I Now Pronounce You Reactive and Consistent

Handling Distributed and Replicated State in Reactive Programming

Florian Myter *
Tim Coppieters †
Vrije Universiteit Brussel
Pleinlaan 2
Elsene, Belgium
fmyter / tcoppiet@vub.ac.be

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

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

Abstract

Developing modern collaborative applications burdens the programmer with local event handling (e.g. user interaction), remote event handling (e.g. updates from the server) and shared state (e.g. in order to allow operations while being disconnected). Several solutions have been developed at the programming language level in order to reduce the complexity of these aspects. On one hand, distributed reactive models (e.g. DREAM) tackle both local and remote event handling. On the other hand recent replicated consistency models (e.g. CRDT's and CloudTypes) hide the complexity of shared, replicated state. Both solutions only partially alleviate the complexity associated with developing collaborative applications. To the best of our knowledge, none or very little effort has been undertaken to provide a single unified model able to tackle both event handling and shared state. In this paper we argue the need for such a united model. To that end we present Direst, a domain specific language which enhances traditional reactive abstractions (i.e. signals) with replication and consistency features. Direst reduces the complexities of writing truly collaborative applications by providing a framework in which elegantly handling events and easily managing shared state are not mutually exclusive.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed Programming

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

† Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

REBLS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4644-3/16/11...$15.00
http://dx.doi.org/10.1145/3001929.3001930

Keywords reactive programming, distributed programming, data consistency, shared state, eventual consistency

1. Introduction

From simple chat applications over document or source editing to virtual whiteboards, interactive collaborative applications are omnipresent. When one adds modern requirements such as scalability and offline availability (i.e. allowing local operations while the client is disconnected), writing such application becomes a particularly tedious task. We distil two main challenges presented to programmers when writing such applications:

Event handling Given the distributed and interactive nature of these applications programmers need to spend considerable resources handling various events. These events can either be internally (e.g. UI events) or externally (e.g. html requests) produced. Traditionally one employs callbacks and the observer pattern to deal with these events. However, these techniques negatively impact the maintainability and readability of code (Salvaneschi et al. 2014).
The reactive programming paradigm (Bainomugisha et al. 2013) is a solution to the problems that emerge from the use of callbacks. At the heart of this solution lie three concepts: First, events and time-varying values (e.g. the system time) are represented as first-class citizens called *signals*. Second, programmers are able to declaratively specify dependencies between these signals through the use of *lifted* functions. Third, when a signal changes it is the language runtime which makes sure that all dependant signals are updated accordingly. These three concepts allow programmers to explicitly write down, and reason about relations between events.
The reactive paradigm has traditionally been applied to user interfaces (Czaplicki and Chong 2013) and client-side logic (Meyerovich et al. 2009). However, recent work has focused on applying these solutions in the context of distributed systems. In a nutshell, these approaches enable programmers to specify dependencies between signals across

distributed clients (Margara and Salvaneschi 2014) or across tiers (Reynders et al. 2014).

Replication Collaborative applications naturally involve a notion of shared state. This includes distributing the state to the clients, coordinating updates and maintaining a consistency model such that replicas converge. In a more traditional approach, called pessimistic replication, a single globally consistent view of the replicate state is maintained. While this model is easy to comprehend and work with for the programmer, this is not always scalable and does not allow disconnected operations. Models that do allow concurrent updates on the shared state, called optimistic replication, provide weaker guarantees over the state and introduce an enormous overhead for the programmer. Namely, when a local update happens it is up to the programmer to propagate this update to all replicas and to detect and resolve or prevent possible conflicts (Saito and Shapiro 2005).
A significant amount of programming models, frameworks and libraries (Burckhardt et al. 2012; Shapiro et al. 2011; Meiklejohn and Van Roy 2015; Lorenz and Rosenan 2014; Coppieters et al. 2016) aim to reduce the burden of maintaining replicated state for the programmer. These *replicated programming* models provide different guarantees over the replicated state, but they all hide the complexity of maintaining these guarantees by providing a simple API to the programmer.

We advocate that the challenges underlying interactive and collaborative applications are greater than the sum of their solutions. In order to easily handle distributed and replicated state in collaborative applications one needs a unified model which embraces the solutions provided by both (distributed) reactive programming as well as replication-based programming.
Distributed reactive programming models lack the abstractions needed to represent mutable reactive values replicated amongst distributed clients. Conversely, replication models lack the abstractions provided by reactive programming to elegantly combine their state changes with event-handling logic.
In this paper we introduce Direst (**Di**stributed **Re**active **St**ate)[1], a domain specific language built on top of AmbientTalk (Cutsem et al. 2014) which marries the worlds of reactive and replicated programming. Direst allows programmers to easily write distributed reactive applications which share state among clients. In order to achieve this, it introduces the novel concept of *replicated* signals. The novelty of these signals lies in the fact that they provide built-in eventual consistency without programmers needing to manually synchronise state. Concretely, Direst provides native constructs to replicate signals over multiple clients. Each clients is able to mutate a replica of a signal at any time

[1] The implementation can be found at `https://github.com/myter/Direst`

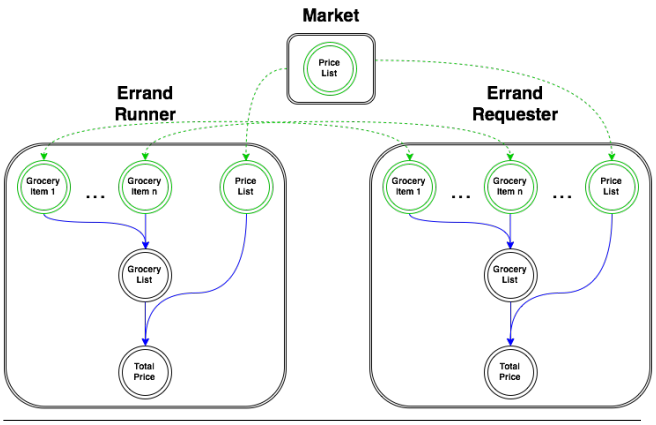

Figure 1. Architecture of the grocery application

(i.e. even during disconnections). It is Direst which ensures that all replicas of a signal are kept consistent across clients. This prototypical language showcases how replication as a first-class reactive programming construct can ease the implementation of interactive and collaborative applications.

2. Motivating Example and Problem Statement

As an example of an application that could benefit from our approach consider a peer-to-peer grocery list application. In this application a number of clients can collaboratively edit a common grocery list. These clients consist of a single *errand runner* which will physically go to a local store to buy the groceries and multiple *errand requesters* which simply add groceries to the list. The following requirements summarise the application's main functionality:

- The *errand runner* is responsible for creating the list. However, all clients are able to add groceries to the list or increase the amount of exemplars to be bought of a given item. Changes made to the grocery list should eventually be visible to all clients.
- Clients should be able to update the grocery list even when they temporarily loose internet connection. Changes to the grocery list made by an offline client should automatically be resolved upon reconnection.
- The grocery list also figures a total price of all groceries. The local supermarket at which the groceries will be bought provides the clients with a list prices for all its items. These prices can be changed solely by the market itself.

The implementation of this example application poses a number of interesting challenges with regard to reactivity and data replication. Figure 1 provides an overview of our application's architecture. Each square represents a distributed client (only one *errand requester* is depicted for the sake of brevity). Within each square the circles represent

reactive values (i.e. values that may change as the result of user interaction with the application). An arrow between two circles indicates a dependency between the reactive values which they represent. We distinguish two types of challenges regarding the implementation of this application: reactive challenges which affect the parts of the architecture highlighted in blue and replication challenges indicated in green.

2.1 Reactive Challenges

Each errand-related client (i.e. the *errand runner* or *errand requesters*) comprises four reactive values or signals: the *grocery list* itself, the amount of exemplars to be bought for each *grocery item*, the *price list* which dictates the price of each item and the *total price* of the list. The market client comprises a single signal: a *price list* of the available items. The dependencies between all these signals is shown in the form of a graph in Figure 1. Each *grocery list* depends on all *grocery item* signals. The list changes as either items are added or the amount of items to be bought for an exemplar changes. Furthermore, each *total price* signal depends on both its *grocery list* as well as the *price list* as a change in either signals should trigger a change in the overall price of the errand.

Traditional distributed reactive programming would provide us with the means to declaratively specify the dependencies between the reactive values in our application (i.e. the blue arrows in Figure 1). Furthermore, the language runtime would make sure that changes to a signal would trickle down the dependency graph. For example, the addition of a new grocery item would trigger a change in the grocery list itself which would eventually change the total price of the list.

However, the distributed reactive paradigm fails to provide us with the means to share and maintain mutable reactive state (i.e. the green arrows in Figure 1). We discern two instances of shared state. First, the *price list* signal which can only be modified by the market. Given that clients are able to add groceries to the list while being disconnected (and therefore the *total price* signal must be updated offline as well), all clients need a replica of the *price list*. Second, *grocery item* signals are shared amongst and can by mutated by the *errand runner* and all *errand requesters*. This entails that a change to or an addition of a *grocery item* signal should trigger a propagation of change for all clients.

> Traditional distributed reactive programming lacks the notion of shared signals. Therefore, in order to implement our application in a reactive way, programmers would need to manually handle all replication and synchronisation of shared signals.

2.2 Replication Challenges

From a replicated programming point of view we distinguish two issues. First, the collection of grocery items can be mutated by all clients which can lead to multiple versions of the collection across clients. Second, although the price list signal is replicated as well we must ensure that only the market is able to mutate its state. Replicated programming would allow us to alleviate these issues as follows. The *errand runner* contains the master version (i.e. the version considered consistent) of the grocery list. All changes made to the list by *errand requesters* must be merged with its version of the list. It is the underlying replication model which makes sure that all views held by other clients will be consistent with the master version. Furthermore, some replication models (Coppieters et al. 2014) would allow us to specify *access control policies* which would enable us to safely replicate the market's price signal. The market would retain the sole right to mutate its state while the errand runner and errand requesters would only be able to read its state.

Although this approach allows us to easily manage shared state, programmers need to manually manage the dependencies between various parts of the application as well as the propagation of changes between those parts. For example, when a client receives an update to the grocery list the programmer needs to make sure that both the list as well as the total price are updated correctly.

> Current replication models and languages lack the high-level abstractions to elegantly write event-driven applications. As a result, programmers are forced to resort to archaic observer patterns and callbacks which quickly lead to issues such as the callback hell (Finne et al. 1999).

3. Programming in Direst

Direst embraces both the reactive as well as the replication challenges exhibited by collaborative applications. To demonstrate this we detail the implementation of our grocery application using our approach. Given that Direst is implemented atop AmbientTalk we first briefly introduce a number of key concepts of the AmbienTalk language in order to make our implementation understandable.

AmbientTalk is an actor-based language where each actor encompasses a heap of objects. Method invocation can either happen synchronously between two objects owned by the same actor, or asynchronously between objects owned by different actors. Objects are able to discover objects owned by other actors through a built-in tag-based publish/subscribe discovery mechanism. By default all objects passed between actors (e.g. an object is provided as an argument to an asynchronous message) are passed by *far reference* (i.e. a reference referring to an object residing in a different actor). However, an exception is made for *isolates* which are objects that are passed by copy rather than by far reference.

We have split the implementation of the grocery application in three components: the *market*, the *errand runner* and the *errand requesters*. Each component is implemented as an actor which can run in complete isolation. Distributing this application is therefore achieved by running each actor on a separate device (e.g. a smartphone).

```
1  def market := actor:{
2   deftype Market;
3   def selfType(){
4    Market
5   };
6   def priceList := signal:{
7    def getPrice(item){
8     //...
9    };
10   def newPrice(item,price){
11    //...
12   };
13  };
14  publish: priceList as: Market mutableIf:{|mutator|
15    mutator.selfType() == Market
16  };
17  def updatePrice(item,newPrice){
18   update: priceList by:{|prices| prices.newPrice(item,newPrice)}
19  }
20 };
```

Listing 1. Implementation of the market

Market The implementation of the market, which provides clients with the latest price of its articles, is given by Listing 1. The market actor is responsible for creating the signal which represents the current price list. Creating this signal (line 6) is done using the $signal$: construct which takes a signal definition and returns the instantiated signal object. Such a signal definition represents the behaviour of the replicated signal (i.e. its fields, accessors and mutators) and can only be accessed through a dedicated updater. This construct, $update$: by :, takes a signal as argument and a function which will be invoked with the signal's behaviour. For example, the *updatePrice* function (line 17) updates the price of a given item by using the *priceList's* mutators.
The market actor is also responsible for publishing the price signal. This is done through the $publish$: as : $mutableIf$: construct (line 14) which accepts a signal, a type tag and a function as arguments. The signal is published under the given type tag which allows other actors to acquire a replica of the signal. The function enables programmers to specify which actors have the right to mutate the signal's state. In our case we specify that an actor can only mutate the price list if it is the market actor (i.e. it has the *Market* type).

Errand runner The errand runner's implementation is given in Listing 2. The errand runner creates a new signal representing the master version of the grocery list (line 6) and publishes it (line 22). In contrast tot the market it does not specify an access policy, which entails that any actor which obtained a replica will be able to mutate it. Direst ensures that changing the state of a replica (or the original signal) will trigger an update for all other replicas. In our example this entails that a change to the grocery list by one client will automatically change the grocery list of all other clients.
Besides creating the replicated grocery list, the errand runner also acquires a replica of the *prices* signal provided by the market (line 23). This is achieved with the $whenSignal$: $discovered$: primitive, which takes a type tag and a function as arguments. This primitive will invoke the function with a replica of the *price signal* published by

```
1  def runner := actor:{
2   deftype Runner;
3   def selfType(){
4    Runner
5   };
6   def groceries := signal:{
7    def items := [];
8    def newItem(item){
9     //...
10   };
11  };
12  def priceCalculator := lift({|groceries,prices|
13   def totalPrice := 0;
14   groceries.items.each:{|item|
15    totalPrice += prices.getPrice(item)
16   };
17   totalPrice
18  });
19  def updateGui  := lift({|groceryList,totalPrice|
20   //...
21  )};
22  publish: groceries as: Runner;
23  whenSignal: Market discovered:{|prices|
24   def totalPrice := priceCalculator(groceries,prices);
25   updateGui(groceries,totalPrice);
26  };
27  def addGrocery(grocery){
28   update: groceries by: {|list| list.newItem(grocery)};
29  }
30 };
```

Listing 2. Implementation of the errand runner

the market peer. It uses these prices and the grocery list to update the user interface shown to the application user. This is done through another construct provided by Direst: *lifted* functions. The $lift$: constructs takes a regular function as argument and lifts it in order to be applicable to signals. The application of such a lifted function returns a signal representing the computation's value. A lifted function can be applied to both regular arguments as well as signals, as soon as one of its signal arguments changes the closure will be re-executed with the new value of the changed signal and the last known value for all other signals as arguments. This re-execution will cause the return signal to be updated which in turn will update all of its dependants. In our example we have two lifted functions. First, the *priceCalculator* which depends on the *grocerySignal* and the *priceSignal* provided by the market. The returning signal represents the total price for the entire grocery list and is updates as soon as either a client adds an item to the grocery list or if the market changes its pricing. Second, the *updateGui* lifted function which updates the user interface to reflect the latest state of the grocery list an the total price of said list.

Errand requesters The code ran by all errand requesters is given in Listing 3, for the sake of brevity we do not show code in common with Listing 2 (i.e. *priceCalculator* and *updateGui*). Each errand requester acquires a replica of the *groceries* signal provided by the errand runner and a replica of the *prices* signal provided by the market. As was the case for the errand runner, two lifted functions assure the update of the user interface. Each client is able to add an element to the grocery list through the *addGrocery* method. As mentioned previously the addition of an element will trigger the underlying replication mechanism to make sure that all clients have the same grocery list and will therefore update the UI of all other clients as well.

```
actor:{
  deftype Requester;
  def selfType(){
   Requester
  };
  def groceryList;
  whenSignal: Runner discovered:{|groceries|
   groceryList := groceries;
   updateListGui(groceries);
   whenSignal: Market discovered:{|prices|
    def totalPrice := priceCalculator(groceries,prices);
    updateGui(groceries,totalPrice)
   }
  };
  def addGrocery(grocery){
   update: groceryList by: {|list| list.newItem(grocery)}
  }
 }
```

Listing 3. Implementation of errand requesters

From this minimal implementation of our grocery application it is clear that marrying the world of reactive programming and replicated programming offers significant benefits regarding the ease with which one can program interactive collaborative applications. Programmers are able to elegantly handle various events by letting Direst track and manage dependencies between signals. Moreover, programmers are freed from the burden of replicating signals and keeping their state consistent across replicas.

4. Replicating State Changes

In order to handle state changes to the replicated signals, any consistency algorithm can be employed. The manner with which the algorithm replicates and synchronises data is entirely decoupled from the behaviour of the replicated signal in the Direst model. A signal simply represents a value that can be replicated across clients/actors. These replicas are kept consistent by means of an underlying algorithm. A consequence of this decoupling is that any algorithm providing eventual consistency (e.g. CRDT, CloudTypes) could be used to implement replicated signals.

In our implementation of Direst we opted for the Repliq consistency model (Coppieters et al. 2016), since the unit of replication in Repliq are objects and Repliq is actor based.
The main idea behind the Repliq model goes as follows. Replicated objects must be isolated entities (i.e. the object does not have access to its lexical scope). Furthermore, arguments to the methods of the replicated object must be isolated entities as well. These restrictions allow the model to infer two crucial properties about these objects. First, the state of a replica can be determined solely using the replica's state at creation time and an ordered list of all methods executed on it. Second, changing the state of a replica can be achieved by undoing, redoing or reordering method invocations. Eventual consistency is then obtained by letting a master actor, which in this case is the actor invoking $signal$:, decide on the order in which the methods should be executed on all the replicas.
The ordering is achieved by implementing a version of the Global Sequence Protocol (Burckhardt et al. 2015).

An intuitive explanation of the protocol goes as follows.

Each actor has two logs per replica, the *confirmed* and *tentative* log. The confirmed log represents the state as observed by the master replica while the tentative log contains a list of operations (i.e. invoked methods) which have been applied locally to the replica. Whenever an operation is performed on a replica it is added to the tentative log. Moreover, a synchronization process will send the operation to the actor containing the master signal. There, the operation is executed on the master replica and added to the confirmed log after which all replicas will be notified of this addition. Upon reception of the notification, every replica will reset its state to the state at creation time, replay the entire confirmed and tentative log and remove the received operation from the tentative log if it is present. This protocol makes sure that eventually all the replicated signals have the same confirmed log, have executed the operations in the same order and thus converge.

This simplified explanation of the algorithm does not account for failures and message loss/reordering. The actual implementation in Direst uses GSP, such that it does work correctly under these circumstances. It makes sure that logs are numbered and durable, correct ordering is tested for and recovery processes are initiated whenever required (after failure or message loss).

Other consensus algorithms, such as Paxos (Lamport 1998; Lamport et al. 2001) and Raft (Ongaro and Ousterhout 2014), could also be used in order to let the logs converge. Yet, we choose to work with a master-replica protocol for two reasons. First, this allows every actor to have ownership over the replicated signals it creates, by owning the master object. This means that they can be in charge over what is allowed on the replicas. Second, this lets the master object act as the single source of truth. Although perhaps less scalable, it allows users to know when their operations are integrated into the master copy[2]. This is an important property when modelling for example client-server architectures using Direst, since it allows clients to know when their operations are made durable on the server. In contrast, this is not possible with a consensus protocol or something like CRDT.

5. Reacting to State Changes

This section describes each primitive offered by Direst and describes its semantics. The general idea behind our language is to provide distribution and distributed state solely through replication and reaction. Although we cannot prohibit programmers from employing traditional AmbientTalk functionality (e.g. asynchronous message sends), Direst provides all the functionality needed to write interactive and collaborative applications and should therefore be used as-

[2] We left the constructs that allow you to do this out of this paper, because they are not strictly required for Direst and are dependent on the replication algorithm.

is. We divide Direst's API into two categories: natives used for signal creation and natives used to initiate and handle propagation of change.

5.1 Signal Creation

Construction of signals can happen through one of three natives:

$signal$: One can explicitly create a signal using $signal$:, which takes a signal definition as argument and returns a signal object. A signal definition is a block statement which can contain fields and methods but does not have access to its enclosing scope. Given the distributed nature of our signals, this ensures that signals are self-sufficient entities. The resulting signal object is read and write protected (i.e. the object has no public interface).

$lift$ Direst allows programmers to lift regular functions (i.e. standard AmbientTalk functions), to work on signals. $lift$ takes a regular AT function as argument and returns a *lifted* version of that function. This lifted function differs from the original one in two ways. First, when applied to one or more signals the function will automatically track changes made to its signal arguments. That is, as soon as one of its arguments has changed the function will re-apply itself with the latest value for each argument. Second, the first application of a lifted function returns a signal which contains the result value of the original (i.e. non-lifted) function. As signal arguments to this application change, so does the value of the return signal. Applying a lifted function thus adds a dependency between the return signal and all argument signals.

$publish$: **and** $whenSignal$: $discovered$: Signals can also be obtained through the use of our publish/subscribe system which re-uses AmbientTalk's built-in discovery system in the background. Actors can publish a signal using the $publish$: as : construct which takes a signal and a type tag and will publish the signal under the given type tag. Subscribing to a signal (i.e. obtaining a replica of the signal) is done by invoking $whenSignal$: $discovered$: which takes a type tag and a function as argument. Given that a signal was published under the given type tag, the function is called with a replica of the signal as argument. Optionally one can also publish a signal using $publish$: as : $mutableIf$:, which takes an additional function specifying which actors can mutate the replicas of the signal. Concretely, this function will be attached to all replicas of the published signal. As soon as an actor tries to mutate the replica's state this function will be invoked with the mutating actor as argument. Only if the function evaluates to true will the actor be able to mutate the replica's state.

5.2 Change Propagation

Our DSL provides a single mean to start propagation: $update$: by :, which takes a signal and a function as argument. The function will be applied to the signal's current state (i.e. its behaviour) and will be able to invoke its mutators. We impose four limitations regarding this imperative update of a signal. First, only source signals (i.e. signals having no dependencies to other signals) can be imperatively updated. This also entails that imperatively updating a replica of a signal can only be done if the replica's master version is a source signal. This ensures that the declarative nature of our DSL is upheld. For example, when a lifted function is applied to a number of signals this can be viewed as a declarative specification of a dependency between the function's return signal and the argument signals. Allowing programmers to then imperatively change the resulting signal even if no argument signal has changed would be counter intuitive. Second, the update should not happen in the context of a lifted function application. This ensures that updates happen in a non-circular way (i.e. a signal cannot be updated as the result of its own update). However, due to the dynamic nature of AmbienTalk this last constraint cannot statically be enforced. Third, the function provided as argument to $update$: by : must only cause side-effects on the signal it is updating. As explained in Section 4, all updates to a signal might be replayed by the underlying synchronisation algorithm. If the update changes any other state than the signal's internal state this replaying may cause undesirable effects. This constraint can again not statically be enforced by Direst. Fourth, one can only mutate the replica of a signal if the $mutableIf$: function allows it. In case the master signal was published without this optional argument this last requirement is always met.

As soon as a source signal has been changed two things happen. First the changed signal's new value is propagated, in a topological order, to all signals which depend on this signal within the same actor. Given that dependencies are constructed using applications of lifted functions, this propagation entails a cascade of function re-executions. Second, our replication mechanism (see Section 4) starts, which will eventually entail that all replicas are changed as well. This will trigger a propagation of change on all actors which have a replica of the changed signal.

6. Related Work

We categorise two fields of related work. First, work pertaining to reactive programming in a distributed context. Second, libraries and models aimed at handling distributed state and replication.

6.1 Distributed Reactive Programming

AmbientTalk/R (Carreton et al. 2010) (AT/R) is an extension of the AmbientTalk interpreter which provides constructs for distributed reactive programming. As is the case for Direst it provides native constructs to create,update and distribute signals which it calls *ambient behaviors*. In this regard AT/R closely resembles Direst. However, distributed behaviours

are exported by copy between actors. As a result, changing the value of a copy of an ambient behaviour does not impact other copies.

DREAM (Margara and Salvaneschi 2014) is a middleware tailored towards distributed reactive programming. DREAM offers consistency guarantees that pertain to causality of events, glitch freedom and atomicity. It provides distribution of signals (or *observable objects* in DREAM terminology) through a publish/subscribe system. However, an observable object can only be modified by the component (i.e. distributed client) in which it was created. Moreover, the publish/subscribe system is used to notify *reactive objects* (which are similar to Direst's lifted functions) of updated values. This contrasts with the replication of actual signals in Direst. These two properties of DREAM make it unfit to easily represent shared mutable state.

Flapjax (Meyerovich et al. 2009) is a programming language designed to write web applications in a reactive fashion. However, flapjax only targets the client-side of web applications and does not provide reactivity for client-to-client interaction. As a result one cannot express distributed state in a reactive fashion.

The work presented in (Reynders et al. 2014) introduces a DSL for multi-tier functional reactive programming. In this work one is able to write full-stacked web applications (i.e. both client and server) using a single reactive language. The presented language allows programmers, amongst other functionality, to replicate streams between client and server. These replicated streams automatically push event values across tiers as they arise. However, streams are not shared amongst clients (i.e. two clients cannot mutate the same stream).

6.2 Replication Mechanisms

The research field related to consistency models for replicated state has produced a plethora of work since its conception (Saito and Shapiro 2005). Yet, recently a surge can be seen in novel models that aim to provide such consistency at the programming language level. We shortly discuss some of the more prominent and recent of these.

Most closely related is Lasp (Meiklejohn and Van Roy 2015). Lasp is a domain specific language that implements CRDT's at the programming language level. Furthermore, it also allows to use a restricted set of functions (functional and set-theoretic) over these values to create new CRDT's. By providing functions such as *map, filter, union,* etc. Lasp effectively integrates CRDT's as first-class values into the programming model. Yet, by no means does it incorporate the reactive behaviour of the CRDT's with the local reactivity such as user interaction and the remote reactivity of dependencies on non-replicated state.

CloudTypes (Burckhardt et al. 2012), just like CRDT's, offers pre-defined replicated values that have a pre-defined behaviour in a replicated setting. Again, it has no integrated reactive model to interact with other local and remote events. Also Vercast (Lorenz and Rosenan 2014) provides such application-level semantics, but has no integrated reactive model.

Finally, the introduction of replicated values in a distributed object-oriented system has been proposed before in Emerald with Gaggles (Black and Immel 1993). The key difference is that gaggles provide an interface to invoke methods on a foreign object that might have multiple instances. Consistency of the objects is not built-in, but can be built on top of the provided constructs. Our work specifically focusses on providing replicated objects such that actors can always access and use them locally, preserving actor locality. This inherently entails the usage of an optimistically replicated algorithm. Furthermore, gaggles also have no integration with or notion of reactivity.

7. Limitations and Future Work

A signal may only be published by an actor if that actor is the owner of the signal. We say that an actor owns a signal if it has either created the signal or if that signal solely depends on signals, directly or indirectly, which it has created. For example, a return signal of a lifted function which was applied to a replica of a signal is not owned by the applying actor. The sole reason to enforce this limitation is to avoid decentralised distributed glitches (Drechsler et al. 2014). Explaining this problem in detail would bring us out of the scope of this paper. However, current solutions to the problem cannot be applied to Direst without enforcing a centralised coordinator. Our model therefore simplifies dependencies between actors: one can never obtain a replica of another replica.

As explained in Chapter 4, the semantics of Direst are entirely decoupled from the consistency model used for the replicated signals. However, we provide no means yet to elegantly insert different consistency algorithms. Part of our ongoing research is to allow programmers to easily create their own replicated signals, providing their consistency model. This entails designing a suited meta protocol that provides the necessary hooks to implement every consistency model, ranging from entirely distributed peer-to-peer eventually consistent models such as CRDT, over master-slave replication such as Repliq to strongly consistent models using consensus.

8. Conclusion

Reactive and replicated programming both provide partial solutions to the problems one faces when programming interactive collaborative applications. Reactive programming allows for elegant handling of events and time changing variables such as UI input and network events. Replicated programming enables programmers to abstract away from state replication and synchronisation. However, both approaches have thus far only been developed and applied in isolation. Programmers using the reactive paradigm need to handle replicated and shared state manually while programmers us-

ing a replicated approach lack the high-level abstractions provided by the reactive paradigm to efficiently write event-driven code.

In this paper we introduce Direst: a domain-specific language which aims at marrying the best of both worlds. In Direst programmers are able to elegantly tackle events through the use of traditional reactive constructs such as signals and lifting. Moreover, signals in Direst provide built-in replication and synchronisation. Signals can be replicated across distributed clients using a publish/subscribe mechanism. Each client is able to mutate the state of these replicas while Direst automatically ensures that the states of these replicas are kept eventually consistent. Although the language currently exhibits some limitations, it is the first distributed reactive programming language to provide replicated state.

References

E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013. ISSN 0360-0300.

A. P. Black and M. P. Immel. Encapsulating plurality. In *European Conference on Object-Oriented Programming*, pages 57–79. Springer, 1993.

S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

A. L. Carreton, S. Mostinckx, T. Van Cutsem, and W. De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 41–60. Springer, 2010.

T. Coppieters, L. Philips, W. De Meuter, and T. Van Cutsem. An open implementation of cloud types for the web. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 2. ACM, 2014.

T. Coppieters, W. De Meuter, and S. 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*, page 3. ACM, 2016.

T. V. Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, and W. D. Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(34):112 – 136, 2014. ISSN 1477-8424.

E. Czaplicki and S. 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. ISBN 978-1-4503-2014-6.

J. Drechsler, G. Salvaneschi, R. Mogk, and M. 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. ISBN 978-1-4503-2585-1.

S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 114–125, New York, NY, USA, 1999. ACM. ISBN 1-58113-111-9.

L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

L. Lamport et al. Paxos made simple. 2001.

D. H. Lorenz and B. Rosenan. Versionable, branchable, and mergeable application state. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 29–42, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1.

A. Margara and G. 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. ISBN 978-1-4503-2737-4.

C. Meiklejohn and P. 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. ISBN 978-1-4503-3516-4.

L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. *SIGPLAN Not.*, 44(10): 1–20, Oct. 2009. ISSN 0362-1340.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

B. Reynders, D. Devriese, and F. 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. ISBN 978-1-4503-3210-1.

Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005. ISSN 0360-0300.

G. Salvaneschi, S. Amann, S. Proksch, and M. 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. ISBN 978-1-4503-3056-5.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7.