Handling Partial Failures in Distributed Reactive Programming

Florian Myter*
Vrije Universiteit Brussel
Elsene, Belgium
fmyter@vub.be

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

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

Abstract

Distributed reactive programming enables programmers to reuse the abstractions provided by reactive programming to elegantly implement distributed systems. However, distributed reactive approaches have thus far neglected to address an inherent property of distributed systems: partial failures. This forces programmers to either disregard failures and write poor distributed code or try to detect failures manually (e.g. through time-outs and heartbeats). Moreover, this prohibits distributed reactive runtimes to garbage collect remote references to failed parts of the reactive network. In this paper we present a first attempt at failure handling for distributed reactive applications. To this end we introduce the novel concept of leased signals which allow both programmer and runtime to react to partial failures in distributed reactive applications. We implement leased signals in a distributed reactive TypeScript framework for the development of microservice applications.

CCS Concepts • **Computing methodologies → Distributed programming languages**;

Keywords distributed reactive programming, failure handling, leasing, reactive microservices

ACM Reference Format:
Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2017. Handling Partial Failures in Distributed Reactive Programming. In *Proceedings of 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3141858.3141859

*Funded by Innoviris (the Brussels Institute for Research and Innovation) through the Doctiris program (grant number 15-doct-07)

REBLS'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5515-5/17/10…$15.00
https://doi.org/10.1145/3141858.3141859

1 Introduction

Reactive programming [1] has shown its merits as a paradigm to elegantly implement event-driven applications. In a nutshell, it reifies time-varying values (e.g. mouse clicks, system time, etc.) as first-class citizens called *signals* . Programmers are able to declaratively specify dependencies between these signals, which form a *dependency graph.* The underlying reactive runtime ensures that changes to a signal are propagated through the dependency graph, thereby updating the application's state.

Distributed reactive programming (DRP) [6, 10, 13] applies the concepts of reactive programming to distributed systems. Using DRP programmers are able to declaratively combine both local and remote signals. This entails that part of the dependency graph might reside on physically distributed machines. This strain of reactive programming allows developers to specify data dependencies between distributed components. The underlying DRP framework or language ensures that all components remain up to date as data enters the distributed system.

However, current DRP approaches fail to address an essential property of distributed systems: *partial failures.* Given that these systems are comprised of multiple physically distributed machines, these partial failures can happen for a number of reasons (e.g. network partition, the failing of a machine, etc.). This lack of resilience by current DRP approaches negatively impacts the development of distributed reactive applications in two ways:

- Programmers are only able to write code which reacts to values being propagated through the distributed dependency graph. Current approaches fail to provide programmers with abstractions to write code which reacts to the *absence* of values. In other words, programmers are unable to specify the system's behaviour in case of network partitions and partial failures.
- The failure of part of a distributed reactive application invalidates paths in the underlying distributed dependency graph. As remote signals fail, their descendants no longer react to changes. For the sake of memory efficiency, these "dead" parts of the dependency graph should be garbage collected. However, determining which parts of the dependency graph are

dead requires the underlying framework or language to embrace partial failures.

In this paper we introduce the concept of *leased* signals. These signals allow both the programmer as well as the underlying framework to handle failures in distributed reactive applications. Concretely, leased signals solve the aforementioned problems as follows. First, application-level code is able to react to the expiration of a lease on a signal. This allows programmers to write code which reacts to the failure of a remote signal. Second, our framework uses the leasing information of signals to determine when parts of the distributed dependency graph can be garbage collected. We integrate this novel reactive concept in an extension of the Spiders.js web framework [11] which allows programmers to implement *reactive microservices* [1].

2 Motivating Example and Problem Statement

Throughout this paper we use a motivating example. The example application stems from our cooperation with Emixis [2], a company developing fleet management software, in the context of a research project [3]. In a nutshell, the application consists of a fleet of vehicles equipped with tracking beacons which periodically upload packets of sensory input data (e.g. current position, speed, etc.) to a server-side application implemented by microservices. Emixis' clients, which own the fleet of vehicles, are able to visualise this data through an online dashboard. Concretely, the server's main tasks are the following:

Parsing Data packets are serialised and compressed by fleet members for efficiency reasons, the server must therefore deserialise these packets for further use. Moreover, data packets are stored in a database to avoid the loss of data.

Geolocalisation The location data contained in each data packet is reverse geocoded to a concrete address. These addresses are used to update a map which provides an overview of the fleet's current location.

Driving The server calculates various driving parameters (e.g. eco-efficiency, speed limitation violations, etc.) based on sensory data and the result of the reverse geocoding. These calculations are made available to the client through the dashboard.

An overview of the application's architecture and how data flows through the various microservices is depicted in Figure 1(A). Data enters the system as members of the connected fleet send their packets of sensory input data to the *Parsing* microservice. This service deserialises the data, persists it and forwards it to the *Geo* and *Driving* microservices. The *Geo* service reverse geocodes the position information contained in the data sent by the *Parsing* service and sends addresses to the *Driving* and *Dashboard* services. The *Driving* service combines the data from the *Parsing* service with addresses from the *Geo* service to send driving-related information to the *Dashboard* service. The *Dashboard* service pinpoints the location of each fleet member on a map using the addresses sent by the *Geo* service and provides an overview of the member's driving statistics sent by the *Driving* service. This map of fleet members and their driving statistics are provided to clients which visualise this data in their browsers.

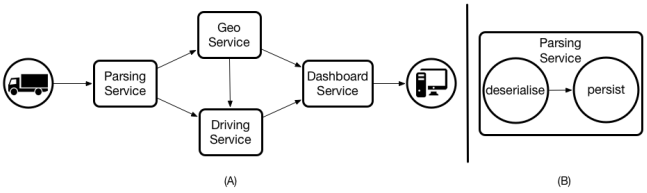

Figure 1. (A) Overview of data flowing through the fleet management application. (B) Overview of data flowing through the parsing microservice.

Implementing this example using DRP would go as follows. The fleet members and services all export signals which can be imported by other services or clients to compose them into new signals. The arrows in Figure 1(A) symbolise the dependencies between these exported signals. For example, the *Dashboard* service combines signals exported by the *Geo* and *Driving* service to form its own signal which is exported to the clients. In a nutshell, DRP would allow programmers to easily implement the flow of data across microservices.

Additionally, each individual microservice could be programmed using (non-distributed) reactive programming as well. In other words, the internal flow of data in each microservice could be modelled using the same constructs employed to model the flow of data across microservices. Figure 1(B) provides an overview of data flowing through the *Parser* microservice. The data flow is represented by a dependency graph, where each node in the graph represents the application of a lifted function on signals and edges represent dependencies between the signals resulting from these applications. Concretely, the parsing service is implemented by applying two lifted functions. First a *deserialise* function is applied to the incoming signal provided by a fleet member. Each time the fleet member's sensory input data changes this function deserialises this input data which is received as a string. The *persist* function depends on the signal resulting from applying *deserialise* to the incoming data signal. As data is changed by the fleet member and the *deserialise* function is reevaluated the *persist* function stores (or updates) the data in the database.

[1] Our implementation is available at https://github.com/myter/Spiders.js/tree/ReactiveMicroservices

[2] http://www.emixis.com/

[3] Doctiris project *"A tierless reactive cloud-oriented middleware to enhance the quality of internet-based fleet management applications"*, funded by Innoviris

However, the lack of constructs to deal with partial failures in current distributed reactive approaches hinders the development of our fleet management application. Concretely, the following two issues arise when implementing our example in a distributed reactive fashion:

Handling partial failures as a programmer Current approaches do not provide constructs which allow programmers to write code reacting to the *failure* of a remote signal. For example, a requirement of our fleet example might be that fleet members which have lost connection with the server should be greyed-out on the map generated by the *Dashboard* service. To fulfil this requirement, the application's programmer is forced to detect these failures manually (e.g. by resorting to time-outs or heartbeat functionality). This manual approach unnecessarily burdens the programmer and breaks the reactive paradigm which explicitly does away with callback-based programming.

Handling partial failures as a reactive runtime Applying a lifted function creates a dependency between the signals passed as arguments and the function's return signal. In our case this entails that a graph such as the one depicted in Figure 1(B) is created *per* fleet member (given that each fleet member is implemented as a remote signal). Consequently, the failure of a fleet member renders an entire graph invalid given that it no longer propagates values. However, the runtime is unable to garbage collect this graph given that it is unable to detect the failure of remote signals. Although this might seem acceptable for our small example, graphs in real-world applications would incur a much bigger memory overhead. Moreover, each service in our application has parts of its dependency graph which needs to be garbage collected.

Partial failures are an essential property of any distributed system. In the case of a microservice architecture these failures can arise, for example, from services crashing or clients losing connection with the server. Our simple example showcases the need for failure handling mechanisms, both for programmers and runtimes, in distributed reactive programming.

3 Reactive Microservices in Spiders.js

Before explaining how we tackle partial failures in distributed reactive programming we detail a subset of our fleet management application's implementation in order to familiarise the reader with Spiders.js. In a nutshell, Spiders.js is a framework implemented in Typescript [4] which provides actor-based concurrency constructs. Actors in Spiders.js are distribution-ready: they are able to communicate across client/server boundaries through built-in asynchronous messaging and promises.

[4] A typed superset of JavaScript

```
class FleetData extends Signal{
 memberId : string
 currentLat : number
 currentLong : number
 currentspeed : number

 constructor(id){
  this.memberId = id
 }

 @mutator
 actualise(lat,long,speed){
  this.currentLat = lat
  this.currentLong = long
  this.currentspeed = speed
 }
}
```

Listing 1. Defining the FleetData signal

```
class FleetMember extends MicroServiceClient{
 dataSignal

 init(){
  let myId = beacon.id
  this.dataSignal = this.newSignal(FleetData,myId)
  let serialise = this.lift((fleetData) => {
   return compress(fleetData)
  })
  let compressed = serialise(this.dataSignal)
  let topic = this.newTopic("FleetData")
  this.publish(topic,compressed)
  update()
 }

 update(){
  let newLat = beacon.latitude
  let newLong = beacon.longitude
  let newSpeed = beacon.speed
  this.dataSignal.actualise(newLat,newLong,newSpeed)
  setTimeout(() => {
   this.update()
  },3000)
 }
}
```

Listing 2. Implementation of the fleet members

```
class ParsingService extends MicroService{
 init(){
  let inTopic = this.newTopic("FleetData")
  let outTopic = this.newTopic("ParsedData")
  let deserialise = this.lift((fleetDataString) => {
     return decompress(fleetDataString)
  })
  let persist = this.lift((fleetData) => {
   db.put(fleetData.memberId,fleetData)
  })
  this.subscribe(inTopic).each((dataSignal) => {
   let deserialised = deserialise(dataSignal)
   persist(deserialised)
   this.publish(outTopic,deserialised)
  })
 }
}
```

Listing 3. Implementation of the Parsing microservice

We extend this framework with constructs to implement reactive microservices and their clients. These constructs extend Spiders.js' actors with a topic-based publish/subscribe mechanism which allows them to share signals. To introduce these extensions we focus on the implementation of the fleet members (i.e. the vehicles equipped with beacons uploading data to the application) and the *Parsing* service.

Listing 1 contains the definition of the *FleetData* signal, which is the signal published by each member of the fleet. In Spiders.js one defines a signal by extending the *Signal* class. This definition closely resembles that of an object class: it defines instance variables, a constructor and methods. Additionally, programmers can annotate the methods of a signal with the *@mutator* annotation. By using this annotation, programmers inform the Spiders.js runtime that an invocation of this method mutates the signal's state. Subsequently, the Spiders.js runtime propagates this change to all dependants of the signal. In our example all *FleetData* signals contain instance variables for each piece of sensory input data (i.e. latitude and longitude of the fleet member and the speed at which it is travelling) as well as a mutating method to update these variables.

The code executed by each fleet member is given in Listing 2. The task of a fleet member is to publish a *FleetData* signal and change its value at regular intervals. To this end it employs two methods: *init* and *update*. The former is called by the Spiders.js runtime at the start of the application. It creates a *FleetData* signal by invoking the built-in *newSignal* method which takes a signal class and constructor arguments as parameters. We say that the fleet member *owns* the created signal. The resulting signal is used as argument to call *serialise*, which is a lifted function. Each time the signal changes, the base function (i.e. the function passed as parameter to *lift*) is re-invoked with the signal's latest value. Moreover, calling *serialise* returns a new signal whose value represents the last computed return value of the base function. In our example the base function returns a compression of the provided argument (see Line 8). Therefore, each time *dataSignal* changes the compression is recalculated and *compressed* is updated. Finally, the *init* method creates a topic and uses it to publish the compressed signal returned by *serialise*. The *update* method changes *dataSignal*'s value (i.e. through the signal's mutating *actualise* method) and recursively calls itself every three seconds.

Listing 3 provides the definition of the *Parsing* microservice which deserialises fleet data, publishes this deserialised data and stores it in a database. To acquire the fleet data,the service subscribes to the topic used by the fleet members (see Line 11). For each compressed fleet data signal acquired the service invokes the *deserialise* and *persist* lifted functions. The semantics of lifted functions are the same regardless of whether the signal is local or remote (i.e. published by another service or client). In our example, as a particular fleet member changes its data the *deserialise* and *persist* functions are re-invoked automatically. Lastly, the service publishes the *deserialised* signal which is used by other services.

The code shown in this section serves as an introduction to Spiders.js and the reactive microservice extension. It lacks failure handling functionality. More precisely, it lacks the constructs needed to specify how the application should react to failing fleet members.

```
1  @lease(4000)
2  class FleetData extends Signal{
3   //Definition of the FleetData signal omitted for brevity
4  }
```

Listing 4. Leasing the FleetData signal

```
1  class ParsingService extends MicroService{
2   init(){
3    let inTopic = this.newTopic("FleetData")
4    let deserialise = this.lift((fleetDataString) => {
5       return decompress(fleetDataString)
6    })
7    let fail = this.liftFailure((fleetData) => {
8     db.delete(fleetData.memberId)
9    })
10   this.subscribe(inTopic).each((dataSignal) => {
11    let deserialised = deserialise(dataSignal)
12    fail(deserialised)
13   })
14  }
15 }
```

Listing 5. Reacting to lease expirations for fleet data

4 Leased Signals

Our solution to partial failures in distributed reactive programs is centred around a single concept: leased signals. A leased signal extends the concept of a remote signal, introduced by related work such as [13] and [5], with a semantic agreement between the signal's owner (i.e. the service or client invoking *newSignal*) and its subscribers. This agreement dictates the time frame for which the signal is valid. Concretely, in Spiders.js a lease allows programmers to specify how often a signal changes value (e.g. a lease of 1 second assumes that the leased signal changes *at least* every second). If a signal fails to adhere to its leasing restriction we *assume* the owner of the signal to have failed (e.g. by crashing or losing connection with the network). One must note that a lease expiring does not strictly imply the failure of the signal's owner or vice versa.

Listing 4 showcases how we apply leased signals to our fleet management application. By annotating the *FleetData* signal class with the *@lease* annotation we inform the Spiders.js runtime that signals produced by fleet members should change at least every four seconds (we allow for a second of delay between the fleet member changing the signal and dependants witnessing this change). The definition of *FleetData* (i.e. the methods and instances variables) remain identical to the version given by Listing 1 and are therefore omitted from Listing 4. The remainder of this section details how Spiders.js handles partial failures both at application and runtime-level.

4.1 Reacting to Partial Failures

In Spiders.js programmers are able to write code which reacts to the expiration of a lease. Moreover, this code adheres to the reactive paradigm meaning that these reactions can be arbitrarily composed through lifted functions. For example, consider Listing 5 which extends the implementation

of the *Parsing* microservice given in Section 3 (unchanged code has been omitted for the sake of brevity). The failure-specific additions to the code are located on Line 7 and Line 12. First, the programmer creates a lifted failure function using *liftFailure*. In contrast to lifted functions created using *lift*, lifted failure functions are not re-executed whenever their argument signals change. Rather, these functions are executed whenever the lease on an argument signal expires. In our example the lifted failure function removes the failed member's fleet data from the database.

On Line 12 the lifted failure function is applied to the *deserialised* signal, which is the result of applying the *deserialise* lifted function to *dataSignal*. This showcases how lease inheritance works in Spiders.js. The programmer only explicitly leases *dataSignal*, given that it is an instance of *FleetData* (see Listing 4). However, *deserialised* is dependent on *dataSignal* and therefore inherits *dataSignal*'s lease. In case a signal depends on multiple leased signals the lease with the smallest value is taken to be the inherited lease.

Lease inheritance spans across services and clients. For example, a requirement of our fleet management application is to grey-out vehicles which have failed to recently update their data. In other words, the *Dashboard* microservice should detect whenever a fleet member fails. Listing 6 shows how the *Dashboard* service achieves this. For the sake of brevity we assume that the *Geo* and *Driving* services publish signals under the *"GeoData"* and *"DrivingData"* topics. Moreover, we assume the existence of a *ui* signal which the *Dashboard* publishes. To react to failing fleet members, the *Dashboard* service invokes a lifted failure function over the *geoData* signal (see Line 17). This signal depends on the *deserialised* signal published by the *Parsing* service which in turn depends on the signal published by fleet members. Spiders.js ensures that the original lease (i.e. the lease defined on the *FleetData* definition) is inherited by all dependants, local or remote. Therefore, whenever a lease expires on a signal published by a fleet member this expiration propagates throughout the distributed dependency graph. In our example this propagation eventually triggers the execution of the lifted failure function *greyOut* which is applied to *geoData*.

liftFailure adheres to the same semantics as *lift*: the return value of applying a lifted failure function is a signal. This allows programmers to declaratively specify failure-handling code on combinations of leases expiring within the reactive paradigm. Concretely, *liftFailure* creates a special dependency graph (i.e. a failure graph) through which events only propagate once: upon failure of its source nodes. In our example the source node for the failure graph is *deserialised* on which the signal *fail* depends.

4.2 Garbage Collection as a Reaction to Partial Failures

Assume a vehicle v which has published a *FleetData* signal and has regularly been updating it. Each microservice in our application has a dependency graph (e.g. Figure 1(B)) which handles changes to this signal. Now assume that v loses network connectivity, prompting leases to expire throughout the system. Concretely this entails that all signals which depend directly or indirectly on v's signal are rendered unresponsive. Therefore, it is Spiders.js' task to initiate (distributed) garbage collection cycles whenever a lease expires on a signal. Determining which signals to garbage collect is straightforward, given that the dependency graph explicitly contains this information. For example, whenever a *FleetData* signal expires the depending *deserialise* and *persist* signals created by the *Parsing* service are to be garbage collected. All subsequent updates of the expired signal are ignored, given that it is assumed to have failed.

```
1  class DashboardService extends MicroService{
2   init(){
3    let geoTopic = this.newTopic("GeoData")
4    let drivingTopic = this.newTopic("DrivingData")
5    let dashTopic = this.newTopic("DashData")
6    let drawMap = this.lift((geoData) => {
7     ui.update(geoData)
8    })
9    let drawDriving = this.lift((drivingData) => {
10    ui.update(drivingData)
11   })
12   let greyOut = this.liftFailure((geoData) => {
13    ui.grey(geoData)
14   })
15   this.subscribe(geoTopic).each((geoData) => {
16    drawMap(geoData)
17    greyOut(geoData)
18   })
19   this.subscribe(drivingTopic).each((drivingData) => {
20     drawDriving(drivingData)
21   })
22   this.publish(dashTopic,ui)
23  }
24 }
```

Listing 6. Implementation of the Dashboard service

For our simple use case, garbage collecting all dependants of an expired signal is acceptable. For each published *FleetData* signal a *separate* dependency graph is created on each microservice. Therefore, upon failure of said signal all of its dependants can be garbage collected. However, this approach is too coarse-grained to suit all distributed reactive applications. For example, imagine an application which combines values from different source signals. In contrast to our example, the dependency graphs within the microservices combine values depending on different source signals. The failure of a single source signal should not trigger the garbage collection of the entire distributed dependency graph.

In analogy with strong and weak variable references, we introduce *strong* and *weak* signals. Only weak signals are garbage collected whenever their leases expire. Programmers have explicit control over a signal's strong or weakness through dedicated constructs. We divide these constructs based on when they are applied to a given signal:

At definition time Spiders.js provides programmers with two annotations, *@weak* and *@strong*, to annotate signal classes. This informs the runtime that all signals

instantiated from this class should be weak or strong respectively.

At lifting time Using *liftWeak* and *liftStrong* programmers are able to force the resulting signal of a lifted (non-failure) function to be weak or strong. However, *liftStrong* can only be used if all signal arguments to the lifted function application are strong as well. If one of the signal arguments would be weak it could be garbage collected at some point in the program. This would leave the signal resulting from the application with a missing dependency. Spiders.js ensures that this cannot happen by throwing a runtime exception if this rule is violated by the programmer.

At transmission time Both *publish* as well as *subscribe* have weak and strong variants (e.g. *publishWeak*, *subscribeStrong*). *publishWeak* ensures that the signal it publishes is transmitted to all subscribers as a weak signal. *subscribeWeak* turns all subscribed signals into weak signals. As is the case for *liftStrong*, Spiders.js ensures that programmers cannot create strong signals which depend on weak signals. Therefore, applying *publishStrong* to a weak signal throws a runtime exception. Similarly, *subscribeStrong* ensures that its callback is never invoked with a weak signal.

5 Related Work

We do not claim to contribute to the fields of leasing [7] [3] or microservices [8]. We therefore restrict our discussion of related work to the field of distributed reactive programming.

AmbientTalk/R [5] is a reactive extension to the AmbientTalk language. Much like Spiders.js it allows for publish/-subscribe transmission of signals. Moreover, it allows for temporal disconnections of peers by buffering changes to signals until reconnection. However, it does not allow programmers to write application-level code which reacts to partial failures.

Distributed REScala [6] assumes a distribution model where partial failures are absent. Although Dream [10] provides a form of fault-tolerance by caching values of remote signals, it neither provides programmers with ways to react to partial failures nor does its runtime garbage collect remote references upon failure. In [12] Reynders et al. present a multi-tier language which replicates signals amongst clients and server in the context of web applications. As explicitly mentioned in [12] the language neglects error handling.

A number of GALS (globally asynchronous locally synchronous) systems have been specifically tailored towards distribution [2, 4, 9]. To the best of our knowledge none of these systems address the issues tackled by leased signals.

6 Future Work

We envision two areas of future work concerning leased signals. First, we aim to increase the programmer's control over our leasing mechanism. To do so we plan on introducing constructs which allow programmers to override the inheritance of leases or to manually renew leases. This would allow programmers to manually specify the lease of a derived signal (e.g. to take varying network latency into account).

Second, our approach only allows programmers to specify a lower bound on the rate at which a signal produces values. Future work will focus on allowing programmers to specify an upper bound on this rate. Our framework could then use this upper and lower bound information to provide real-time guarantees about the distributed reactive system.

7 Conclusion

Current distributed reactive approaches fail to address an issue inherent to distributed systems: partial failures. This forces programmers of distributed reactive systems to manually deal with partial failures. Moreover, the lack of failure handling disallows the reactive runtime to garbage collect dependencies to failed remote signals.

In this paper we present a novel reactive construct tailored towards handling partial failures: leased signals. This novel construct tackles partial failures in distributed systems on two levels. First, programmers are able to write reactive code which is triggered by the failure of a remote signal. Second, our reactive runtime uses this leasing information to garbage collect parts of the (distributed) dependency graph involving failed signals.

We implement leased signals in an extension of the Spiders.js framework tailored towards reactive microservices. Concretely, programmers specify leasing information on signals through annotations and use special lifting operators to react to the expiration of leases. Moreover, programmers are able to tweak how garbage collection happens through the use of strong and weak signals.

References

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 52:1–52:34 pages.

[2] Albert Benveniste, Paul Le Guernic, and Pascal Aubry. 1998. Compositionality in dataflow synchronous languages: specification and code generation. *Lecture notes in computer science* (1998), 61–80.

[3] Elisa Gonzalez Boix, Tom Van Cutsem, Jorge Vallejos, Wolfgang De Meuter, and Theo DâĂŹHondt. 2009. A leasing model to deal with partial failures in mobile ad hoc networks. In *International Conference on Objects, Components, Models and Patterns*. Springer, 231–251.

[4] Benoit Caillaud, Paul Caspi, Alain Girault, and Claude Jard. 1994. *Distributing Automata for Asynchronous Networks of Processors*. Research Report RR-2341. INRIA. https://hal.inria.fr/inria-00074336

[5] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer-Verlag, Berlin, Heidelberg, 41–60.

[6] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. 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)*. ACM, New York, NY, USA, 361–376.

[7] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. ACM, New York, NY, USA, 202–210.

[8] J. Lewis and M. Fowler. 2014. Microservices. Common characteristics of architectural style. (2014). https://martinfowler.com/articles/microservices.html

[9] Olivier Maffeis and Paul Le Guernic. 1994. Distributed implementation of Signal: Scheduling & graph clustering. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 547–566.

[10] Alessandro Margara and Guido Salvaneschi. 2014. 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)*. ACM, New York, NY, USA, 142–153.

[11] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2016. Many Spiders Make a Better Web: A Unified Web-based Actor Framework. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM, New York, NY, USA, 51–60.

[12] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. 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)*. ACM, New York, NY, USA, 55–68.

[13] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. 2013. Towards distributed reactive programming. In *International Conference on Coordination Languages and Models*. Springer, 226–235.