receiveInvocation* method on *c2*'s application actor. This method takes as arguments a far reference to the sender, a reference to the target object, the invoked method's name and the arguments used for the invocation.

Step 5. The *resolve* method is called on the mirror of each argument used in the original invocation. This method is the dual of *pass*: it allows an object to determine how it should be deserialised.

Step 6. The method invocation which initiated the protocol (i.e. *newMessage* in our example) is applied to the target object. This is done

	Method	**Signature**
Introspection	getId	number
	getFields	Array<Field>
	getMethods	Array<Method>
	getField	string → Field
	getMethod	string → Method
Self-modification	addField	string → any → void
	addMethod	string → Function → void
Intercession	invoke	string → Array<any> → void
	access	string → any
	write	string → any → void
	pass	Object
	resolve	Object

Table 6.3: Mirror on objects API.

by invoking the *invoke* method on the target object's mirror. We discuss this meta-method in step 1.

6.5.3 Meta-level Constructs

This section provides an exhaustive overview of the meta-constructs available in Spiders.js. For each construct we briefly discuss its API and functionality. We divide the interface offered by both mirrors on objects and mirrors on actors according to the *kind* of metaprogramming they provide. According to [KDRB91], metaprogramming can be divided into three kinds. ***Introspection*** allows a program to access its own structure. ***Self-modification*** allows a program to modify its own structure. ***Intercession*** allows a program to redefine parts of the language-level operations. We classify the methods offered by Spiders.js' object and actor mirrors based on these three kinds of metaprogramming.

6.5.3.1 Object Mirror

Table 6.3 shows an overview of the methods provided by object mirrors. Concretely, object mirrors in Spiders.js support the aforementioned three kinds of metaprogramming as follows.

Kind	Method	Signature
Introspection	getObjects	Array<Object>
	getObject	number → Object
Self-modification	addObject	Object → number
Intercession	initialise	void
	receiveInvocation	FarRef → Object → string → Array → void
	receiveAccess	FarRef → Object → string → void
	sendInvocation	FarRef → string → void
	sendAccess	FarRef → string → void

Table 6.4: Mirrors on actors API.

Introspection Through the mirror on objects one is able to inspect an object's fields and methods. A *Field* is a tuple containing the field's name and its value. Similarly, a *Method* is a tuple containing the method's name and the actual closure. *getFields* and *getMethods* return all of the objects fields and methods respectively. Moreover, *getField* and *getMethod* return a specific field or method.

Self-modification *addField* and *addMethod* allow one to modify an object through its mirror. The methods respectively add a new field or method to the mirrored base object.

Intercession One can override the default semantics for Spiders.js objects in two ways. First, by providing a custom implementation of *invoke*, *access* or *write*. These methods implement the semantics of method invocation, reading fields and writing fields respectively. Second, by providing a custom implementation of *pass* or *resolve*. These methods are invoked to serialise and deserialise an object when it is sent between actors.

6.5.3.2 Actor Mirror

Table 6.4 provides an overview of the methods offered by actor mirrors. Concretely, actor mirrors in Spiders.js support the aforementioned three kinds of metaprogramming as follows:

Introspection Actor mirrors provide an interface to inspect an actor's object heap. One can either request all objects in the heap through *getObjects* or request a specific object using its identifier through *getObject*.

Self-modification The *addObject* method modifies the actor's heap by adding the provided object. The method returns the added object's identity in the heap.

Intercession Actor mirrors provide three entry points to override an actor's default behaviour. First, the *initialise* method that is invoked right after an actor is spawned and before its *init* method is invoked. Second, the *receiveInvocation* and *receiveAccess* methods that are invoked upon receiving an asynchronous message from another actor. Depending on whether this message contains a method invocation on one of the receivers objects or a field access the *receiveInvocation* or the *receiveAccess* method is called. Third, the *sendInvocation* and *sendAccess* methods are invoked whenever a method is invoked on a far reference or a far reference's field is accessed. By default these methods send an asynchronous message to the actor owning the far referenced object.

6.6 Evaluation

We evaluate Spiders.js along two dimensions (i.e. qualitatively and quantitatively): performance and ease of use for programmers. To evaluate the former we compare the runtime results for three implementations of the Savina benchmark suite [IS14]: one using web workers, one using child processes and one using Spiders.js. Moreover, we measure the speedup obtained by parallelising an example application and compare it to the speedup obtained with built-in JavaScript actors. To evaluate the latter we compare the implementation of a multiplayer version of the arcade game *Pong*. Moreover, we use Spiders.js' meta facilities to implement two novel classes of replicable state.

Our evaluation shows that Spiders.js exhibits a certain performance overhead when compared to built-in JavaScript actors, especially when it comes to actor creation and message passing. However, as our *Pong* case study shows, Spiders.js allows one to write complete web applications more effectively than is the case for plain JavaScript. Moreover, our implementation of two novel classes of replicable state informally shows how Spiders.js' meta facilities easily allow one to extends its built-in semantics.

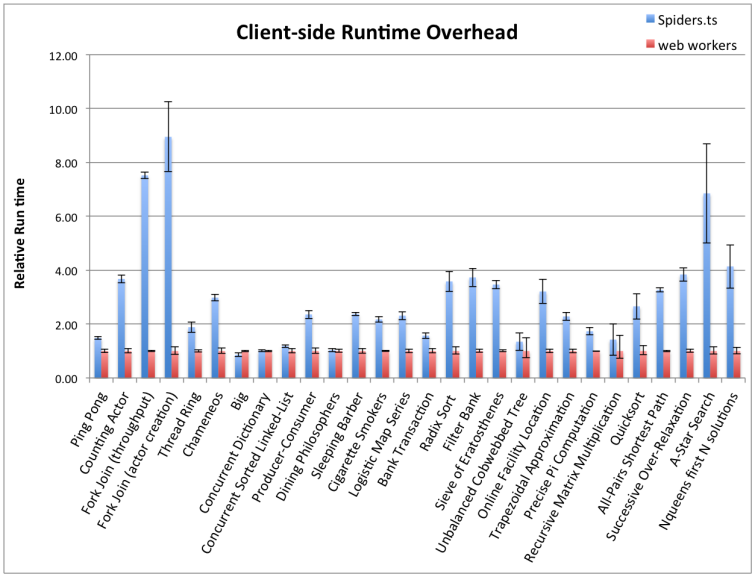


Figure 6.6: Comparing Spiders.js and web workers in the Savina Benchmark Suite. Run times are normalised to the results of web workers. Error bars indicate the 95% confidence interval.

6.6.1 Performance

All benchmarks discussed in this section are performed both on the server and client-side. Client-side benchmarks are performed in Google Chrome (version 56.0.2924.87) on a Macbook Pro with a 2,8 GHz Intel core i7 processor, 16GB of RAM memory running Mac OSX Sierra(version 10.12.3). Server-side benchmarks are performed on an Ubuntu 14.04 server with two dual core Intel Xeon 2637 processors at 3.5 GHz with 265 GB of RAM memory.

6.6.1.1 Runtime Overhead

To measure Spiders.js' overall overhead we compare the runtime performance of a Spiders.js implementation and a web workers implementation of the **Savina benchmark suite** [IS14]. Figure 6.6 shows the mean run time to completion for each implementation of each application in the

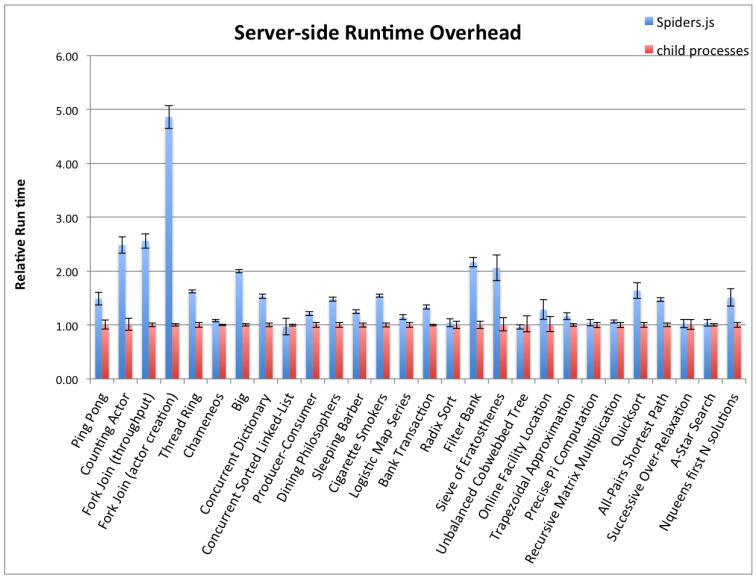


Figure 6.7: Comparing Spiders.js and child processes in the Savina Benchmark Suite. Run times are normalised to the results of child processes. Error bars indicate the 95% confidence interval.

suite [2]. The run times are normalised to the results obtained by web workers to better highlight the overhead introduced by Spiders.js. The results immediately show Spiders.js' biggest overhead: actor creation (as shown by the *Fork Join(actor creation)* application). Web workers are simply spawned by providing the path to a JavaScript source file containing their behaviour. In Spiders.js the object representing the actor must first be serialised and sent to a newly spawned web worker which deserialises the object before accepting any messages (see Section 6.4.1). Spiders.js also underperforms with regards to message passing overhead and throughput (respectively shown by the *counting actor* and *Fork Join(throughput)* applications). This is due to Spiders.js' underlying runtime, which requires a considerable amount of meta-data to be attached to every message sent between actors (e.g. to handle the return values of asynchronous method invocations). These weaknesses of Spiders.js are further showcased by our

[2]The results for the *big* benchmark contain an unknown anomaly (i.e. Spiders.js slightly outperforms web workers for some measurements). We suspect an optimisation of the JavaScript engine is responsible for the anomaly. Further experiments are needed to conclusively determine the cause of the anomaly.

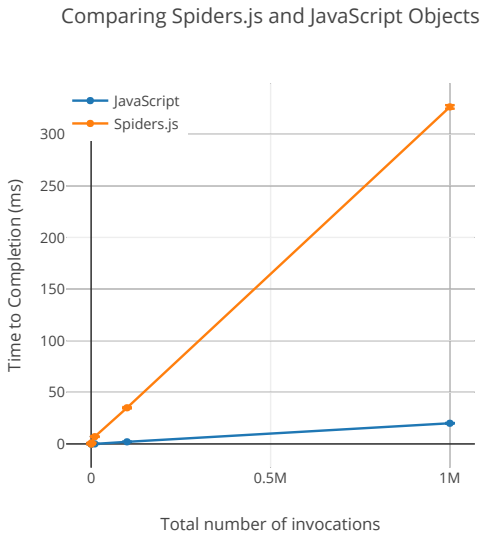


Figure 6.8: Comparing method invocation performance for Spiders.js and JavaScript objects. Error bars indicate the 95% confidence interval.

server-side comparison using a child process implementation of the Savina suite. As Figure 6.7 shows, the overhead is less pronounced but still substantial when comparing Spiders.js and child processes.

A source of overhead not directly showcased by the Savina benchmark suite is Spiders.js' meta layer. Consider Figure 6.8, the figure shows results of a micro benchmark involving a Spiders.js isolate and a JavaScript object[3]. We increasingly perform a number of method invocations on both objects and measure the time it takes for the objects to return from all invocations. As the figure showcases, method invocations on Spiders.js objects are orders of magnitude slower than is the case for plain JavaScript objects. This is primarily due to the fact that all interactions with objects in Spiders.js are first captured by its meta layer.

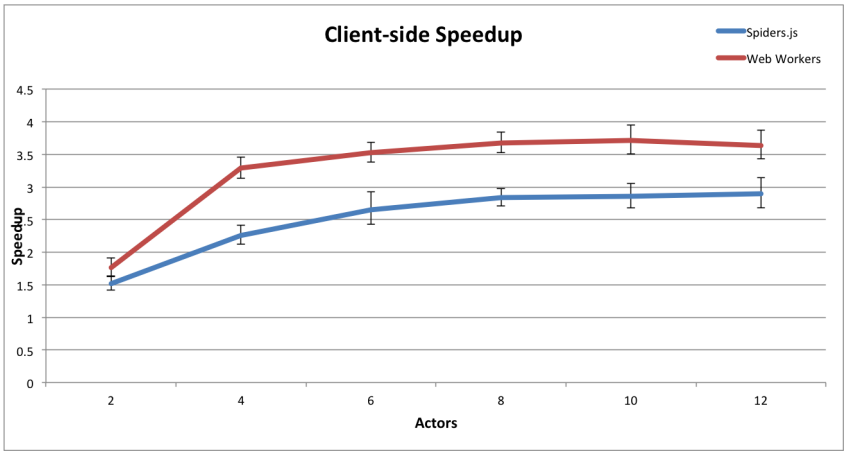


Figure 6.9: Speedup obtained by comparing the sequential Monte Carlo application with parallel versions using Spiders.js and web workers. Error bars indicate the 95% confidence interval.

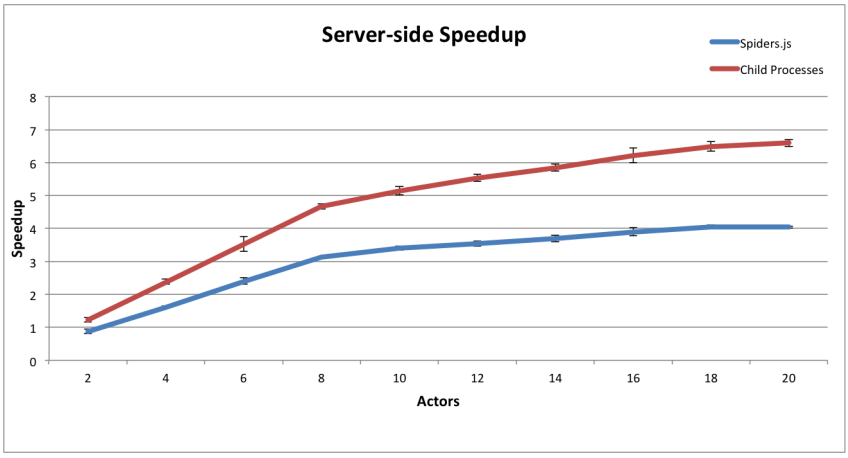


Figure 6.10: Speedup obtained by comparing the sequential Monte Carlo application with parallel versions using Spiders.js and child processes. Error bars indicate the 95% confidence interval.

6.6.1.2 Parallelism

As the Savina benchmarks show, Spiders.js introduces an overhead compared to JavaScript. However, programmers are still able to speed up their applications using the parallelism provided by Spiders.js actors. To show this we measure the speedup obtained by approximating π using the *Monte Carlo* method in Spiders.js. We compare the time needed by a sequential application to approximate π a given number of times with a parallel version which spreads the same workload across a variable amount of actors. We specifically choose this application as it does not require a large number of actors nor does it require extensive communication between these actors. As such, this benchmark alleviates Spiders.js' weaknesses (i.e. messaging overhead and actor creation overhead) and purely shows Spiders.js' parallel abilities.

Figure 6.9 shows the measured results while running this comparison in-browser. Comparing the speedup obtained by the Spiders.js and web workers implementation shows that Spiders.js suffers from an overhead. This overhead is also present with regards to child processes, as shown by Figure 6.10. However, the overhead remains within acceptable bounds: Spiders.js is still able to significantly parallelise client and server-side web applications.

6.6.2 Code Complexity

To measure the expressive power of Spiders.js over native JavaScript we implement and compare a multiplayer version of the arcade game *Pong*[4]. We classify the code of each implementation into three categories:

Message Handling Code that implements how a part of the application is to respond to a given message. This includes implementing and registering callbacks, dispatching on message types and implementing actor methods.

Message Sending Code which implements the communication between clients and between the clients and the server. This includes creating

[3] `https://gitlab.soft.vub.ac.be/fmyter/triumvirate/tree/master/MicroBenchmark`

[4] `https://github.com/myter/Spiders.js/tree/master/SpiderPong`

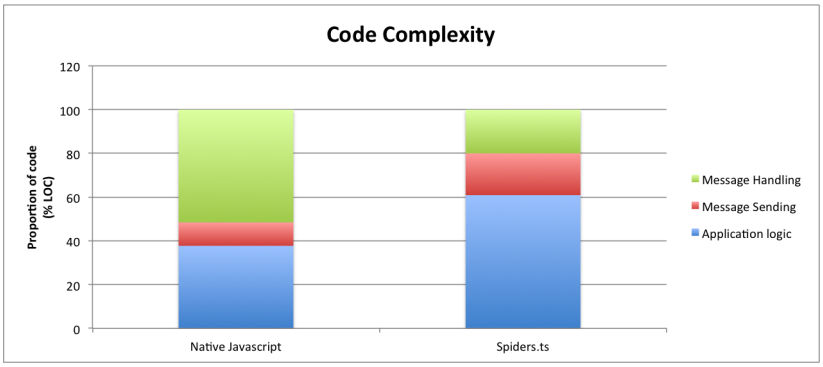


Figure 6.11: Proportion, in percentage of the total lines of code, dedicated to each category of code for a native JavaScript and Spider.js implementation of multiplayer *Pong*.

and sending messages, opening sockets and invoking methods on far references or defining *isolate* objects.

Application Logic Code which implements the game's core functionality (e.g. updating the user interface when a player's score has changed).

To compare both implementations we measure the proportion of each category of code to the application's total lines of code. Ideally an application should mostly be comprised of *application logic* code. Appendix A.1 and A.2 contain both implementations highlighted according to the three categories.

Figure 6.11 shows how each implementation is divided into the three categories. For each category of code, the figure shows the percentage it represents with regards to the application's total number of lines of code. The biggest difference between both version is the proportion of the application dedicated to message handling. In contrast to regular JavaScript, message handling in Spiders.js is done implicitly (i.e. an actor's methods act as message handlers). Moreover, in Spiders.js two clients are able to send messages to each other simply by using far references. For our pong example this entails that a client can directly send updates to his opponent, Spiders.js will ensure that the server routes messages correctly. In the version implemented using plain JavaScript this routing must be implemented manually, adding additional code for message handling on

both server and client. In conclusion, Spiders.js allows programmers to focus on the essential complexity and logic of their application while relegating accidental complexity such as message handling and routing to the Spiders.js runtime.

6.6.3 Mirrors in Action

Section 2.2 of Chapter 2 discusses four requirements for programming models that target DRIAs. The fourth requirement stipulates that a programming model for DRIAs should allow programmers to define custom parameter passing and state update semantics. To this end Spiders.js, and by extension Triumvirate, provides a mirror-based meta layer.

This section showcases how Spiders.js' meta layer meets the fourth requirement for DRIA-oriented programming models. To do so we suplement the two classes of replicable state from Chapter 4 (i.e. eventual replicas based on GSP [BLPF15] and strong replicas based on far references [CGS+14]) with two new classes of replicable state. These novel classes are implemented solely using Spiders.js' mirrors.

- The first new class of replicable state is based on CRDTs [SPBZ11] and is available and partition-tolerant with regards to the CAP theorem [Bre00].
- The second new class of replicable state is based on the two-phase commit protocol [LS76] and is consistent and partition-tolerant with regards to the CAP theorem.

6.6.3.1 Strong Eventual Replicas

Eventual replicas, as discussed in Section 4.3.2 of Chapter 4, guarantee eventual consistency (see Definition 4). In a nutshell, this guarantees that all replicas will eventually be in the same state provided that updates to all replicas stop.

Conflict-free replicated data types [SPBZ11] guarantee *strong eventual consistency*. Simply put, CRDTs guarantee that all replicas will eventually be in the same state and that replicas never roll back operations, which is not guaranteed by "regular" eventual consistency. This latter type of consistency is already provided by eventual replicas and their GSP-based implementation. CRDTs are able to provide this stronger guarantee by

requiring the operations that apply over them to be associative, commutative and idempotent.

```
1 class CRDT extends Isolate{
2  constructor(){
3   super(new CRDTMirror())
4  }
5 }
```

Listing 6.8: CRDTs in Triumvirate.

Listing 6.8 shows the definition of CRDTs in Triumvirate, which are wrappers for Spiders.js isolates. It is important to note that there is no predefined API for CRDTs. Rather, the API depends on the specific data type (e.g. sets, counters, etc.). The bulk of CRDT logic is therefore implemented by the CRDT mirror given by Listing 6.9

```
1  class CRDTMirror extends IsolateMirror{
2   broker : CRDTBroker
3
4   constructor(){
5    super()
6    this.broker = new CRDTBroker(this.base)
7   }
8
9   invoke(methodName,args) {
10   this.broker.newOperation(methodName,args)
11  }
12
13  resolve() {
14   let newBroker = new CRDTBroker(this.base)
15   this.broker.newInstance(newBroker)
16   this.broker = newBroker
17   return super.resolve()
18  }
19 }
```

Listing 6.9: CRDTMirror implementation.

The mirror's task is to ensure that a CRDT replica maintains references to all other replicas. Moreover, the mirror must capture method invocations on its replica and broadcast the invocations to all other replicas. We implement this functionality by overwriting two methods of Spiders.js' default mirror: *invoke* and *resolve*. Invoke (line 9) delegates all invocations to a *broker* object that broadcasts the invocations instead of only invoking the method on the replica it mirrors. We discuss this *broker* object further in this section.

```
1  class CRDTBroker{
2   instances : Array<FarRef<CRDTBroker>>
3   crdt      : CRDT
4   constructor(crdt){
5    super()
6    this.instances  = []
7    this.crdt       = crdt
8   }
9
10  newInstance(instanceRef : FarRef<CRDTBroker>){
11   this.instances.push(instanceRef)
12   this.instances.forEach(instance=>{instanceRef.newInstance(
       instance)})
13  }
14  applyOperation(methodName : PropertyKey,args : Array<any>){
15   this.crdt[methodName](...args)
16  }
17
18  newOperation(methodName,args){
19   this.instances.forEach(instance =>{ instance.applyOperation(
       methodName,args)})
20   this.applyOperation(methodName,args)
21  }
22 }
```

Listing 6.10: Maintaining CRDT bookkeeping information.

As we discuss in Section 6.5.2, resolve is invoked when an actor receives and deserialises an isolate. It is important to note that although an isolate and its mirror are sent across actors by copy, the same does not necessarily hold for their fields. For example, the *broker* field of a CRDT mirror is a regular object and therefore adheres to pass-by-far-reference semantics. In other words, the *broker* field in *resolve* contains a far reference to the *broker* object residing in the sending actor's heap. Our implementation of resolve invokes the *newInstance* method on this far reference before replacing the field with a freshly created (local) *broker* object.

Listing 6.10 gives the definition of the *broker* used by CRDT mirrors. Each *broker* is associated to a single CRDT replica, and therefore also a single CRDT mirror. It implements two pieces of functionality. First, it maintains an array of far references to the *brokers* of all other replicas. Second, it uses this array to broadcast method invocations on its associated CRDT replica.

The first task is implemented by the *newInstance* method (line 10). This method is invoked whenever a new replica is created and ensures

that *brokers* form a full-mesh network (i.e. all *brokers* have references to all other *brokers*). *newOperation* (line 18) implements the second task. The method instructs all *brokers* part of the full-mesh network to apply the provided method using the given arguments. Subsequently, the *broker* locally applies the method to its associated CRDT (line 20).

Strong Eventual Replicas in Practice

```
1  class GSet<T> extends CRDT{
2   values : Array<T>
3
4   constructor(){
5    super()
6    this.values = []
7   }
8
9   addValue(val : T) {
10   if(!this.values.has(val)){
11    this.values.push(val)
12   }
13  }
14 }
```

Listing 6.11: CRDTs in Practice.

As an example of how CRDTs are defined in Triumvirate consider Listing 6.11. More concretely the listing implements a grow-only set (or *GSet*) [Baq15], which is a specific type of CRDT. As the name implies, a grow-only set only allows elements to be added and does not support the removal of elements. The set supports a single method (i.e. *addValue*) which adds a value to the set provided that the value is not present in the set yet.

Triumvirate programmers use *GSets* as first class values in their distributed applications. Our meta-implementation of CRDTs frees the programmer from manually doing the bookkeeping of CRDT replicas across actors and guarantees strong eventual consistency.

6.6.3.2 Multi-replication Strong Replicas

Strong replicas guarantee sequential consistency (see Definition 1). This ensures programmers that two strong replicas will never be in an inconsistent state (at the cost of not always being available). However, the implementation of strong replicas using far references (see Section 4.3.1

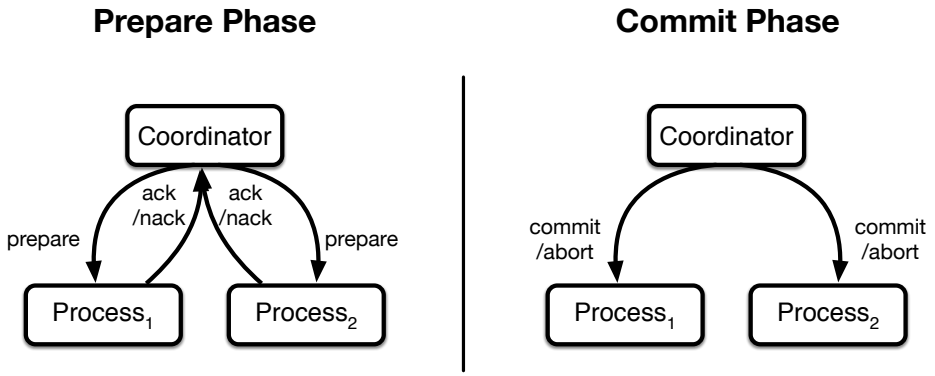


Figure 6.12: Two-phase commit.

of Chapter 4) only offers a replication factor of one. Concretely, only one actor maintains an actual replica while all other actors only have proxies to said replica. A failure for this single actor therefore entails the unavailability of all proxies.

However, increasing the replication factor of strong replicas engenders new challenges. Multiple actors now have actual replicas containing methods and state instead of proxies to a single replica. Triumvirate must coordinate the access to these actual replicas in order to maintain their strong consistency. To this end we extend Triumvirate's strong replicas with multi-replicated strong (MRS) replicas using Spiders.js' meta facilities. We start by briefly introducing the mechanism used to coordinate MRS replicas, two-phase commit, before detailing their implementation.

Two-phase Commit

The mechanism used to coordinate access to MRS replicas is the two-phase commit [LS76] protocol. This protocol allows a number of processes to either all commit a certain transaction, given by a central coordinator, or all abort the transaction. As the name suggests the protocol operates in two phases (see Figure 6.12):

Prepare Phase In the prepare phase the central coordinator tells all processes to prepare to commit a given transaction. Processes either return an acknowledgement if they were able to execute the transaction or they return a failure otherwise.

Commit Phase The coordinator instructs all processes to commit the transaction to persistent storage if they all returned an acknowledgement in the prepare phase. Otherwise the coordinator instructs all processes to abort the transaction and rollback.

The protocol guarantees that when it finishes all processes have either committed or aborted the transaction.

Applying Two-phase Commit to Replicas

Before detailing our exact implementation in Spiders.js we first provide a high-level overview of how we apply the two-phase commit to coordinate accesses to replicas. We differentiate between two kinds of replicas. First, a single *master* replica assumes the role of *coordinator* in the two-phase commit protocol. In other words, this replica ensures that transactions either commit or abort. Such a replica is created by instantiating an object from a MRS replicable class (i.e. using the *new* operator). Second, *worker* replicas assume the role of processes in the two-phase commit protocol. In other words, their role is to apply transactions and return acknowledgements to the *master* replica. *Worker* replicas are created when a coordinating or *worker* replica crosses actor boundaries (e.g. as part of an asynchronous message send).

Writing to a MRS replica's field starts the two-phase commit protocol. The *master* replica first inquires whether all *worker* replicas are ready to perform the write operation. If all replicas respond positively in a timely fashion the *master* replica orders all *worker* replicas to perform (i.e. commit) the write. Otherwise the *master* replica aborts and *worker* replicas return to an idle state. Reading from a MRS replica only returns a value if there is currently no ongoing transaction. This ensures that reading at a given point in time always returns the same value across replicas.

Coordinating Replicas using Two-phase Commit

```
class MRSReplicable extends Isolate{
 constructor(){
  let tm    = new TransactionManager()
  super(new MRSReplicableMirror(tm))
 }
}
```

Listing 6.12: MRS replicas.

Listing 6.12 shows the implementation of MRS replicas. A MRS replica is a Spiders.js isolate with a custom mirror (i.e. a MRSReplicableMirror). Programs create *master* replicas by invoking *MRSReplicable*'s constructor. This creates a *transaction manager* that implements the two-phase commit protocol. We discuss the implementation of this transaction manager after explaining the mirrors for MRS replicas.

```
class MRStrongReplicableMirror extends IsolateMirror{
 coordinator : TransactionManager

 resolve(){
  this.coordinator.registerNewReplica(this)
  return super.resolve()
 }

 access(fieldName){
  return this.coordinator.tryRead().then(()=>{
   return super.access(fieldName)
  })
 }

 write(fieldName,value){
  return this.coordinator.startWriteTransaction(fieldName,value)
 }

 prepare(){
  return true
 }

 commit(fieldName,value){
  return super.write(fieldName,value)
 }
}
```

Listing 6.13: Mirror on MRS replicas.

Listing 6.13 defines the mirror used by all MRS replicas. The mirror overrides Triumvirate's default semantics using three methods. First, a custom implementation of *resolve* is used to register *worker* replicas. When a

MRS replica is deserialised by an actor the mirror registers the deserialised replica as a *worker* replica. Second, reading the field of a MRS replica is asynchronous give that *access* returns a promise. This promise is resolved by the *master* replica's transaction manager when no transactions are ongoing. Third, writing to a replica's field starts a distributed transaction. The actual write only happens when the transaction completes. In other words when the *master* replica's transaction manager invokes the mirror's *commit* method.

```
class TransactionManager{
 replicas : Array<FarRef<MRSReplicableMirror>> = []
 inTransaction       : boolean
 bufferedOps         : Array<Function>

 constructor(){
  this.inTransaction = false
  this.bufferedOps = []
 }

 registerNewReplica(rep : FarRef<TransactionManager>){
  if(this.inTransaction){
   this.bufferedOps.push(()=>{
    this.replicas.push(rep)
   })
  }
  else{
   this.replicas.push(rep)
  }
 }

 tryRead(){
  return new Promise(resolve=>{
   if(inTransaction){
    this.bufferedOps.push(resolve)
   }
   else{
    resolve()
   }
  })
 }

 startWriteTransaction(fieldName,value){
  if(this.inTransaction){
   this.bufferedOps.push(()=>{
    this.startTransaction(fieldName,value)
   })
  }
  else{
   this.startTransaction(fieldName,value)
  }
 }
```

```
43
44  startTransaction(fieldName,value){
45   return new Promise((resolve,reject)=>{
46    let proms = this.replicas.map((replica)=>{
47     return replica.prepare()
48    })
49    let prepareResolved = false
50    Promise.all(proms).then((acks : Array<boolean>)=>{
51     prepareResolved = true
52     let committed  = this.replicas.map((replica)=>{
53      replica.commit(fieldName,value)
54     })
55     Promise.all(committed).then(()=>{
56      this.inTransaction = false
57      this.flushOps()
58      resolve()
59     })
60    })
61    setTimeout(()=>{
62     if(!prepareResolved){
63      reject(new Error("Unable to commit write"))
64     }
65    },5000)
66   })
67  }
68
69  flushOps(){
70   this.bufferedOps.forEach((op : Function)=>{
71    op()
72   })
73  }
74 }
```

Listing 6.14: Implementing the transaction manager.

Listing 6.14 shows the implementation of the transaction manager. A transaction manager allows replicas to register themselves as *workers*, determines when it is safe to read a field and allows replicas to write a value to a field. These three operations can only safely be executed by the transaction manager when it is idle (i.e. if there is no ongoing transaction). To this end the transaction manager maintains a *inTransaction* flag. If one of the three operations is issued while this flag is set to true the transaction manager buffers the operation in the *bufferedOps* array. Buffered operations are automatically performed once the transaction returns to an idle state (i.e. once the *inTransaction* flag is set to false) (see the *flushOps* method on line 69).

Besides the buffering functionality the first two operations' implementations are straightforward. First, registering a new replica (see the *reg-*

isterNewReplica method on line 11) happens by storing the provided reference to the *worker* replica in an array. Second, when a *worker* replica tries to read a field it invokes the transaction manager's *tryRead* method (line 22). The promise returned by this method only resolves when the transaction manager is idle and hence when a replica can safely read the field's value.

The bulk of the transaction manager's code is dedicated to the transaction logic implemented by the *startTransaction* method on line 44. The method is invoked by a replica when it tries to mutate one of its fields after which it immediately returns a promise. The transaction manager resolves the returned promise in case the ensuing transaction completes or rejects it otherwise. The method starts by invoking each registered replica's *prepare* method (line 47). This method always returns true, it solely serves the transaction manager to determine which replicas are connected. If all replicas are online the *proms* promises all resolve and the method continues (line 50). Otherwise, a timeout expires which rejects the return promise and stops the transaction (line 61).

If all replicas are online the transaction manager invokes their *commit* methods (line 53). The transaction manager only returns to the idle state once all *commit* invocations return (line 55).

6.6.3.3 Concluding Remarks and Shortcomings

Our adaptation of the two-phase commit protocol to replicas still exhibits a number of weaknesses. For example, the master replica has the sole responsibility of committing or aborting a transaction. If the master replica crashes or becomes otherwise unavailable no other replica is able to read or write to a field. Ideally our implementation should elect (e.g. through the paxos algorithm [Lam98]) a new master replica when the current one becomes unavailable. We leave this extension to our MRS replicas as future work.

More systematic experiments are needed to validate our claim that Spiders.js' meta facilities allow programmers to easily implement novel kinds of distributed state. Through the implementation of strong eventual replicas and MRS replicas this section informally showcases the ease with which Triumvirate programmers can extend its built-in parameter passing and state update semantics. Spiders.js' mirrors allow programmers to override or extend exact the semantics attributed to various kinds

of distributed state. This allows them to only write the code needed for their novel distributed state's logic while relying on Spiders.js for the boilerplate code.

6.7 Related Work

We are not the first to propose an actor framework for web applications. However, to the best of our knowledge we are the first to design and implement an actor framework able to run locally within a single tier (i.e. client or server) as well as across tiers. What follows is a discussion of actor-based solutions for JavaScript.

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. Web workers run in a completely isolated environment which entails that they do not have access to the lexical scope in which they are created. Moreover, scope isolation also includes graphical elements such as the *document object model* (i.e. *DOM*). 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 6.1.

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

The integration of the CEL model in web applications has already been discussed by related work [MVC11]. So far, the most notable step towards this integration is the Q-connection[5] 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, in Q-connection web workers must explicitly export an object before another worker can acquire a far reference to it. Furthermore, in Q-connection actor references are second class. We discuss this problem in detail in Section 6.1.

[5] `https://github.com/kriskowal/q-connection` (last accessed 31-01-2019)

Akka.js [SPH15] 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 closely resembles Spiders.js in two ways. First, it maps actors onto web workers. Second, it allows for different actor runtimes (i.e. server and client runtimes) to seamlessly communicate. Spiders.js' goal is to provide JavaScript developers a coherent actor-based distributed programming model. However, Akka.js aims to provide Akka/Scala programmers the means to easily deploy their application to JavaScript runtimes.

Generic workers [WHSAT10] 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 in the browser and on a server. Furthermore, generic workers provide the same communication API regardless of the tier in which the communication partner resides. We share the vision that web applications need a unified actor framework. However, Spiders.js explicitly steps away from the traditional web worker interface in favour of a more expressive API through communicating event-loop actors.

Syndicate [GJF16] is an 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. This implementation differs from Spiders.js in two ways. First, Syndicate applications run their actors on the main thread whereas Spiders.js actors have their own thread of control. Second, to the best of our knowledge Syndicate's JavaScript implementation does not allow client-side actors to communicate with server-side actors or client-side actors residing on a different machine.

Ambient.js [GDPDMS18] is a JavaScript/Titanium[6] library which allows the development of cross-platform mobile applications. Spiders.js resembles Ambient.js in two ways: both are heavily influenced by the AmbientTalk [CGS+14] actor language and both implement communicating event-loop actors. However, both the aim as well as the implementation of Ambient.js differs widely from Spiders.js. First, Ambient.js operates in the context of mobile applications where peers are homogeneous (i.e. there is no client/server distinction). Second, actors in Ambient.js do not

[6] `http://www.appcelerator.com` (last accessed 31-01-2019)

operate under their own thread of control (i.e. all actors run atop the main JavaScript event-loop).

Reo@JS [KJ17] is a coordination language for web workers. As is the case for Spiders.js, it aims to provide high-level actor-based abstractions for web applications. Reo@JS focuses on communication patterns between web workers and offers abstractions to easily implement these patterns. However, it does not allow to coordinate workers across different clients or to coordinate client and server workers.

Actrix[7] is an actor library which supports client and server-side actors. Actrix actors are implemented by reusing JavaScript's event-loop (i.e. actors are not supported by their own thread of control. However, Actrix does support communication between actors spanning over client/server boundaries. Spiders.js and Actrix differ in the category of actors they implement, as defined by [DKVCDM16]. Spiders.js actors are instances of the communicating event-loops model while Actrix actors are instances of the *active objects* model. In a nutshell, Actrix actors are represented as a single object. These actors are able to send message to each other (i.e. invoke each other's methods). All arguments to these remote method invocations are passed by copy. Actrix does allow programmers to specify the protocol used to sent these remote method invocations (e.g. socket.io, http,etc.). Concretely, actors communicate through high-level *channels* which hide the actual implementation of message sending. In contrast, Spiders.js communicate through sockets by default. However, using Spiders.js' meta level architecture one could implement custom messaging protocols.

The technique employed by Spiders.js to copy data from an actor's lexical scope at construction time closely resembles Scala's *spores* [MHO14]. In a nutshell, spores allow programmers to create closures which can be safely distributed (e.g. by enforcing that spores and the variables they capture are serialisable). Both approaches rely on the programmer to specify which variables in the actor's or spore's lexical scope are to be captured. However, the spores approach is more substantial as it includes a type system which can enforce safety properties at compile time.

[7] `https://github.com/ismailhabib/actrix` (last accessed 31-01-2019)

6.8 Spiders.js and Requirements for a DRIA-Oriented Programming Model

Spiders.js enables Triumvirate to fulfil requirements R_1 and R_4 for DRIA-oriented programming models as follows:

R_1: The model provides modular abstractions for distributed logic
This requirement is met by Spiders.js' actors. Actors essentially represent parts of a distributed application's logic. Moreover, Spiders.js actors can be sent across the network and instantiated on both client and server and provide built-in communication mechanisms. Finally, programmers are able to compose actor behaviours through inheritance.

R_4: The model allows for extensible parameter passing and state update semantics
This requirement is met by Spiders.js' mirror-based reflection layer. Programmers are able to implement custom mirrors for both actors and objects in Spiders.js. This allows them to implement custom parameter passing and state update semantics. We illustrate this through the implementation of strong eventual replicas and multi-replication strong replicas.

6.9 Chapter Summary

In this chapter we introduce Spiders.js, the TypeScript library on top of which Triumvirate is implemented. More specifically, Triumvirate exposes Spiders.js' actors and meta layer facilities to its programmers. Underlying Spiders.js are JavaScript's primitive concurrency abstractions: web workers that are used client-side and child processes that are used server-side. We identify three shortcomings which preclude programmers from using these constructs to distribute their web application's logic. First, web workers and child processes provide different abstractions, APIs and semantics. Second, values can only be sent between web workers or child processes using pass-by-copy semantics. Third, web workers or child processes are only able to reference each other if they reside on the same host.

Spiders.js claims to solve these issues as follows. First, Spiders.js actors provide the same actor abstraction and API regardless of tiers. Second, Spiders.js supports pass-by-copy semantics for isolates or pass-by-far-reference semantics for all other objects. Third, actor references are first-class language constructs and can be shared between actors and objects. Each Spiders.js actor maps directly onto a web worker or child process. As such, Spiders.js enables vertical as well as horizontal distribution of logic.

Finally, we discuss Spiders.js' metaprogramming interface. In a nutshell, Spiders.js provides a mirror-based reflection layer. Each object and actor in Spiders.js is associated with a mirror which implements its semantics. For example, a single method (i.e. *invoke*) in an object mirror implements object method invocation. Programmers can extend the default object and actor mirrors to extend Spiders.js built-in semantics.

Chapter 7

Conclusion

This final chapter summarises our dissertation. We start by revisiting DRIAs in general and repeat the requirements that a programming model should fulfil in order to support them. Subsequently, we provide a general overview of Triumvirate and how it fulfils these requirements. Finally, we shed light on possible avenues of future work.

7.1 Programming Distributed Rich Internet Applications

Distributed programming models typically allow programmers to express that an object is local or remote with respect to the executing code. For example, this is the case in models such as RMI [Mic98], tuple spaces [Gel85], far references [CGS+14], etc. In this dissertation we argue that the distributed needs of DRIAs require more expressiveness on behalf of distributed programming models.

In recent years the applications running atop the world wide web have become increasingly complex from a distributed point of view. Early web applications were executed by a single centralised server. This server was responsible for the application's logic and state. Clients were only able to view the state or request the server to perform operations over the state. In these early web applications all state was considered local with respect to the server's code. From the viewpoint of the clients' code all state was considered to be remote.

In contrast, a typical DRIA relies on multiple servers. Internally, each server is implemented as multiple independent services as well. We discuss an example of such DRIAs, derived from our collaboration with Emixis, at length in Chapter 2. In general DRIAs distribute both logic and state vertically (i.e. between client and server) as well as horizontally (i.e. amongst servers and amongst clients). In other words, clients and servers require the globally distributed state to be locally accessible in order to execute their logic. As such, DRIAs blur the lines between what is considered local and remote state.

In Section 2.2 of Chapter 2, we identify four requirements that are essential for a DRIA-oriented distributed programming model:

R_1**: The model provides modular abstractions for distributed logic**
The logic of DRIAs is distributed between clients and servers. Moreover, within the servers and clients the logic is also distributed amongst different autonomous components (e.g. threads, web workers, etc). This requires the programmer to divide the application's monolithic logic into multiple units of logic and deploy these components across nodes in the network. A web-oriented programming model should aid the developer in this task by providing modularity abstractions for distributed logic.

R_2**: The model guarantees data consistency properties specified by the programmer**
The state of a DRIA is, similarly to its logic, distributed horizontally and vertically. As such, this state can concurrently be updated by multiple nodes in the network. According to the CAP theorem (see Section 1.1.1 in Chapter 1) each of these concurrent updates trades off consistency, availability and partition tolerance. This requires the programmer to encode this trade-off per operation on a piece of state in their DRIA. A web-oriented programming model should allow programmers to declaratively specify the CAP trade-off for operations on shared state. Moreover, the model should enforce the chosen trade-off throughout the state's lifetime.

R_3**: The model supports multiple parameter passing semantics**
A DRIA moves parts of its state between various nodes in the network. Moreover, maintaining the state's various consistency guaran-

tees requires significant amounts of bookkeeping. This requires the programmer to implement various data dissemination strategies and to maintain the necessary bookkeeping information. A web-oriented programming model should aid developers in this task by providing built-in support for multiple parameter passing semantics. These parameter passing semantics allow programmers to distribute state as first-class values while being freed from manually keeping track of state bookkeeping.

R_4**: The model allows for extensible parameter passing and state update semantics**
In general, the implications of the CAP theorem on distributed systems is still an active field of research [DPMDT+19, LPR18, SBP+18]. As such, new consistency models and conflict resolution algorithms are regularly developed. A web-oriented programming model should allow programmers to extend or override its built-in parameter passing and state update semantics. This allows programmers to easily implement novel consistency models and algorithms within the same programming model.

In Chapter 3 we discuss the state of the art in distributed programming for the web in light of these four requirements. Moreover, Chapter 4 and Chapter 5 discuss specific sub fields within this state of the art in more detail. In summary, all distributed programming models at least partially fulfil one requirement. However, web programmers currently lack a programming model which fulfils all four requirements and are therefore unable to elegantly tackle the complexity of DRIAs using a single model.

7.2 Triumvirate

In this dissertation we have envisioned a programming model that fulfils the aforementioned four requirements. Triumvirate, an object-oriented actor-based distributed programming language, serves as a reference implementation of this model.

7.2.1 The Three Triumvirs

Triumvirate's main novelty is its support for various types of distributed state. Concretely Triumvirate programmers implement their web application's state by extending one of three classes:

Duplicable represents **locally-active** distributed state. Concretely, an actor's control flow determines when and how to apply updates on a duplicate (i.e. an instance of the duplicable class). Moreover, duplicates are sent between actors by copy. When an actor updates a duplicate this only affect the actor's local copy of said duplicate. All primitive Triumvirate data types (e.g. strings, numbers, booleans, etc.) are duplicable.

Replicable represents **remotely-active** distributed state. As is the case for duplicates, updates to replicas (i.e. instances of the replicable class) happen actively (i.e. as dictated by an actor's control flow). In contrast to duplicates, all updates made by an actor to a replica affect all other copies of this replica. To determine which copies to update, Triumvirate employs *pass-by-replica* semantics. Each time a replica is sent between actors a copy is made that keeps a reference to the original replica.

Triumvirate provides two kinds of built-in replicas (i.e. *strong* and *eventual* replicas). Each kind offers a different trade-off with regards to the CAP theorem: strong replicas are consistent and partition-tolerant while eventual replicas are available and partition-tolerant. In both cases the Triumvirate runtime manages concurrent, possibly conflicting, updates and upholds the replica's CAP guarantees. Moreover, as we showcase in Section 6.6.3 of Chapter 6, Triumvirate easily allows one to implement novel kinds of replicable classes.

Derivable represents **reactive** state. Updates to this kind of state are not determined by an actor's control flow. Rather, the reactive part of an application's state automatically updates as a result of updates to the active part of the application's state on which it depends. Triumvirate programmers specify these dependencies using *derivation functions*. These functions take a number of objects (i.e. duplicates, replicas or derivations) as arguments and return a new derivation (i.e. an instance of the derivable class). The return derivation's

state is recomputed each time an update is applied to one of the input objects.

Triumvirate keeps track of the dependencies that span multiple actors (i.e. and therefore possibly multiple physically distributed machines) by employing *pass-by-derivation* parameter passing semantics. In a nutshell, each time a derivation is sent between actors an edge is added to a distributed dependency graph. This graph dictates the order in which derivations are to be updated. As we discuss in Chapter 5, it is important to ensure that these reactive updates happen correctly (i.e. without causing glitches). To this end Triumvirate relies on a novel update algorithm called QPROP. QPROP is the first distributed reactive update algorithm that is able to guarantee this correctness without relying on central coordination.

Programmers combine duplicates, replicas and derivations to implement their DRIAs' distributed state. Triumvirate curtails these combinations of distributed state through a set of laws to prevent programmers from breaking the guarantees given by duplicates, replicas and derivations.

7.2.2 Curtailing Interactions with Triumvirate's Laws

Figure 7.1 shows how duplicates, replicas and derivations interact. In a nutshell, Triumvirate separates the three kinds of distributed state into *active* and *reactive* state. Active state (i.e. duplicates and strong or eventual replicas) is imperatively updated by actors. As we discuss in Section 4.2 of Chapter 4, the **preservation of availability law** (i.e. Law 1) and the **preservation of consistency law** (i.e. Law 2) prohibit any interaction between strong and eventual replicas. However, Triumvirate allows programmers to convert strong replicas to eventual replicas and vice versa using two built-in functions: *thaw* and *freeze* respectively. Triumvirate guarantees that replicas can safely be sent between actors by enforcing the **preservation of serialisability law** (i.e. Law 3).

Reactive state (i.e. derivations) updates automatically as a result of imperative updates to active state. Active and reactive state inherently provide two different programming paradigms (i.e. imperative and declarative programming respectively). As we discuss in Section 5.3 of Chapter 5, Triumvirate separates active from reactive state using the **activity-**

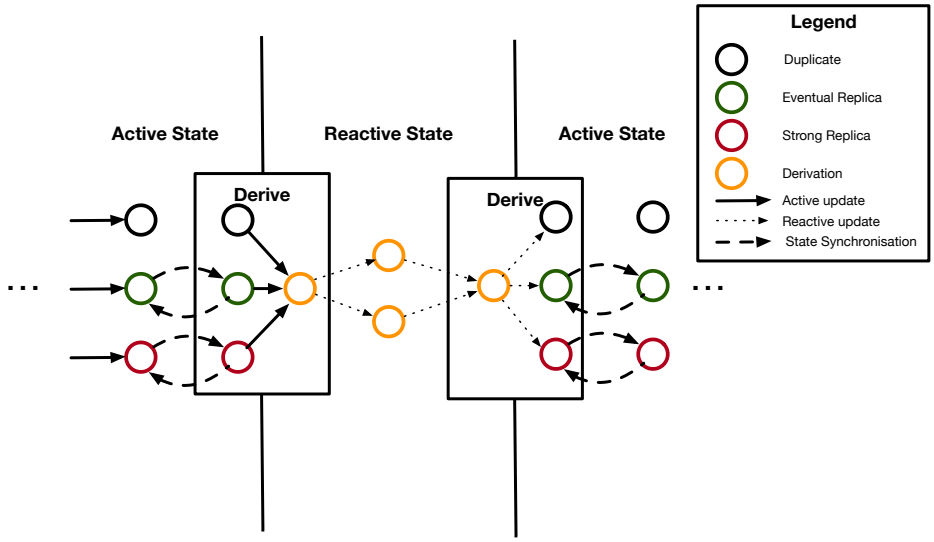


Figure 7.1: Overview of state interactions within Triumvirate.

reactivity isolation law (i.e. Law 4). However, programmers are able to translate active updates to reactive ones and vice versa using Triumvirate actors' built-in *derive* function.

7.2.3 Triumvirate: a DRIA-oriented Programming Model

Triumvirate fulfils the requirements for a DRIA-oriented programming model as follows:

R_1: **The model provides modular abstractions for distributed logic**

Triumvirate programmers use **actors** to implement their application's logic. These actors are modular and composable units which can readily be deployed both on the client as well as the server. Besides offering horizontal and vertical distribution, Triumvirate actors therefore also allow programmers to implement parallel applications.

Triumvirate actors provide two built-in communication mechanisms. First, actors explicitly communicate by **invoking** each other's methods (i.e. by sending asynchronous messages). Second, actors implicitly communicate with each other by **updating** distributed shared state (i.e. replicas or duplicates).

R_2: The model guarantees data consistency properties specified by the programmer

Triumvirate natively supports locally-active, remotely-active and reactive state in the form of **duplicates**, **replicas** and **derivations** respectively. Triumvirate maintains the consistency guarantees of a piece of distributed state throughout its lifetime as follows:

Duplicates guarantee availability and partition tolerance, but do not provide any consistency. As such, Triumvirate does not implement any specific update mechanisms for duplicates.

Replicas are sub-divided into strong and eventual replicas. Strong replicas guarantee partition tolerance and strong consistency. Triumvirate guarantees this by implementing strong replicas as far references [CGS+14]. Eventual replicas guarantee availability, partition tolerance and non-strong eventual consistency. Triumvirate ensures this guarantee by implementing eventual replicas using the global sequence protocol [BLPF15]. Moreover, in Section 6.6.3 of Chapter 6 we implement two additional types of replicas. The first, enhances strong replicas by guaranteeing strong consistency for multiple replicas using two-phase commit [LS76]. The second, enhances eventual replicas by guaranteeing strong eventual consistency through the use of conflict-free replicated datatypes [SPBZ11].

Derivations guarantee availability, partition tolerance and eventual consistency. Triumvirate ensures this guarantee through a novel reactive propagation algorithm called QPROP (see Chapter 5). This algorithm guarantees that derivations update glitch freely without resorting to central coordination.

As we discuss in the previous section, Triumvirate stipulates interaction rules between duplicates, replicas and derivations.

R_3: The model supports multiple parameter passing semantics

Triumvirate supports **three parameter passing semantics**: *pass-by-copy*, *pass-by-replica* and *pass-by-derivation*. Each of these semantics provide support for the three major categories of distributed state (i.e. duplicable, replicable and derivable respectively). The Triumvirate runtime automatically chooses the right parameter passing semantic when a piece of distributed state is sent between actors.

This frees the programmer from manually serialising and deserialising distributed state.

R_4: **The model allows for extensible parameter passing and state update semantics**
Triumvirate allows programmers to extend its built-in semantics by implementing custom mirrors at the **meta level**. These mirrors offer a number of overridable hooks into the life cycle of actors and distributed objects. As we showcase in Section 6.6.3 of Chapter 6 these hooks allow programmers to easily implement novel kinds of distributed state.

7.3 Overview of the Contributions

The general vision (see Chapter 2) behind this dissertation is too vast to rigorously realise and scientifically validate in the course of a single doctoral dissertation. As such, we divide this dissertation into three concrete contributions to the vision that have been validated and published at peer-reviewed venues. These contributions are the following:

- **Spiders.js** [MSDM18b] is the first programming language for web applications that uses communicating event-loop actor as its core model. Programmers implement clients and servers as collections of actors. Spiders.js offers a single API regardless of an actor's locality (i.e. client or server-side). Therefore, Spiders.js is the first language to unify distribution and parallelism for web applications across client and server tiers.

  Spiders.js was heavily influenced by the AmbientTalk [CGS+14] language and as such employs a mirror-based reflection layer. This allows programmers to extend or override Spiders.js' built-in semantics by implementing custom mirrors on objects or actors. We showcase the strength of this approach by implementing two novel kinds of distributed state (i.e. strong eventual replicas and multi-replication strong replicas) in Section 6.6.3 of Chapter 6.

  Our benchmarks showcase that Spiders.js introduces a performance overhead compared to JavaScript. However, our benchmarks also showcase that this overhead does not compromise Spiders.js' ability to parallelise web applications. Moreover, qualitative benchmarks

show that Spiders.js frees the programmer from writing boilerplate code thereby allowing them to focus on their DRIA's logic.

- **Triumvirate** is the first distributed programming model to explicitly classify distributed state into three categories: duplicable, replicable [MSDM18a] and derivable [MSDM19]. The Triumvirate programming language incorporates these into its design as first-class replicated objects. We showcase the applicability of one of these classes (i.e. replicable) during a real-world experiment conducted at the Onward!2018 conference[1].

- Triumvirate guarantees the **eventual consistency** of derivations. Moreover, Triumvirate guarantees that derivations update without causing **glitches**. To do so we develop the first propagation algorithm to guarantee glitch freedom of distributed reactive applications without resorting to central coordination: QPROP [MSDM19]. We prove the correctness of our algorithm and show that it significantly outperforms existing approaches through a set of benchmarks.

7.4 Triumvirate Beyond the Current Mandate

Triumvirate serves as an initial prototype of a programming language tailored to DRIAs. This dissertations showcases and validates the core ideas behind Triumvirate. However, given the novelty of Triumvirate and its underlying concepts it also opens a number of avenues for future research. Concretely, we foresee the following as future work to improve Triumvirate as a language and programming model:

Overall Performance Spiders.js, and by extension Triumvirate, suffers from a lack of performance on two fronts. First, Spiders.js introduces a non-negligible overhead (i.e. up to 5X) compared to JavaScript. This is largely due to the fact that Spiders.js heavily relies on JavaScript's meta-level facilities. For example, objects in Spiders.js are wrapped by JavaScript proxies in order to implement Spiders.js' mirror-based reflection. Moreover, distributed objects (i.e. duplicates, replicas or derivations) in Triumvirate extend

[1] `https://www.youtube.com/watch?v=17cHhDDpJbg`

Spiders.js' built-in mirrors thereby adding an additional meta-layer. We showcase this overhead in Section 6.6.1.1 of Chapter 6.

Spiders.js also lacks performance with regards to its memory consumption. More precisely, Spiders.js currently does not perform any distributed garbage collection. For example, far references held by actors are kept indefinitely in the actor's memory.

We only encountered memory-related issues in long-running Spiders.js prototypes at Emixis. However, distributed garbage collection is necessary for Triumvirate and Spiders.js to be usable as platforms for extensive real-world scenarios. Triumvirate and Spiders.js can benefit from a number of existing approaches with regards to distributed garbage collection. These range from general-purpose distributed garbage collection approaches [AR98] to more specific methods such as *tombstoning* for conflict-free replicated data types [SPBZ11]. We have already started research into garbage collection for derivations [MSDM17].

Mixing Consistency Models Triumvirate allows programmers to implement applications which require various consistency models for their state. For example, using strong and eventual replicas in a same application effectively mixes sequential and non-strong eventual consistency.

As we discuss in Section 4.2 of Chapter 4 and Section 5.3.3 of Chapter 5 we restrict the direct interactions between distributed objects. However, programmers are still able to combine values read from different distributed objects. For example, imagine two counters: one implemented as a strong replica and the other as an eventual replica. Both counters maintain a single numerical field *value* (i.e. a duplicate). Triumvirate does not prohibit an application from reading and combining the values of both counters (e.g. printing these out to a console). This combination does not infringe on the interaction rules of Triumvirate given that both values are technically duplicates. However, from an application perspective both counter values could offer two views with different levels of consistency over the same data. As such, combining these values could lead to application-specific inconsistencies.

Languages such as MixT [MM18] previously identified this problem. In the specific case of MixT the solution comes in the form of a type system which tracks the flow of values with varying consistency levels through the program. MixT's type system is able to statically determine whether a program is correct (i.e. if generalised non-interference can be enforced). However, MixT implements distributed objects as elements in replicated data stores. This contrasts with Triumvirate where distributed objects are first-class entities. An avenue of future work is therefore to investigate to what extend type systems such as the one used in MixT are portable to the Triumvirate programming model.

Extensible Interaction Rules Triumvirate implicitly enforces a number of rules with regards to the interaction between duplicates, replicas and derivations. For example, programmers cannot apply derivation functions to a mix of replicas and derivations or arbitrarily compose strong and eventual replicas. These rules are hardcoded in Triumvirate's runtime and cannot readily be extended by the programmer. For example, in Section 6.6.3 of Chapter 6 we implement two novel kinds of replicas. However, the programmer would have to manually reimplement the correct rules for both novel replicas.

An avenue of future research is therefore to provide programmers with a declarative way to specify interaction rules between distributed objects (e.g. through annotations). As is the case for Spiders.js' mirrors, programmers could extend or override certain rules for their custom distributed objects. Moreover, Triumvirate could statically verify whether an application is correct using these declaratively specified rules.

Programming Support Triumvirate currently enforces its laws through runtime exceptions. This technique solely guarantees that the Triumvirate laws cannot be broken by developers. However, these runtime exceptions do not provide the support programmers need to *avoid* breaking the Triumvirate laws. For example, imagine a scenario in which a Triumvirate developer relies on a third-party Triumvirate library to implement his DRIA. There are two ways for the developer to detect that its use of the library breaks Triumvirate laws. First, by running the DRIA and catching runtime exceptions.

Second, by manually going through the implementation of the library and checking its usage of it for any violations of Triumvirate's laws.

Ideally, Triumvirate should provide the necessary tools for the programmer to pre-emptively and automatically catch any violation of Triumvirate's laws. We foresee two possible avenues of future work to solve this lack of programmer support on behalf of Triumvirate.

1. Extending TypeScript's type system would allow us to cover most violations of Triumvirate's laws. For example, Laws 1 and 2 can be enforced by restricting the type of arguments one can provide to methods of strong and eventual replicas. In the case of Law 4 one could rely on TypeScript's conditional types to ensure that the *derive* function cannot be invoked with a combination of active and reactive state as input arguments. A *spore*-like solution [MHO14] would allow for the static enforcement of Law 3.
2. A static analysis tool could symbolically execute [Kin76] DRIAs before they are deployed onto a network of nodes. Subsequently, the tool could detect violations of Triumvirate's laws and inform the programmer of said violations. Moreover, the tool could use the symbolic information gathered during the execution to provide examples of the input data that lead to the violation. However, it is unclear whether this solution would scale for larger Triumvirate applications.

7.5 Closing Remarks

Programmers of distributed applications face a number of challenges that are inherent to distributed systems (e.g. concurrency, lack of central resource management, etc. [KWWW94]). This dissertation focuses on distributed applications for which the logic is disseminated across multiple nodes of a distributed computing platform. As an example of such a platform this dissertation focuses on the world wide web. DRIAs are characterised by a number of servers and clients all concurrently running parts of the application's logic. As a result, each server or client requires part of the application's state in order to execute its logic.

Current distributed programming models do not provide the abstractions necessary to elegantly tackle the complexities of DRIAs. We identify four requirements for a programming model to do so. First, the model should provide modular abstractions for distributed logic. Second, the model should support multiple parameter passing semantics. Third, the model should allow programmers to specify consistency guarantees about the application's state. Fourth, the model should allow programmers to extend or override its semantics.

This dissertation aims to contribute to a new generation of programming languages tailored to DRIAs. We present Triumvirate, a novel programming model which fulfils the aforementioned four requirements. Key to Triumvirate are its triumvirs: three classes of objects (i.e. duplicable, replicable and derivable) that allow programmers to implement their application's distributed state. These triumvirs are used by communicating event-loop actors to execute the application's distributed logic.

This dissertation presents a first implementation of Triumvirate atop Spiders.js [MSDM18b], a general-purpose distributed programming language for the web. Spiders.js unifies distribution and parallelism for web applications by providing a single actor model for both server and client-side web applications. Moreover, Spiders.js provides a mirror-based reflection API that allows programmers to extend and override its built-in semantics.

Triumvirate uses Spiders.js meta-facilities to implement the three triumvirs. Replicas [MSDM18a] come in two flavours, eventual replicas are partition-tolerant and eventually consistent while strong replicas are partition-tolerant and strongly consistent. Both kinds of replicas rely on

state-of-the-art update algorithms to handle concurrent updates. In this dissertation we present two implementations of eventual replicas (i.e. one using the global sequence protocol [BLPF15] and another using conflict-free replicated data types [SPBZ11]) and two implementations of strong replicas (i.e. one using far references [CGS+14] and another using the two-phase commit protocol [LS76]).

Concurrent updates to derivations are handled by a custom propagation algorithm called QPROP [MSDM19]. QPROP is the first reactive propagation algorithm for distributed systems that can guarantee correctness without relying on central coordination.

Appendix A

Code Complexity Comparison

A.1 Highlighted Native Implementation of Pong

A.1.1 Server Implementation

```
var io = require('socket.io');
var socket = io(8000);
var clients = new Map();
var games = new Map();
var occupation = new Map();
class Player {
    constructor(ref, name) {
        this.ref = ref;
        this.name = name;
    }
}
function newClient(nickName, ref) {
    clients.set(nickName, new Player(ref, nickName));
    games.forEach((creator, roomName) => {
        ref.emit('message', ["newGameCreated", roomName, creator.name]);
        if (occupation.get(roomName) > 1) {
            ref.emit('message', ["updateRoomInfo", roomName]);
        }
    });
}
function createNewGame(roomName, creatorRef, creatorName) {
    games.set(roomName, new Player(creatorRef, creatorName));
    occupation.set(roomName, 1);
    clients.forEach((client) => {
        if (client.name != creatorName) {
            client.ref.emit('message', ["newGameCreated", roomName, creatorName]);
        }
    });
}
function playerJoined(roomName, playerName) {
    var otherPlayer = games.get(roomName);
    occupation.set(roomName, occupation.get(roomName) + 1);
    clients.forEach((client) => {
        if (client.name != playerName && client.name != otherPlayer.name) {
            client.ref.emit('message', ["updateRoomInfo", roomName]);
        }
    });
}
function forwardPlayerJoins(to, playerNick) {
    var player = clients.get(to);
    player.ref.emit('message', ["playerJoins", playerNick]);
}
function forwardGetPortal(to, playerNick) {
    var player = clients.get(to);
    player.ref.emit('message', ["getPortal", playerNick]);
}
function forwardReceivePortal(to, x, y, r, c) {
    var player = clients.get(to);
    player.ref.emit('message', ["receivePortal", x, y, r, c]);
}
function forwardReceiveBall(to, x, y, vx, vy) {
    var player = clients.get(to);
    player.ref.emit('message', ["receiveBall", x, y, vx, vy]);
}
function forwardScoreChange(to, score) {
    var player = clients.get(to);
    player.ref.emit('message', ["scoreChange", score]);
}
function forwardReceivePowerup(to, type) {
    var player = clients.get(to);
```

A.1.2 Server Implementation

```
    player.ref.emit('message', ["receivePowerup", type]);
}
socket.on('connect', (client) => {
    client.on('message', (data) => {
        switch (data[0]) {
            case "newClient":
                newClient(data[1], client);
                break;
            case "createNewGame":
                createNewGame(data[1], client, data[2]);
                break;
            case "playerJoined":
                playerJoined(data[1], data[2]);
                break;
            case "forwardPlayerJoins":
                forwardPlayerJoins(data[1], data[2]);
                break;
            case "forwardGetPortal":
                forwardGetPortal(data[1], data[2]);
                break;
            case "forwardReceivePortal":
                forwardReceivePortal(data[1], data[2], data[3], data[4], data[5]);
                break;
            case "forwardReceiveBall":
                forwardReceiveBall(data[1], data[2], data[3], data[4], data[5]);
                break;
            case "forwardScoreChange":
                forwardScoreChange(data[1], data[2]);
                break;
            case "forwardReceivePowerup":
                forwardReceivePowerup(data[1], data[2]);
                break;
            default:
                console.log("Server did not understand message : " + data[0]);
        }
    });
});
```

A.1.3 Client Implementation

```
import {Socket} from "net";
var graph = require('./graphics')

class NativePongClient{
    serverRef   : Socket
    nickName    : string
    currentGame

    constructor(nickName : string){
        this.serverRef  = require('socket.io-client')('http://127.0.0.1:8000')
        this.nickName   = nickName
        var that        = this
        this.serverRef.on('message',(data)=>{
            switch(data[0]){
                case "updateRoomInfo":
                    that.updateRoomInfo(data[1])
                    break
                case "newGameCreated":
                    that.newGameCreated(data[1],data[2])
                    break
                case "playerJoins":
                    that.playerJoins(data[1])
                    break
                case "getPortal":
                    that.getPortal(data[1])
                    break
                case "receivePortal":
                    that.receivePortal(data[1],data[2],data[3],data[4])
                    break
                case "receiveBall":
                    that.receiveBall(data[1],data[2],data[3],data[4])
                    break
                case "scoreChange":
                    that.scoreChange(data[1])
                    break
                case "receivePowerup":
                    that.receivePowerup(data[1])
                    break
                default:
                    console.log("Client did not understand message : " + data[0])
            }
        })
        this.serverRef.emit('message',["newClient",this.nickName])
        document.getElementById("newRoomButton").onclick = () => {
            var roomName = (document.getElementById('roomName') as
HTMLTextAreaElement).value
            this.currentGame = new graph.game(that,roomName)
            this.serverRef.emit('message',["createNewGame",roomName,this.nickName])
            this.currentGame.start(true)
        }
    }

    private joinGame(roomName : string,gameCreator : string){
        this.currentGame        = new graph.game(this,roomName)
        this.currentGame.setOpponentReference(gameCreator)
        this.serverRef.emit('message',["playerJoined",roomName,this.nickName])
        this.serverRef.emit('message',["forwardPlayerJoins",gameCreator,this.nickName])
        this.serverRef.emit('message',["forwardGetPortal",gameCreator,this.nickName])
        this.currentGame.start(false);
    }
```

A.1.4 Client Implementation

```
    newGameCreated(roomName : string,gameCreator : string){
        var row                 = (document.getElementById('roomList') as
HTMLTableElement).insertRow()
        var nameCell            = row.insertCell()
        var noPlayersCell       = row.insertCell()
        row.id                  = roomName
        nameCell.innerHTML      = roomName
        noPlayersCell.innerHTML = "1/2"
        var that                = this
        row.onclick             = function() {
            if(noPlayersCell.innerHTML === "1/2") {
                that.joinGame(roomName,gameCreator)
            }
        };
    }

    playerJoins(nickName : string){
        this.currentGame.setOpponentReference(nickName)
        this.currentGame.playerJoined(nickName)
    }

    updateRoomInfo(roomName){
        (document.getElementById(roomName) as any).cells[1].innerHTML = "2/2"
    }

    getPortal(requester : string){
        var gamePortal = this.currentGame.getPortal()
        this.serverRef.emit('message',
["forwardReceivePortal",requester,gamePortal.x,gamePortal.y,gamePortal.r,gamePortal.c])
    }

    receivePortal(x,y,r,c){
        this.currentGame.receivePortal({x:x,y:y,r:r,c:c})
    }

    receiveBall(x,y,vx,vy){
        this.currentGame.receiveBall({ x: x, y: y, vx: vx, vy: vy })
    }

    //Invoked by the UI
    sendBallTo(opponent : string,x,y,vx,vy){
        this.serverRef.emit('message',["forwardReceiveBall",opponent,x,y,vx,vy])
    }

    //Invoked by the UI
    sendScoreChangeTo(opponent : string,score){
        this.serverRef.emit('message',["forwardScoreChange",opponent,score])
    }

    //Invoked by the UI
    sendPowerupTo(opponent : string,type){
        this.serverRef.emit('message',["forwardReceivePowerup",opponent,type])
    }

    scoreChange(score : number){
        this.currentGame.receiveOpponentScore(score)
    }

    receivePowerup(type : string){
```

A.1.5 Client Implementation

```
            this.currentGame.receivePowerup(type)
        }
    }
    (window as any).start = ()=>{
        var nickName              = (document.getElementById('nickname') as
    HTMLTextAreaElement).value
        new NativePongClient(nickName)
    }
```

A.2 Highlighted Spiders.ts Implementation of Pong

A.2.1 Server Implementation

```
import {SpiderLib, FarRef} from "../src/spiders";
var spiders : SpiderLib = require("../src/spiders")

class Player{
    ref     : FarRef
    name    : string
    constructor(ref,name){
        this.ref    = ref
        this.name   = name
    }
}
class SpiderPongServer extends spiders.Application{
    games       : Map<string,Player>
    occupation  : Map<string,number>
    clients     : Map<string,Player>

    constructor(){
        super()
        this.games      = new Map()
        this.clients    = new Map()
        this.occupation = new Map()
    }

    newClient(nickName : string,ref : FarRef){
        this.clients.set(nickName,new Player(ref,nickName))
        this.games.forEach((creator : Player,roomName : string)=>{
            ref.newGameCreated(roomName,creator.ref)
            if(this.occupation.get(roomName) > 1){
                ref.updateRoomInfo(roomName)
            }
        })

    }

    createNewGame(roomName : string,creatorRef : FarRef,creatorName){
        this.games.set(roomName,new Player(creatorRef,creatorName))
        this.occupation.set(roomName,1)
        this.clients.forEach((client : Player)=>{
            if(client.name != creatorName){
                client.ref.newGameCreated(roomName,creatorRef)
            }
        })
    }

    playerJoined(roomName : string,playerRef : FarRef,playerName : string){
        var otherPlayer : Player = this.games.get(roomName)
        this.occupation.set(roomName,this.occupation.get(roomName)+1)
        this.clients.forEach((client : Player)=>{
            if(client.name != playerName && client.name != otherPlayer.name){
                client.ref.updateRoomInfo(roomName)
            }
        })
    }
}
new SpiderPongServer()
```

A.2.2 Client Implementation

```
import {SpiderLib, FarRef} from "../../../src/spiders";
var spiders : SpiderLib = require("../../../src/spiders")
var graph               = require("./graphics")


class PortalIsolate extends spiders.Isolate{
    x
    y
    r
    c
    constructor(x,y,r,c){
        super()
        this.x = x
        this.y = y
        this.r = r
        this.c = c
    }
}

class SpiderPongClient extends spiders.Application{
    serverRef    : FarRef
    nickName     : string
    currentGame

    constructor(nickName : string){
        super()
        this.nickName = nickName
        this.remote("127.0.0.1",8000).then((serverRef : FarRef)=>{
            this.serverRef = serverRef
            serverRef.newClient(this.nickName,this)
        })
        document.getElementById("newRoomButton").onclick = () => {
            var roomName = (document.getElementById('roomName') as
HTMLTextAreaElement).value
            this.currentGame = new graph.game(roomName,this)
            this.serverRef.createNewGame(roomName,this,this.nickName)
            this.currentGame.start(true)
        }
    }

    private joinGame(roomName : string,gameCreator : FarRef){
        this.currentGame         = new graph.game(roomName,this)
        this.currentGame.setOpponentReference(gameCreator)
        this.serverRef.playerJoined(roomName,this,this.nickName)
        gameCreator.playerJoins(this,this.nickName)
        gameCreator.getPortal().then((portal)=>{
            this.currentGame.receivePortal(portal)
        })
        this.currentGame.start(false);
    }

    newGameCreated(roomName : string,gameCreator : FarRef){
        var row                 = (document.getElementById('roomList') as
HTMLTableElement).insertRow()
        var nameCell            = row.insertCell()
        var noPlayersCell       = row.insertCell()
        row.id                  = roomName
        nameCell.innerHTML      = roomName
        noPlayersCell.innerHTML = "1/2"
        var that                = this
```

A.2.3 Client Implementation

```
            row.onclick             = function() {
                if(noPlayersCell.innerHTML === "1/2") {
                    that.joinGame(roomName,gameCreator)
                }
            };
        }

        playerJoins(player : FarRef,nickName : string){
            this.currentGame.setOpponentReference(player)
            this.currentGame.playerJoined(nickName)
        }

        updateRoomInfo(roomName){
            (document.getElementById(roomName) as any).cells[1].innerHTML = "2/2"
        }

        getPortal(){
            var gamePortal = this.currentGame.getPortal()
            return new PortalIsolate(gamePortal.x,gamePortal.y,gamePortal.r,gamePortal.c)
        }

        receiveBall(x,y,vx,vy){
            this.currentGame.receiveBall({ x: x, y: y, vx: vx, vy: vy })
        }

        scoreChange(score : number){
            this.currentGame.receiveOpponentScore(score)
        }

        receivePowerup(type : string){
            this.currentGame.receivePowerup(type)
        }

        //Invoked by UI
        sendBall(x,y,vx,vy,ref){
            ref.receiveBall(x,y,vx,vy)
        }

        //Invoked by UI
        sendScoreChange(score,ref){
            ref.scoreChange(score)
        }

        //Invoked by UI
        sendPowerup(type,ref){
            ref.receivePowerup(type)
        }
}

(window as any).start = ()=>{
    var nickName            = (document.getElementById('nickname') as
HTMLTextAreaElement).value
    new SpiderPongClient(nickName)
}
```

Bibliography

[ACHM11] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, pages 249–260. Citeseer, 2011.

[ADNL15] Marina Andrić, Rocco De Nicola, and Alberto Lluch Lafuente. Replica-based high-performance tuple space computing. In *Proceedings of the 17th International Conference on Coordination Languages and Models*, pages 3–18. Springer, 2015.

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[Baq15] Carlos Baquero. delta-enabled-crdts. `https://github.com/CBaquero/delta-enabled-crdts`, 2015. Accessed: 2019-07-19.

[BB16] P. A. Bernstein and S. Bykov. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing*, 20(5):71–75, Sep. 2016.

[BBB+17] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. *Proceedings of the*

ACM on Programming Languages, 1(OOPSLA):107:1–107:26, October 2017.

[BC18] Sebastian Burckhardt and Tim Coppieters. Reactive caching for composed services: Polling at the speed of push. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):152:1–152:28, October 2018.

[BCC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[BCLG00] Albert Benveniste, Benoit Caillaud, and Paul Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. Technical report, 2000.

[BFLW12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BKS13] Henri Binsztok, Adam Koprowski, and Ida Swarczewskaja. *Opa: Up and Running: Rapid and Secure Web Development.* " O'Reilly Media, Inc.", 2013.

[BLPF15] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared

State. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, 2015.

[BMG18] Jim Bauwens, Florian Myter, and Elisa Gonzalez Boix. Constraining the eventual in eventual consistency. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '18, pages 2:1–2:3, 2018.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM.

[Cap14] Linda Caputo. Multithreaded recalculation in excel, 2014. Accessed: 19-11-2016.

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[CDMB16] Tim Coppieters, Wolfgang De Meuter, and Sebastian Burckhardt. Serializable eventual consistency: Consistency through object method replay. In *Proceedings of the 2nd Workshop on the Principles and Practice*

of Consistency for Distributed Data, PaPoC '16, pages 3:1–3:3, New York, NY, USA, 2016. ACM.

[CGS+14] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3-4):112 – 136, 2014.

[CJPR+07] Alfrânio Correia Jr, José Pereira, Luís Rodrigues, Nuno Carvalho, Ricardo Vilaça, Rui Oliveira, and Susana Guedes. Gorda: An open architecture for database replication. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications*, pages 287–290. IEEE, 2007.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer, 2006.

[CMA+12] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM.

[CMVCDM10] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Proceedings of the 48th International Conference on Objects, Models, Components,*

Patterns, TOOLS'10, pages 41–60, Berlin, Heidelberg, 2010. Springer-Verlag.

[DFWB98] Nigel Davies, Adrian Friday, Stephen P Wade, and Gordon S Blair. L 2 imbo: a distributed systems platform for mobile computing. *Mobile Networks and Applications*, 3(2):143–156, 1998.

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer International Publishing, 2017.

[DKVCDM16] Joeri De Koster, Tom Van Cutsem, and Wolfgang 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*, AGERE 2016, pages 31–40, New York, NY, USA, 2016. ACM.

[DPMDT+19] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. Putting order in strong eventual consistency. In *Proceedings of the 19th International Conference on Distributed Applications and Interoperable Systems*, pages 36–56. Springer, 2019.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 434–442, 1986.

[DSMM14] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed rescala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[ecm19] Ecmascript 2018 language specification. `https://www.ecma-international.org/ecma-262/9.0/index.html#Title`, 2019. Accessed: 2019-05-15.

[FCBC10] Piero Fraternali, Sara Comai, Alessandro Bozzon, and Giovanni Toffetti Carughi. Engineering rich internet applications with a model-driven approach. *ACM Transactions on the Web*, 4(2):7, 2010.

[FRSF10] Piero Fraternali, Gustavo Rossi, and Fernando Sánchez-Figueroa. Rich internet applications. *IEEE Internet Computing*, 14(3):9–12, 2010.

[GDPDMS18] Elisa Gonzalez Boix, Kevin De Porre, Wolfgang De Meuter, and Christophe Scholliers. Ambientjs: A mobile cross-platform actor library for multi-networked mobile applications. In *Programming with Actors: State-of-the-Art and Research Perspectives*, pages 32–58. Springer International Publishing, 2018.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GG10] A. Gamatie and T. Gautier. The signal synchronous multiclock approach to the design of distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):641–657, May 2010.

[GJF16] Tony Garnock-Jones and Matthias Felleisen. Coordinated concurrent programming in syndicate. In Peter Thiemann, editor, *Proceedings of the 25th European Symposium on Programming*, pages 310–336, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GPS16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA, 2016. USENIX Association.

[Gro12] Object Management Group. Common object request broker architecture, 2012. Accessed: 2019-06-03.

[gRP06] grpc: A high performance, open-source uni-versal rpc framework. `https://grpc.io`, 2006. Accessed: 2019-05-10.

[GSMD14] Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, and Theo D'Hondt. Programming mobile context-aware applications with totam. *Journal of Systems and Software*, 92:3 – 19, 2014.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HBZ+16] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM.

[HSH05] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Technical Report UCAM-CL-TR-633, University of Cambridge, Computer Laboratory, May 2005.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[ICK06] Daniel Ignatoff, Gregory H Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, pages 259–276. Springer, 2006.

[IS14] Shams M. Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80, New York, NY, USA, 2014. ACM.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[KB17] M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, Oct 2017.

[KDRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KJ17] Marco Krauweel and Sung-Shik Jongmans. Simpler Coordination of JavaScript Web Workers. In *Proceedings of the 19th International Conference on Coordination Languages and Models*, pages 40–58, June 2017.

[KKW19] Maciej Kokociński, Tadeusz Kobus, and Paweł T Wojciechowski. On mixing eventual and strong consistency: Bayou revisited. *arXiv preprint arXiv:1905.11762*, 2019.

[KLL+97] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, volume 97, page 481. ACM, 1997.

[KWWW94] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[LGTLL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(03):261–303, 2003.

[LLC+14] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, 2014. USENIX Association.

[LPR18] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 359–372, Boston, MA, 2018. USENIX Association.

[LS76] Butler Lampson and H Sturgis. Crash recovery in a distributed system. Technical report, Xerox PARC, 1976.

[MBS+18] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant Distributed Reactive Programming. In *Proceedings of the 32nd European Conference on Object-Oriented Programming*, volume 109, pages 1:1–1:26, 2018.

[McA95] Jeff McAffer. Meta-level programming with coda. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the 9th European Conference on Object Oriented Programming*, pages 190–214, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[MCSDM16] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2016, pages 1–8, 2016.

[Mei17] Christopher S. Meiklejohn. On the design of distributed programming models. In *Proceedings of the 2nd workshop on Programming Models and Languages for Distributed Computing*, PMLDC '17, pages 1:1–1:6, New York, NY, USA, 2017. ACM.

[MGB+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[MHO14] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *Proceedings of the 28th European Conference Object-Oriented Pro-*

gramming, pages 308–333, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[Mic98] Sun Microsystems. Java remote method invocation specification, 1998. Accessed: 2019-06-03.

[MM18] Matthew Milano and Andrew C. Myers. Mixt: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM.

[MPR01] A. L. Murphy, G. P. Picco, and G. . Roman. Lime: a middleware for physical and logical mobility. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[MS14] Alessandro Margara and Guido Salvaneschi. We have a dream: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA, 2014. ACM.

[MS17] Alessandro Margara and Guido Salvaneschi. Consistency types for safe and efficient distributed programming. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs*, FTFJP'17, pages 8:1–8:2, New York, NY, USA, 2017. ACM.

[MSDM16] Florian Myter, Christophe Scholliers, and Wolfgang 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*, AGERE 2016, pages 51–60, 2016.

[MSDM17] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Handling partial failures in distributed reactive programming. In *Proceedings of the 4th ACM*

SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS 2017, pages 1–7, 2017.

[MSDM18a] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. A capable distributed programming model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 88–98, 2018.

[MSDM18b] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Parallel and distributed web programming with actors. In *Programming with Actors: State-of-the-Art and Research Perspectives*, pages 3–31. Springer International Publishing, 2018.

[MSDM19] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *The Art, Science, and Engineering of Programming*, 3(3), 2019.

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan 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.

[MVC11] Mark S. Miller and Tom Van Cutsem. Communicating event loops, an exploration in javascript. http://soft.vub.ac.be/ tvcutsem/talks/presentations/WGLD_CommEventLoops.pdf, 2011.

[MVCT+09] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in ambienttalk. *Software: Practice and Experience*, 39(7):661–699, 2009.

[MVR15] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195, New York, NY, USA, 2015. ACM.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications*, PERCOM '04, pages 263–, Washington, DC, USA, 2004. IEEE Computer Society.

[OIT92] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Al-1/d: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture*, volume 11, 1992.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association.

[PB17] José Proença and Carlos Baquero. Quality-aware reactive programming for the internet of things. In *Proceedings of the 20th International Conference on Fundamentals of Software Engineering*, pages 180–195. Springer, 2017.

[PBCB06] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[PDMDR15] Laure Philips, Wolfgang De Meuter, and Coen De Roover. Tierless programming in javascript. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 831–832. IEEE Press, 2015.

[PDRVCDM14] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. Towards tierless web development without tierless languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 69–81, New York, NY, USA, 2014. ACM.

[PLCSF07] Juan Carlos Preciado, Marino Linaje, Sara Comai, and Fernando Sánchez-Figueroa. Designing rich internet applications with web engineering methodologies. In *Proceedings of the 9th IEEE International Workshop on Web Site Evolution*, pages 23–30. IEEE, 2007.

[RDP14] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 55–68, New York, NY, USA, 2014. ACM.

[SAK07] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[SAPM14] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 564–575, New York, NY, USA, 2014. ACM.

[SBP+18] Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-Right Consistency: reconciling availability and safety. Technical Report 9145, Inria Paris; Sorbonne Universités; Tech. U. Kaiserslautern; U. Nova de Lisboa; U. Catholique de Louvain, Paris, France, January 2018.

[SK09] Marc Shapiro and Bettina Kemme. Eventual consistency. *Encyclopedia of Database Systems*, pages 1071–1072, 2009.

[SKJ15] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[SP16] Manuel Serrano and Vincent Prunet. A glimpse of hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 180–192, New York, NY, USA, 2016. ACM.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[SPH15] Gianluca Stivan, Andrea Peruffo, and Philipp 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.

[SRI+13] Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[SSR17] Rodrigo CM Santos, Francisco Sant'Anna, and Noemi Rodriguez. A gals approach for programming distributed interactive multimedia applications. In *Anais*

do XXIII Simpósio Brasileiro de Sistemas Multimidia e Web, 2017.

[SW18] Kazuhiro Shibanai and Takuo Watanabe. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2018, pages 13–22, New York, NY, USA, 2018. ACM.

[SYS07] Haifeng Shen, Zhonghua Yang, and Chengzheng Sun. Collaborative web computing: From desktops to webtops. *IEEE Distributed Systems Online*, 8(4):3–3, 2007.

[TCPL11] Giovanni Toffetti, Sara Comai, Juan Carlos Preciado, and Marino Linaje. State-of-the art and trends in thesystematic development of rich internet applications. *Journal of Web Engineering*, 10(1):070–086, 2011.

[TG11] Eli Tilevich and Sriram Gopal. Expressive and extensible parameter passing for distributed object systems. *ACM Transactions on Software Engineering and Methodology*, 21(1):3:1–3:26, December 2011.

[VdVDKMDM17] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. Tackling the awkward squad for reactive programming: The actor-reactor model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2017, pages 27–33, 2017.

[VdVMDKDM17] Sam Van den Vonder, Florian Myter, Joeri De Koster, and Wolfgang De Meuter. Enriching the internet by acting and reacting. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 24:1–24:6, 2017.

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Proceedings of the 10th Computing Colombian Conference*, pages 583–590, Sep. 2015.

[VTR+14] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP'14, pages 360–384, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[WHSAT10] Adam Welc, Richard L. Hudson, Tatiana Shpeisman, and Ali-Reza 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.

[Win99] Dave Winer. Xml-rpc specification. `http://xmlrpc.scripting.com/spec.html`, 1999. Accessed: 2019-09-27.

[WKS18] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129:1–129:30, October 2018.

[WS17] Takuo Watanabe and Kensuke Sawada. Towards reflection in an frp language for small-scale embedded systems. In *Proceedings of the Companion to the 1st International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 10:1–10:6, 2017.

[WY14] Takuo Watanabe and Akinori Yonezawa. *Reflection in an Object-Oriented Concurrent Language*, pages 44–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[ZN16] Nosheen Zaza and Nathaniel Nystrom. Data-centric consistency policies: A programming model for distributed applications with tunable consistency. In *Proceedings of the 1st Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, pages 3:1–3:4, New York, NY, USA, 2016. ACM.