We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees

Alessandro Margara
Dept. of Computer Science
Vrije Universiteit Amsterdam
a.margara@vu.nl

Guido Salvaneschi
Software Technology Group
Technische Universität Darmstadt
salvaneschi@cs.tu-darmstadt.de

ABSTRACT

The reactive programming paradigm has been proposed to simplify the development of reactive systems. It relies on programming primitives to express dependencies between data items and on runtime/middleware support for automated propagation of changes. Despite this paradigm is receiving increasing attention, defining the precise semantics and the consistency guarantees for reactive programming in distributed environments is an open research problem.

This paper targets such problem by studying the consistency guarantees for the propagation of changes in a distributed reactive system. In particular, it introduces three propagation semantics, namely *causal*, *glitch free*, and *atomic*, providing different trade-offs between costs and guarantees. Furthermore, it describes how these semantics are concretely implemented in a Distributed REActice Middleware (DREAM), which exploits a distributed event-based dispatching system to propagate changes.

We compare the performance of DREAM in a wide range of scenarios. This allows us to study the overhead introduced by the different semantics in terms of network traffic and propagation delay and to assess the efficiency of DREAM in supporting distributed reactive systems.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Software**]: Concurrent Programming—*Distributed Programming*

Keywords

Dream, Distributed Reactive Programming, Consistency Guarantees, Glitch-Freedom, Event-Based Middleware

1. INTRODUCTION

Many modern software systems are *reactive*: they respond to the occurrence of events of interest by performing some computation, which may in turn trigger new events. Examples range from graphical user interfaces, which react to the input of the users, to embedded systems, which react to the signals coming from the hardware, to monitoring and control applications, which react to the changes in the external environment. Designing, implementing, and maintaining reactive software is arguably difficult. Indeed, reactive code is asynchronously triggered by event occurrences. Because of this, it is hard to trace and understand the control flow of the entire system [25].

The *reactive programming* paradigm [1] was proposed to mitigate these issues and simplify the development of reactive systems. It is based on three key concepts: time-varying values, tracking of dependencies, and automated propagation of changes. To explain the core principles of reactive programming, let us consider the following pseudocode snippets (pseudocode is sans-serif in the rest of the paper) that define a variable `a` and a variable `b` that depends on `a`:

```
1 var a: int = 10
2 var b: int = a + 2
3 println(b) // 12
4 a = 11
5 println(b) // 12
```

```
1 var a: int = 10
2 var b: int := a + 2
3 println(b) // 12
4 a = 11
5 println(b) // 13
```

In conventional imperative programming (left), any future change to the value of `a` does not impact the value of `b`. In reactive programming (right), by defining the second line as a *constraint* (denoted by `:=`) rather than a statement, we ensure that `b` gets constantly updated to reflect the latest value of `a`. In particular, the runtime environment identifies the dependency between `a` and `b` and propagates every change from `a` to `b`, forcing the recomputation of the latter.

This solution presents several advantages over the traditional Observer design pattern [20] adopted in event-based architectures to propagate changes. In particular, the programmer does not have to implement the update logic. Instead, she declaratively specifies the dependencies between variables and entirely delegates the update process to the runtime environment. Furthermore, the runtime environment takes care of ensuring the correctness of the propagation. For example, it can avoid the occurrence of *glitches*, i.e., temporary violations of data flow invariants [6]. This results results in more compact and readable code, and reduces the possibility of subtle errors. For example, in 2008 a half of the bugs reported for Adobe's desktop applications was generated in code for event handling [25].

Despite many reactive applications are intrinsically distributed (e.g., Web applications, monitoring applications, mobile applications) and despite the benefits of reactive pro-

DEBS'14, May 26 - 29, 2014, MUMBAI, India.

http://dx.doi.org/10.1145/2611286.2611290.

gramming in distributed settings have been recognized in the literature [35, 1, 28], most existing solutions for reactive programming do not support distribution [1]. Furthermore, the problem of defining suitable properties and consistency guarantees for the propagation of changes (e.g., to avoid glitches) has received little or no attention in distributed implementations [1, 28].

We argue that a precise definition of such properties and guarantees is an essential requirement for every system that supports the reactive programming paradigm. We also claim that the programmer should be able to choose the level of guarantees she prefers, based on the requirements in terms of semantics as well as computation and communication costs.

This paper focuses on distributed reactive programming (DRP) and specifically addresses the issues described above. First, it introduces an abstract model for DRP systems and defines some consistency guarantees for the propagation of changes. Starting from this analysis, it proposes three propagation semantics, providing different trade-offs between the properties they ensure and their implementation costs.

Next, the paper presents a concrete implementation of these semantics in a Distributed REActive Middleware (DREAM). DREAM is entirely implemented in Java to promote interoperability with existing software (e.g., Android applications) and exploits the REDS distributed event dispatching system [10] to propagate changes. Finally, the paper offers an extensive evaluation of the overhead introduced by the three proposed semantics in terms of network traffic and propagation delay.

To summarize, the paper contributes to the research on reactive programming in several ways: (*i*) to the best of our knowledge, it represents the first attempt to precisely study the consistency guarantees for the propagation of changes in DRP systems; (*ii*) it proposes three propagation semantics and extensively studies their costs; (*iii*) it presents the design, implementation, and evaluation of DREAM, a concrete middleware for DRP.

The rest of the paper is organized as follows. Section 2 presents the reactive programming paradigm and motivates the work. Section 3 introduces an abstract model for DRP and studies desirable consistency guarantees. Section 4 and Section 5 discuss the API and the implementation of DREAM. Next, Section 6 assesses the performance of the middleware. Finally, Section 7 surveys related work and Section 8 concludes the paper.

2. BACKGROUND AND MOTIVATION

To describe reactive programming in more details and to motivate the requirements for different levels of guarantees in the propagation of changes, let us consider a financial application system. The system holds a certain amount of stock options and performs forecast analysis on the stock market. The code snippet below sketches up a possible implementation for such system using reactive programming.

A data source module offers three time-changing variables: `marketIndex`, `stockOpts`, and `news`, which provide the current value of the stock market, the amount of stocks currently hold by the company, and the latest economics news from the Web. These variables are processed by three modules that compute the forecasts `f1`, `f2`, and `f3` for the hold stocks trend according to different financial models.

Constraints (denoted by `:=`) bind a variable to a *reactive expression* that involves other variables. For example, the variable `f1` is bound to the reactive expression `Model1.compute(marketIndex,stockOpts)`. When any of `marketIndex` or `stockOpts` changes its value, the variables that depend on them through a reactive expression are not valid anymore. For example, when `marketIndex` changes, the value of the model `f1` must be recomputed using the most recent value of `marketIndex`. In reactive systems, the runtime is responsible for automatically propagating changes and for recomputing some values, when needed. In the rest, we describe three alternative variants of the system (**V1**, **V2** and **V3**) and analyze their requirements in terms of consistency guarantees.

```
1  var marketIndex = InputModule.getMarketIndex()
2  var stockOpts = InputModule.getStockOpts()
3  var news = InputModule.getNews()
4
5  // Forecasts according to different models
6  var f1 := Model1.compute(marketIndex,stockOpts)
7  var f2 := Model2.compute(marketIndex,stockOpts)
8  var f3 := Model3.compute(marketIndex,news)
9
10 var gui := Display.show(f1,f2,f3)                          V1
11
12 var financialAlert := ((f1+f2+f3)/3) < MAX                V2
13 if (financialAlert) decrease(stockOpts)
14
15 var financialAlert_n := computeAlert_n(f1,f2,f3)          V3
16 if (financialAlert_n) adjust_n(stockOpts)
```

V1. Mobile application. In the first variant (box V1), the forecasts are transmitted to a GUI in a mobile application that displays their current values. As already discussed, a change in `marketIndex` triggers a recomputation of `f1`, `f2`, and `f3`. Let us assume that the recomputation of `f1` terminates first. The change propagates to the `gui`, which is updated to display the new value of `f1`. In this scenario, `gui` is recomputed with the *new* value of `f1` but still the *old* values of `f2` and `f3`. The user experiences a temporary inconsistency in the values that are displayed (a *glitch*, as we will explain soon). However, such an inconsistency does not represent a serious issue for this application: as soon as the new values of `f2` and `f3` are computed, the `gui` is refreshed and reflects the current value of `marketIndex`.

V2. Models aggregator. In the second variant of the system (box V2), forecasts `f1`, `f2`, and `f3` are aggregated in the `financialAlert` variable. To prevent money loss, an automated procedure starts selling stock options when the `financialAlert` issues a warning, i.e. when the mean of `f1`, `f2`, and `f3` drops below a threshold `MAX`.

Let us consider again a change in the `marketIndex`. As in the previous example, as soon as `f1` is recomputed, the value of `financialAlert` is invalidated. However, if the expression associated to `financialAlert` is immediately recomputed with the *new* value of `f1` but the *old* values of `f2` and `f3`, `financialAlert` can hold a *wrong* value.

For example, let us assume that the following set of changes occurs `f1`: 110 → 90, `f2`: 95 → 111, `f3`: 99 → 103 and `MAX`=100. If the mean is recomputed with the new value of `f1` and the old values of `f2`, `f3`, it amounts to (90+95+99)<100 and the alert is triggered. The error is only temporary, because as soon as `f2` and `f3` are available, the expression bound to `financialAlert` is recomputed to the correct value. It is clear, however, that in variant V2 such a behavior is potentially catastrophic. In our example,

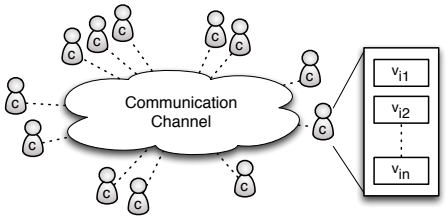


Figure 1: A Model for DRP.

the application can place a selling order because the value of `financialAlert` is temporarily wrong.

The arising of temporary spurious values, commonly referred to as *glitches* is a well known issue in (local) reactive systems, which control the order of updates to avoid them. In our example, waiting for `f1`, `f2`, and `f3` before recomputing `financialAlert` would avoid the occurrence of glitches. In a distributed environment, where different variables can be stored at different nodes, enforcing a specific evaluation order introduces some overhead for node synchronization. Such overhead should be paid only in case glitch freedom is a strict requirement.

V3. Multiple aggregators. In the third variant of the system (box V3), the decision making process is split among `n` components, each one computing a different `financialAlert` and taking autonomous decisions on how to adjust the `stockOpts` variable. The actions of the different components are complementary: however, the system provides the expected behavior only under the assumption that all the components see the changes to `f1`, `f2`, and `f3` in the same order. For example, if different components see different interleaving of `marketIndex` and `news` updates, they may disagree on the value of `f3` over time and take contradictory measures to adjust the `stockOpts` variable.

This variant shows that in some cases glitch freedom is not enough to guarantee the correct behavior of the application and exemplifies the need for a system-wide agreement on the order of updates.

In summary, the previous discussion shows that there is no *one size fits all* solution. Instead, different levels of consistency guarantees are required and developers must be able to select the best trade-off between performance and consistency for their application.

3. A MODEL FOR DRP

This section introduces a model for DRP and the terminology we adopt throughout the paper. In particular, Section 3.1 presents the entities involved in a DRP system and the operations they perform. Next, Section 3.2 formalizes a set of consistency guarantees for DRP.

3.1 System Architecture

Figure 1 shows the high level view of our model for DRP.

Components. We consider a distributed system consisting of n *components* $c_1 \ldots c_n$ that can exchange messages using *communication channels*. We model the state of each component c_i as a set of *variables* $V_i = \{v_{i1} : \tau_{i1} \ \ldots \ v_{im} : \tau_{im}\}$, where v_{ij} represents the name of variable j in component c_i and τ_{ij} its type. We say that a variable v_{ij} is defined in the *scope* of component c_i if $v_{ij} \in V_i$. Each component can directly access (read and write) only the value of variables defined in its scope.

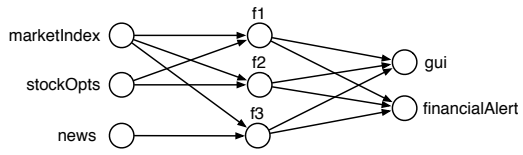


Figure 2: Dependency graph for the stock market scenario.

Variables. Beside traditional *imperative* variables, components can define *reactive* and *observable* variables. A *reactive variable* v is defined through an expression e and gets automatically updated. Whenever the value of one of the variables that appear in e changes, e is recomputed to determine the new value of v (lazy evaluation is for future work). A component can read the current value of a reactive variable defined in its scope at any point during its execution. However, it cannot directly modify its value or the expression defining it.

With reference to our stock market scenario, `f1` is a reactive variable defined through the reactive expression `Model1.compute(marketIndex,stockOpts)`. When either `marketIndex` or `stockOpts` change, the value of `f1` gets automatically updated.

Observable variables are those that can be used to build reactive expressions. In the stock market scenario in Section 2, `marketIndex`, `stockOpts`, and `news` are observable variable, used in the definition of `f1`, `f2`, and `f3`. Our model supports `local` observable variables, that are only visible in the scope of component c_i and can be used only to define reactive variables in c_i, and `global` observable variables can be used in every component. By default, observable variables are global. A fundamental feature to support composability is that reactive variables can be (locally or globally) observable as well. This way, reactive variables can be composed into reactive expressions to create new reactive variables. For example, in the stock market scenario of Section 2, the reactive variables `f1`, `f2`, and `f3` are used to define another reactive variable `gui`.

Dependency Graph. We call *dependency graph* a directed graph $D = \{V, E\}$, where V is the set of all the observable and reactive variables in the system and E is the set of all the edges e_{ij} that connect a variable v_i to a variable v_j if and only if v_j is a reactive variable defined through an expression e_j that includes v_i. We say that a variable v_a depends on a variable v_b if there is a path from v_a to v_b. As an example, the stock market scenario in Section 2 produces the dependency graph shown in Figure 2. The `gui` variable depends on both `marketIndex`, `stockOpts`, and `news`.

Reactive programming needs to be supported by the runtime environment, which is responsible for automatically and recursively propagating the changes from a variable v_i to all the variables that depend on v_i.

Events. We represent the occurrence and the propagation of changes in a DRP system using three events that model the possible operations on the variables in the system:

- a *write* event $w_x(v)$ occurs when a component writes the value x on an variable v;
- a *read* event $r_x(v)$ occurs when a component reads the value x from a variable v;
- an *update* event $u(S, w_x(v))$, $S = \{w_{y1}(v_1) \ldots w_{yn}(v_n)\}$ occurs when some changes applied to variables $v_1 \ldots v_n$

(for simplicity, known in advance) are propagated, triggering a change (write of a new value x) in the depending reactive variable v.

The read and write events represent the way components access local variables. We assume that these operations are atomic. Update events model the propagation of changes through the communication channel. Next, we show how to specify some consistency guarantees for DRP as ordering of write and update events.

3.2 Consistency Guarantees

The update of reactive variables takes place orthogonally with respect to the execution flow of the program in each component. Accordingly, it is critical for the programmer to understand how the update process takes place, and in particular which properties and guarantees it provides, e.g., with respect to the order in which updates are propagated. We collectively refer to them as *consistency guarantees*.

Intuitively, there is a trade-off between the level of guarantees offered and the complexity of the update algorithm. Ideally, a change in an observable variable should be propagated atomically and instantaneously through the dependency graph to all the reactive variables that depend on it. This section studies in details some consistency guarantees for DRP systems.

Exactly once delivery. A system provides exactly once delivery if a change in the value of a variable v is propagated once and only once to all variables depending on v. More formally, if $w_x(v)$ occurs, then $u(S_i, w_y(v_i))$, $w_x(v) \in S_i$ occurs once and only once $\forall v_i$ such that v_i depends on v.

This property ensures that, in absence of failures, the communication channel does not loose or duplicate any message used for the propagation of changes.

FIFO ordering. A system provides FIFO ordering if the changes to the value of a variable v in component c are propagated in the order in which they occur in c. More formally, $\forall v_i, v_j$ such that v_j depends on v_i, if $w_{x1}(v_i)$ occurs before $w_{x2}(v_i)$, then $u(S_1)$, $w_{x1}(v_i) \in S_1$ occurs before $u(S_2)$, $w_{x2}(v_i) \in S_2$.

Causal ordering. A system provides causal ordering guarantee if the events that are potentially causally related are seen by every component of the system in the same order. More formally, with reference to [22], we define a *happened before* ($\rightarrow$) partial order relation between the events in a DRP system as follows:

- if two events e_1, e_2 occur in the same process, then $e_1 \rightarrow e_2$ if and only if e_1 occurs before e_2;
- if $e_1 = w_x(v_i)$ and $e_2 = u(S_i, w_y(v_j))$, $w_x(v_i) \in S_i$, then $e_1 \rightarrow e_2$ (a write w happens before every update u depending on w);
- if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$ (the happened before relation is transitive).

We say that a system provides causal consistency guarantee if and only if, for all events e_1, e_2 such that $e_1 \rightarrow e_2$, all the components of the DRP system see e_1 before e_2. Notice that the happened before is a partial order relation: events that are not causally related can be seen in any order by different components. Because of this, the causal ordering guarantee does not imply that all components see all events in the same order.

Glitch freedom. Consider a reactive variable v that depends on the set of variables V_d. Let us call $V_{d1} \subseteq V_d$ the set of variables in V_d that depend, directly or indirectly, from a variable v_1. Each change in variable v_1 reflects on variable v since it impacts on the variables in V_{d1}. We call an update of v $u(S, w_x(v))$ a *partial update* if it is triggered by a change in v_1 and $S \subset V_{d1}$ (i.e., S does not propagate all the changes occurred in the set V_{d1}). A system provides glitch freedom if it never introduces *partial updates*.

To better explain this concept, let us refer again to the stock market monitoring scenario shown in Section 2. When the value of `marketIndex` changes, it influences the forcasts `f1`, `f2`, and `f3`. With reference to the `financialAlert` variable, a glitch occurs if the variable gets recomputed using only *some* of the new forcasts (partial update). No glitch occurs if `financialAlert` is recomputed only when *all* the new values for `f1`, `f2`, and `f3` are available.

Atomic consistency. A system provides atomic consistency guarantee if the following conditions hold: (i) the system provides FIFO ordering guarantee; (ii) each write event $w_x(v)$ on a variable v is atomically propagated to all the variables that depend (directly or indirectly) on v, i.e., all the update events $u(S_i, w_y(v_i))$ triggered (directly or indirectly) by $w_x(v)$ are executed in a single atomic operation.

Intuitively, the atomic consistency guarantee ensures total order of events (it is more strict than the causal order guarantee). At the same time, it also ensures atomicity of propagation: no operation can occur *while* a propagation is taking place. Finally, since propagations occur atomically, it also ensures glitch freedom.

4. DREAM: API

This section describes the API of DREAM. To promote interoperability with existing codebases, DREAM is entirely written in Java. In particular, variables are represented by Java objects. The remainder of this section shows how to define, instantiate, and use observable and reactive objects.

4.1 Observable Objects

In DREAM, observable objects are generic Java objects that inherit from the `Observable` abstract class. As for every Java object, the internal state of an observable object can be accessed (read or modified) through method invocation. We call *observable method* each method of an observable object that returns a non-`void` result.

Let us consider an observable object `obj` with an observable method `obm` and a generic method `m`. We say that `m` *impacts on* `obm` if a call to `m` on `obj` may potentially change the return value of `obm`.

As discussed in the following, an observable method `obm` can be used to define a reactive object. For this reason, the runtime environment needs to capture every call to a method that impacts on the value of `obm` and propagate the change to any depending reactive objects[1].

An automated detection of the impacts on relation among methods is undecidable in the general case [37]. DREAM solves this problem by asking the programmer to make this

[1] We assume that the state of an observable object can be altered only through method invocation. This is a best engineering practice in object oriented programming.

```
1 public class ObservableInteger extends Observable {
2   private int val;
3
4   // Constructors ...
5
6   @ImpactsOn(methods = { "get" })
7   public final void set(int val) {
8     this.val = val;
9   }
10
11  public final int get() {
12    return val;
13  }
14 }
```

Listing 1: Definition of ObservableInteger

relation explicit through an ad-hoc Java annotation. An approximation using static analysis is left for future work.

Listing 1 shows the definition of a simple `Observable` class: `ObservableInteger`. As Listing 1 shows, defining a new type of observable objects in DREAM requires two actions. (*i*) inherit from the `Observable` abstract class. (*ii*) annotate each method `m` with the set of observable methods it impacts on, by using the `@ImpactsOn` annotation. For example, in Listing 1, `set` impacts on the observable method `get` since a call to `set` modifies the value of variable `val` that is read by `get`. DREAM provides built-in observable types for integers and floating point numbers, strings, and different kinds of collections. Users can create new classes of observable objects using the procedure above.

The instantiation of observable objects is shown in Listing 2, Lines 2-5. The code snippet is commented in details in the rest, after introducing DREAM support for naming and discovery.

4.2 Reactive Objects

In DREAM, a reactive object is created from a reactive expression using the `ReactiveFactory` class. A reactive expression can combine constant values and return values of observable methods. For example, the reactive object generated in Listing 2, Line 9, contains an integer value defined from the observable method `get` of object `obInt` and the constant value `2`. The reactive object generated in Line 11 contains a string value defined starting from two observable methods `get` from objects `obStr1` and `obStr2`. The value of a reactive object cannot be directly modified, but only retrieved by invoking a `get` method (Line 14).

When defining a reactive object `r` in DREAM, the programmer can specify two initialization semantics. The *blocking* semantics blocks every read access to `r` until the current values of all the observable methods `r` depends on have been retrieved and the value of `r` has been successfully computed. The *non-blocking* semantics enables the programmer to specify an initial value that the variable holds until its expression can be computed.

Similar to observable objects, DREAM provides several built-in types of reactive objects. Programmers can use them to define the state of more complex objects. Additionally, they can define their own types of reactive objects and the expressions to combine them[2].

4.3 Naming and Discovery

In DREAM, we assume that a unique name identifies each component of the distributed system. Similarly, a (locally) unique name identifies each observable object inside

[2]In DREAM, expressions are defined using ANTLR grammars [32].

```
1 // Component c1
2 ObservableInteger obInt =
3   new ObservableInteger("obInt1", 1, LOCAL);
4 ObservableString obStr1 =
5   new ObservableString("obStr1", "a", GLOBAL);
6 ObservableString obStr2 = ...
7
8 // Component c2
9 ReactiveInteger rInt = ReactiveFactory.
10   getInteger("obInt.get()*2");
11 ReactiveString rStr = ReactiveFactory.
12   getString("obStr1.get()+obStr2.get()");
13 while(true){
14   System.out.println(rStr.get())
15   Thread.sleep(500)
16 }
17
18 // Component c3
19 ReactiveInteger strLen =
20   ReactiveFactory.getObservableInteger
21   ("c1.obString1.get().length()", "obString1Len");
```

Listing 2: Creation of observable and reactive objects

a component. The programmer can reference an observable method `obm` of an object `obj` inside a component `c` using the notation `c.obj.obm`. The name of the component can be omitted when referencing a local object.

The name of an observable object and, optionally, its visibility (local or global) are assigned at instantiation time. For example, consider Listing 2. The statements in Lines 2-6 are executed in component `c1`. Line 2 defines an `ObservableInteger` object announced as `obInt1`, an initial value of `1`, and local visibility. Line 4 defines an `ObservableString` object announced as `obStr1`, initial value `a`, and global visibility. Reactive objects can also be defined as observable by simply specifying a name for them. For example, Line 19 in Listing 2 (executed in component `c3`) defines a `ReactiveInteger` object that contains the lenght of string `c1.obString1` and makes it an observable object by associating a unique name to it (`obString1Len`). Since no additional parameters are specified, the object has default global visibility.

To discover the observable objects available in the system, DREAM offers a flexible discovery service. It allows a component to retrieve the full name of observable objects searching by type, component, local name, or a combination of these fields.

5. DREAM: IMPLEMENTATION

This section describes the implementation of DREAM in details, showing the protocols for automated propagation of changes in a distributed environment, and discussing the consistency guarantees they offer. Figure 3 shows the architecture of the middleware, which implements the abstract model presented in Figure 1.

DREAM consists of two parts: a *client* library, adopted by each component of the distributed system (denoted as `C` in Figure 3) and a distributed event-based communication infrastructure, consisting of one or more *brokers* (`B` circles in Figure 3) connected to form an acyclic undirected overlay network. The broker infrastructure implements the communication channel between components. An optional registry (`R` in Figure 3) acts as a persistent storage.

5.1 Publish-Subscribe Communication

In DREAM, the propagation of changes from observable to reactive objects is implemented using a publish-subscribe

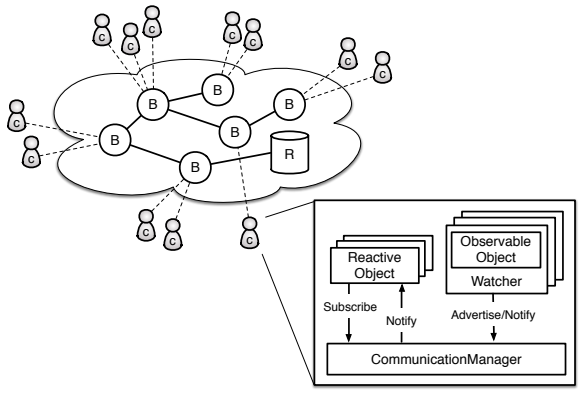


Figure 3: The architecture of DREAM.

communication paradigm [15], based on the following three primitives:

- advertise(c,obj,obm) is used by a component c to notify the middleware that an object obj with an observable method obm has been defined in its scope;
- subscribe(c,obj,obm) is used by a component to notify the middleware about its interest in the value of the observable method obm of object obj in component c;
- notify(c,obj,obm,val) is used to notify the middleware that a new value val has been computed for the observable method obm of object obj in component c.

We refer to the content of these primitives as *advertisement*, *subscription*, and *notification*, respectively. If an advertisement and a subscription refer to the same observable method, we say that the subscription *matches* the advertisement. Similarly, if a subscription and a notification refer to the same method, we say that the notification matches the subscription.

5.2 Clients

The `CommunicationManager` component implements the core functionalities of the DREAM client library. It manages the local communication among objects inside the same component and serves as a proxy for global communication involving multiple components.

Instantiating observable and reactive objects. When the program defines a new type of observable object, DREAM automatically weaves a `Watcher` advice, using aspect-oriented technology [21]. The `Watcher` advice interacts with the `CommunicationManager` without any additional effort for the programmer. The interaction serves two purposes. (1) Advertise new observable objects as soon as they are created. When a new observable object `obj` is instantiated, the `Watcher` notifies the `CommunicationManager` that `obj` has been created and informs the `CommunicationManager` of each observable method in `obj`. (2) Detect changes of the return value of the observable methods in `obj`. For each method `m` of `obj`, the `Watcher` uses Java reflection to identify all the observable methods influenced by `m`, as specified by the `ImpactsOn` annotation. When `m` is called on `obj`, the `Watcher` recomputes the value of each of these methods and notifies the new value to the `CommunicationManager`.

When the programmer instantiates a new reactive object with expression `exp` via the `ReactiveFactory`, DREAM automatically forwards to the `CommunicationManager` a subscription for each observable method in `exp`.

When a component `c` is launched, the `CommunicationManager` of `c` connects to a single broker of the communication infrastructure. It propagates through the broker all the advertisements generated by (global) observable objects defined in `c` and receives subscriptions for them. In addition, it stores subscriptions of local reactive objects.

Notifications. Upon receiving a notification `n` (either from the remote broker or from a local observable object), the `CommunicationManager` dispatches it to all interested objects, i.e., to all the reactive objects providing a subscription that `n` matches. If `n` is a local notification matching at least one remote subscription, the `CommunicationManager` also delivers `n` to the remote broker.

In DREAM, each component is responsible for concretely computing the expression `expr` for each reactive object `obj` defined in its scope. In particular, a recomputation is triggered upon receiving a new notification of change from the `CommunicationManager`. In the case of causal consistency, each change event can be processed immediately. In contrast, glitch freedom may require that the evaluation of a change is postponed until other notifications are received. As we describe soon, the broker network is responsible for detecting dependencies among events and notify them to the component. The `CommunicationManager` employs this information to temporarily queue notifications until they can be processed without introducing glitches.

The `CommunicationManager` implements the communication within a single component using local method invocation. Thus, propagating the change of an observable method to a reactive object in the same component does not involve any overhead due to remote communication.

5.3 Brokers

The communication between brokers is implemented using the subscription forwarding protocol [8] on top of the REDS event dispatching system [10]. This protocol relies on the assumption that the brokers are connected into an acyclic topology and works in three steps: (*i*) Advertisements are propagated to all the brokers of the network. (*ii*) Subscriptions get propagated only towards components that provided matching advertisements. (*iii*) Similarly, notifications are propagated only towards components that provided a matching subscription.

The forwarding algorithm implemented by brokers is shown in Listing 3 (simplified for the presentation). Each broker maintains a list of all its neighbors, i.e., all the nodes (both brokers and components) directly connected to it. Also, it stores the list of advertisements received from each neighbor (`advTable` in Listing 3) and the list of subscriptions received from each neighbor (`subTable` in Listing 3).

Upon receiving an advertisement `adv` (function `processAdv` in Listing 3), the broker first updates the `advTable` structure and then forwards `adv` to all its neighbors, except the sender. Similarly, upon receiving a subscription `sub` (function `processSub` in Listing 3), the broker first updates the `subTable` structure, then looks for the neighbor with an advertisement that matches `sub` and forwards the subscription to it. Finally, upon receiving a notification `n` (function `processNotif` in Listing 3), the

```
1 Map<Node, List<Advertisement>> advTable;
2 Map<Node, List<Subscription>> subTable;
3
4 void processAdv(Advertisement adv, Node sender) {
5   advTable.add(adv, sender);
6   sendToAllNodeExcept(sender, adv);
7 }
8
9 void processSub(Subscription sub, Node sender) {
10  subTable.add(sub, sender);
11  for (Node node : advTable.keySet()) {
12    List<Advertisement> advs = advTable.get(node);
13    if (!node.equals(sender) && matches(sub, advs)) {
14      sendToNode(node, sub);
15      break;
16    }
17  }
18 }
19
20 void processNotif(Notification n, Node sender) {
21  for (Node node : advTable.keySet()) {
22    List<Subscription> subs = subTable.get(b);
23    if (!node.equals(sender) && matches(n, subs)) {
24      sendToNeighbor(node, n);
25    }
26  }
27 }
```

Listing 3: Forwarding algorithm implemented in brokers

broker forwards `n` to all the neighbors with at least one subscription matched by `n`.

Discovery and Initialization. As discussed above, advertisements of globally visible observable objects are delivered to all brokers of the network. This allows DREAM to replicate the discovery service at each broker. Specifically, when a component `c` connected to a broker `B` issues a lookup request (to retrieve observable objects by name, type, or component), `B` can process the request locally, based on the information contained in its list of advertisements.

Concerning initialization, when a new reactive object is defined with an expression `exp`, the DREAM client library forwards a subscription for each observable method in `exp`. This subscription triggers a retransmission of the latest value of all these observable methods, enabling the new reactive object to be initialized.

Finally, DREAM also allows observable objects to be persisted: each notification of change is stored in a special broker (the `Registry`, `R` in Figure 3). Thanks to this solution, if the component defining an observable object `obs` disconnects from the system, the latest value of all the observable methods defined for `obs` can still be retrieved from `R`.

5.4 Implementing Consistency Guarantees

DREAM offers three propagation semantics with an increasing level of consistency guarantees. (*i*) *Causal semantics* provides exactly once delivery with causal ordering. (*ii*) *Glitch free semantics*, adds glitch freedom. (*iii*) *Atomic semantics*, implements atomic consistency.

Causal Semantics. In DREAM, both the component-to-broker communication and the broker-to-broker communication are implemented using point-to-point TCP connections, which provide exactly once delivery guarantee and FIFO order. To ensure that FIFO order is preserved between any given pair of components, processing inside each intermediate broker must preserve message ordering. Currently, DREAM ensures this property by processing messages sequentially, in a single thread. Alternative implementations that feature parallel processing should take care of reordering messages before delivery.

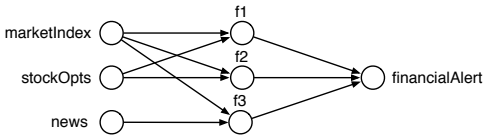


Figure 4: The dependency graph for the stock market system (Variant V2).

Finally, as proved in [19], an acyclic topology that preserves end-to-end FIFO ordering is sufficient to ensure that messages are delivered in causal order.

As mentioned in Section 5.2, in DREAM, each reactive object `obj` is responsible for recomputing the value of its own expression `expr`. In the case of causal consistency, `obj` immediately starts a recomputation upon receiving a notification from an observable object defined in `expr`. No further constraints are required to ensure causal consistency.

Glitch Free Semantics. Existing solutions for reactive programming provide glitch freedom on single machine implementations by propagating changes in the dependency graph in topological order (breadth first). This ensures that an expression is always evaluated after all its dependents [6, 30, 25]. To achieve the same result in a distributed environment, we modified the algorithm for the propagation of updates described in Listing 3 as follows.

First, when a new reactive object is created, its defining expression is propagated to all the brokers in the communication infrastructure. This way, each broker gains complete knowledge of the graph of dependencies among objects and can identify possible sources of glitches. With reference to the definition provided in Section 3.2, a glitch occurs when the value of an observable method changes and the value of a depending reactive object is updated before seeing *all* the effects of the original change.

Let us consider again our stock market scenario. For ease of explanation, the dependency graph for the variant V2 of the system is shown in Figure 4. We assume that each variable is stored on a different component.

In this example, a glitch occurs if a change in `marketIndex` affects the values of `f1`, `f2`, and `f3`, but `financialAlert` gets updated upon receiving only some notifications of change (e.g., only from `f1`). We refer to such a notification as a partial update, since it does not contain all consequences of the original change in `marketIndex`.

To prevent glitches, DREAM implements a queuing mechanism, such that partial notifications are temporarily stored at the components until all the consequences of a change become available. Specifically, before delivering a notification `n` to a component `c`, a broker computes the set of additional notifications that `c` needs to receive before processing `n` and attaches this information to the message containing `n`. With reference to our example in Figure 4, when a broker delivers a notification `n` about a change in `f1` to `financialAlert`, it specifies that `n` can be processed only if two notifications about `f2` and `f3` have been received.

Notice that, when processing `n`, a broker cannot determine the set of notifications a component should wait for by looking only at the content of `n`. In our example, a change in `f3` could be triggered either by a change in `marketIndex` or by a change in `news`. In the first case, the expression defining `financialAlert` can be recomputed immediately upon the reception of `n`. In the second case, the expression

cannot be recomputed until the new values of f2 and f3 (also influenced by a change in marketIndex) are propagated. Because of this, a broker must be aware of the history of changes that caused the notification under processing. In case of glitch free semantics, DREAM attaches this information to each event notification. In our example, a notification about a change in f3 caused by marketIndex will be labeled with marketIndex. Similarly, a notification about a change in f3 caused by news will be labeled with news.

Since the dependencies are always computed in the broker network, the glitch free protocol introduces a significant difference with respect to the causal protocol: it always requires notifications to be forwarded through the broker network. Conversely, in the causal consistency protocol, each component (and in particular the ConnectionManager within each component) could dispatch notifications to local subscribers without contacting the broker network.

Finally, it is easy to see that the protocol for glitch freedom preserves causal ordering since the properties of the communication infrastructure are not changed. The broker-to-broker network is acyclic and guarantees exactly once delivery. FIFO order is preserved since notifications are always delivered inside the broker network. Crucially, the queuing mechanism implemented at the components does not change message order. A notification waits in the queue until all the notifications related to the same reactive expression arrive, which may result in a delay in the expression evaluation but not in a change in the message order.

Atomic Semantics. The atomic semantics adds total order of changes on top of the glitch free semantics. DREAM ensures total order by implementing a locking mechanism based on a special centralized component in the communication infrastructure: the Ticket Granting Server (TGS). Upon receiving a new change notification n from an observable method obm, a broker B contacts the TGS to obtain the right of propagating the change to all the reactive objects that (directly or indirectly) depend on obm. The protocol works as follows: (*i*) B sends n to the TGS. (*ii*) The TGS computes all the leaves in the dependency graph that n should reach. (*iii*) The TGS stores the notifications in a FIFO queue and delivers them to the broker network one at a time. (*iv*) While processing a notification, when a broker B sends an update for a leaf in the dependency graph, it notifies the TGS. Upon receiving an acknowledgement of delivery to all the leaves in the dependency graph, the TGS considers the propagation of a change completed, and starts the propagation of the next change in the queue.

Since the protocol used in the communication infrastructure is identical to the one described for glitch freedom, all guarantees introduced in the previous sections are preserved. The use of a locking mechanism adds the guarantee that only one component at a time can execute an update.

6. EVALUATION

Our evaluation is twofold. First, we perform a *comparative* assessment of the cost of ensuring different consistency levels and analyze the factors that influence such a cost. Second, we investigate the *absolute* performance of our current implementation – including communication overhead and evaluation of reactive expressions – to show that DREAM is usable in practice.

For the first goal, we created an emulated network using the Protopeer network emulator [16], on top of which we run multiple DREAM brokers offering the DRP service to several components. This allowed us to easily control a number of network parameters. For the second goal, we tested the processing and communication overhead of DREAM on a real machine.

Number of brokers	10
Number of components	50
Topology of broker network	Scale-free
Percentage of pure forwarders	50%
Distribution of components	Uniform
Link latency	1 ms–5 ms
Number of reactive graphs	10
Size of dependency graphs	5
Size of reactive expressions	2
Degree of locality in expressions	0.8
Frequency of change for observable objects	1 change/s

Table 1: Parameters used in the default scenario.

6.1 Analysis of the Protocols for DRP

In this first set of experiments, we compare the costs of our DRP protocols while providing different levels of consistency guarantees. We focus on two metrics: network traffic and propagation delay. To compute network traffic, we consider both component-to-broker communication and broker-broker communication and we measure the average traffic flowing in the entire network every second. To compute propagation delay we compute the average difference between the time an event is produced, i.e., when the value of an observable object obs changes, and the time an event is processed, i.e., the change is used to update the value of a reactive object that depends (directly or indirectly) on obs.

Experiment Setup. Several parameters can impact on the performance of our protocols. To isolate their effect, we defined a default scenario, whose values are listed in Table 2. Next, we performed several experiments, changing a single parameter in each one.

Our default scenario consists of 10 brokers connected in a randomly-generated scale-free overlay network and serving 50 components. We assume that 50% of the brokers are pure forwarders, i.e., do not have any directly connected component. Components are uniformly distributed among the remaining brokers. The latency of network links is uniformly distributed between 1 ms and 5 ms. We assume that the time required to process a packet is negligible with respect to the link latency[3].

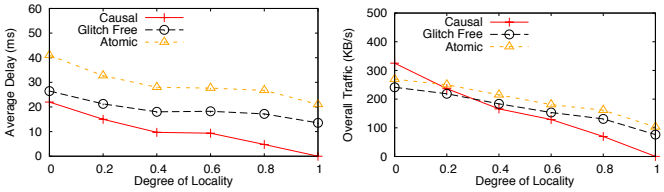
We consider 10 separate dependency graphs, each of them including an observable object (the original source of changes) and 4 reactive objects that depend directly or indirectly from it. Each reactive object is defined by a randomly generated expression including 2 operands (on average). Each observable object changes its value every second (on average). To simulate different application loads, we introduce a locality parameter: given a reactive object r depending on an observable object obs, the locality parameter defines the probability that r and obs are defined in the same component. With locality 1, all objects of the graph are deployed on a single component. Conversely, with locality 0, objects are randomly distributed over all available components.

Our experiments consist of two phases. In the first phase

[3]Several algorithms have been presented in the literature for efficient event dispatching [26, 27, 34]. Further, Section 6.2 validates the assumption using the current DREAM implementation.

	Delay (ms)		Traffic (KB/s)	
	Centr.	Distr.	Centr.	Distr.
Causal	4.77	4.76	68.3	69.8
Glitch free	29.53	17.18	205.4	130.9
Atomic	53.41	26.75	265.5	161.3

Table 2: Centralized vs. distributed broker networks.



(a) Delay (b) Traffic

Figure 5: Locality of Expressions.

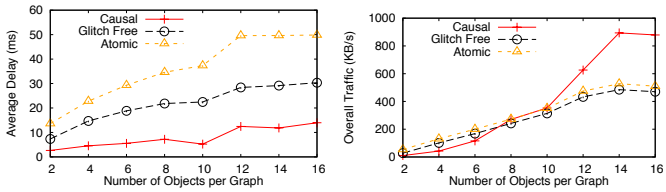
the components define and deploy observable and reactive objects. In the second phase, the value of observable objects is repeatedly changed and propagated over the reactive graphs by the DRP service.

Advantages of Distribution. Our first experiment investigates the advantage of using a network of brokers for the DRP service. We compare a centralized solution with the distributed solution in our default scenario. More in particular, the two solutions consider the same (physical) network topology, but deploy a different number of brokers (1 in the centralized scenario, 10 in the distributed scenario).

Table 2 shows the results we measured: the level of consistency guarantees significantly influences the performance of DREAM in both scenarios. Let us first consider the average change propagation delay. The differences we measured in the default (distributed) scenario (Table 2, Column 2) can be explained as follows. When providing causal consistency, each component processes local changes without contacting the broker network. Since our default scenario exhibits a high degree of locality in expressions, most changes can be processed within components. In contrast, the protocol for glitch freedom requires each and every event to be processed by at least one broker. Atomic consistency requires additional communication to acquire the lock from the TGS.

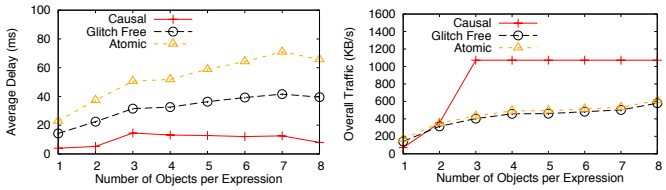
If we consider the centralized scenario (Table 2, Column 1), we observe that, in the case of causal consistency, the delay is almost identical to the one measured in the distributed scenario. Also in this case, most of the events can be processed locally at components. In contrast, the protocols for glitch freedom and atomicity forward all the events though the centralized broker. While in our default scenario the filtering and forwarding of events was performed incrementally at each broker, as close as possible to the event source, in a centralized environment all events need to reach the single broker to be processed and forwarded, which causes a significant increase in the average delay.

The considerations above apply for network traffic as well (Table 2, Columns 3-4). The causal consistency protocol needs to forward less notifications, the protocol for glitch freedom and atomic consistency are comparable, but the latter produces more traffic because it includes the communication with the TGS. We can conclude that the idea of adopting a distributed broker network for the propagation of changes offers significant benefits in terms of network traffic and propagation delay.



(a) Delay (b) Traffic

Figure 6: Size of reactive graphs.



(a) Delay (b) Traffic

Figure 7: Size of Reactive Expressions.

Locality of Expressions. The previous experiments show the capability of the causal consistency protocol to exploit locality. To further investigate this aspect, we tested the performance of DREAM when changing the degree of locality (Figure 5). All protocols reduce the average propagation delay as the degree of locality increases (Figure 5a): indeed, as the locality increases, the number of brokers used to forward an event decreases. As expected, the causal consistency protocol benefits more than the others from the increased degree of locality.

The results concerning network traffic (Figure 5b) are particularly interesting. The glitch free and the atomic protocols generate similar amounts of traffic – the latter being slightly more expensive because of the communication with the TGS. Noticeably, the traffic generated by the protocol for causal consistency drops much faster than the others as the degree of locality increases.

The presence of glitches explains the worse performance of the the causal consistency protocol with a very low degree of locality (<0.2). When a glitch occurs, an intermediate node in a dependency graph is reevaluated (and thus triggers new change notifications) every time it receives a partial update. Conversely, protocols for glitch freedom that accumulate partial changes before triggering a new reevaluation are more efficient in this case.

Size of Reactive Graphs. Figure 6 shows the performance of DREAM when changing the size of the dependency graphs, i.e., the number of (observable and reactive) objects included in each graph. As expected, the propagation delay increases with the size of the graph (Figure 6a) because, on average, more steps are required to propagate the changes to all the interested reactive objects. The overall network traffic also increases with the size of the graph. Interestingly, the size of the graph has a more significant impact for the causal consistency protocol, since is produces more glitches and consequently more (redundant) expression evaluations.

Size of Reactive Expressions. Figure 7 shows the impact of the size of reactive expressions (i.e., the number of observable objects appearing in each expression) on the performance of DREAM. Increasing the number of objects in each expression (with a fixed degree of locality) increases

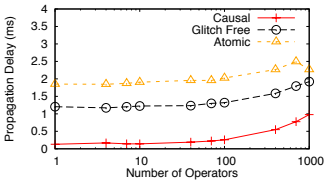


Figure 8: Efficiency of DREAM.

the number of notifications from remote objects that are required to evaluate the expression. For this reason, the delay increases for all protocols (Figure 7a). However, the delay increases faster with protocols for glitch freedom and atomicity, which need to receive *all* the notifications before processing them. Conversely, the causal protocol always processes (local) notifications immediately, without waiting for remote objects. The same considerations explain why increasing the number of objects in each expression also increases the overall network traffic (Figure 7b). In this case, the causal protocol is more affected: indeed, a higher number of dependencies between objects reflects in a higher number of potential glitches, and consequently a higher number of partial updates (not generated by the glitch free and the atomic protocols). However, the number of potential glitches in the graph stabilizes after 3 objects, and the traffic does not increase further.

6.2 Efficiency of DREAM

In the previous section, we evaluated DREAM with a network emulator to easily control a number of parameters. Since the network emulator did not consider the processing times (both at the clients and at the brokers), all results we collected are valid under the assumption that the processing time is negligible with respect to network latency (i.e., processing does not the bottleneck). We now validate this assumption, assessing the performance of DREAM.

To isolate the costs of event processing, we consider a local scenario, where a component `C` and a broker `B` are deployed as two separate Java processes on the same machine[4]. The component defines an observable object `obs` of type double and a reactive object `r` that depends on `obs`. In the test, we increased the complexity of the expression that defines `r`, introducing an increasing number of (arithmetic) operators applied to `obs`. Figure 8 shows the delay introduced by the DREAM middleware to propagate changes from `obs` and to recompute the expression that defines `r`.

In the case of causal consistency, the processing is local to the component: each change to `obs` generates an event that is passed to `r` using local method invocation and triggers the recomputation of the reactive expression. This experiment evaluates the cost of intercepting a method call, propagate the change, and reevaluate the expression using the code generated by ANTLR. We measure a delay that is lower than 0.15 ms with up to 10 operators and lower than 1 ms even in the extreme case of 1000 operators[5].

In the case of glitch free consistency, the propagation involves the broker `B`. In this case, the time we measure includes two communication steps (from `C` to `B` and back), which require serialization and deserialization of every event and processing at `B`. This process introduces a modest additional delay of about 1 ms with respect to causal consistency. Finally, the atomic consistency protocol introduces an additional communication step to contact the `TGS`. Based on these results, we can confirm the assumption of the previous section.

6.3 Discussion and Outlook

To the best of our knowledge, this work is the first attempt to investigate different consistency guarantees for DRP . Our evaluation demonstrates the validity of the major design choices of DREAM and suggests guidelines for future research.

First, our analysis shows the benefits of a distributed infrastructure for the propagation of changes, both in terms of network traffic and propagation delay. As expected, this advantage increases in scenarios where data accesses exhibit forms of locality. Since there is no single point of failure, a decentralized architecture provides an optimal starting point to explore fault-tolerant solutions. Currently, DREAM only supports caching though a globally accessible registry in case a value cannot be retrieved from a node. Future research should focus on how to keep the reactive network operational even when certain nodes are temporarily unresponsive.

Second, we observe that our protocol for causal consistency is usually more efficient than the others, since it enables components to autonomously dispatch local changes. On the other hand, it can potentially produce more traffic and demand for a higher number of expression evaluations due to the presence of glitches. This circumstance demands for an infrastructure that (*i*) supports different consistency protocols inside (different parts of) the same reactive system and (*ii*) can switch among them to optimize performance.

It is worth notice that the glitch free protocol could be optimized to process local changes in a component, without contacting external brokers – in case this cannot introduce glitches. This would require, however, a complete knowledge of the dependency graphs at each component. For this reason, we decided not to implement this solution in DREAM.

Finally, providing atomic consistency does not significantly impact on the overhead of DREAM when the rate of changes remains low. However, it prevents from running multiple propagations concurrently: because of this, in presence of high rate of changes, notifications would be queued at the `TGS`, with a negative impact on propagation delay[6].

7. RELATED WORK

Given the cross-field nature of our research, we organize related work in (*i*) reactive programming and (*ii*) event-based systems.

Reactive programming. Reactive programming refers to language abstractions and techniques to model and support *time changing* values. It was originally proposed in the context of strictly functional languages. Specifically, functional reactive programming (FRP) was first introduced in Haskell to support interactive animations [11].

Recently, reactive programming has gained increasing attention, and several implementations have been developed for various programming languages. We report only the

[4]The experiment is run on Core i7-3615QM @2.3 GHz with 8 GB of Ram and Java 7 on Mac OS 10.9.1.

[5]In case of more computationally intensive operators in expressions, this time can clearly increase. We plan to investigate incremental evaluation for such cases as future work, as discussed in [28].

[6]We did not include experiments showing this phenomenon for space reasons.

most prominent examples; the interested reader can find a recent and detailed survey on reactive programming in [1].

In the context of functional languages, Fran [11, 12] and Yampa [18] extend Haskell with two first class concepts to express time-varying values, *behaviors* for modeling continuous values and *events* for modeling discrete changes. Similar to DREAM reactive objects, behaviors can be combined and various Haskell operators were adapted to work with behaviors. Analogous extensions have been proposed for Scheme (see FrTime [5]) and Scala (Scala.React [25], REScala [36]). In contrast to DREAM, these solutions target only the local setting. Interestingly, in [12] the authors propose hybrid push/pull-based propagation of changes: the former reevaluates a reactive expression as soon as a change is detected, making the new updated value always available; the latter postpones the evaluation until it is strictly required, potentially reducing the computational effort. DREAM implements a pure push-based propagation model: we plan to investigate the feasibility and benefits of pull-based propagation in future work.

Similar to DREAM, Frappe [7] is a reactive programming library for Java. It extends the JavaBeans component model [4] introducing event sources (similar to DREAM observable objects) and behaviors (similar to DREAM reactive objects). Despite these similarities, Frappe does offer distributed deployment.

In the context of Web programming, Flapjax [30] has been proposed as a reactive programming library for JavaScript. The design of Flapjax is mostly based on FrTime. Flapjax supports distribution to enable client/server communication, but, as acknowledged by the authors, glitches are not avoided in distributed reactive applications.

AmbientTalk/R [23] is a reactive extension to AmbientTalk , an actor-based language for mobile applications. Similarly to Flapjax, AmbientTalk/R supports distributed reactive programming but does not ensure consistency guarantees or glitch avoidance.

The need for DRP with consistency guarantees has been widely recognized in literature [28, 1, 35]. Similarly, the integration of reactive and object-oriented programming have been discussed in [37]. In future work, we plan to explore this integration in more detail, focusing on solutions for automated tracking of dependencies, better support for inheritance, and efficient strategies for recomputation of values.

Reactive programming has been influenced by other paradigms based on dataflow and synchronous propagation of change. Synchronous programming [3] is one of the earliest paradigms proposed for the development of reactive systems. It is based on the synchrony assumption, i.e., it assumes reactions to be instantaneous and atomic. This assumption simplifies the program, which can be compiled into finite-state automata. Dataflow programming [39] represents programs as directed graphs, where nodes represent computations and arcs are dependencies between them.

Event-based systems. Event-based systems [31] define a model for dealing with reactive applications that is complementary to the one described in this paper. In reactive programming, events (notifications of changes) are implicit and reactions (recomputation of reactive variables) are declaratively specified. Conversely, in event-based systems events are explicitly notified using call-back functions and the programmer can imperatively specify custom reactions.

DREAM currently relies on a publish-subscribe infrastructure [15] for the propagation of events. In addition to event forwarding, Complex Event Processing (CEP) systems [9, 24, 13] provide operators for event *composition*. For example, they enable to detect (temporal) patterns or apply functions over all the events received in a given time window. We plan to exploit these features to extend the expressiveness of DREAM. The interested reader can refer to our paper [28], where we compare reactive programming and CEP systems and provide a roadmap for integrating them.

Concerning consistency guarantees, [40] discusses ordering guarantees for publish-subscribe sytems. Similarly, [2] presents an analysis of consistency in the domain of CEP.

In the context of event-based systems, it is worth mentioning that several libraries and language extensions have been proposed to support events and event composition as first class language constructs. The most notable examples are EventJava [14], Ptolemy [33], EScala [17] and JEScala [38].

Finally, Rx [29] is a library originally developed for .NET and recently ported to various platforms, including Java. Rx has received great attention since it has been successfully adopted in large-scale projects, including content distribution in the Netflix streaming media provider. Rx shares many similarities with event-based languages, including abstractions for event composition based on LINQ. The programming model of Rx is slightly different from the one offered in FRP and in DREAM. In particular, Rx provides a generic interface for event sources; the programmer can subscribe to an event source by registering a closure that gets executed when an event occurs. Even if Rx supports distributed programming, it does neither ensure consistency guarantees nor glitch freedom.

8. CONCLUSIONS

The reactive programming paradigm has been proposed to ease the development of modern reactive systems. Nevertheless, distributed reactive programming (DRP) has received little attention so far. Most significantly, existing solutions for DRP do not provide precise semantics and guarantees for the propagation of changes. This paper addressed such problem by introducing an abstract model for DRP and by formalizing some desirable consistency guarantees. Starting from this analysis, we defined three propagation semantics, we implemented them in the DREAM middleware, and we provided an extensive evaluations of their costs.

In future work, we will target both the expressiveness and the efficiency of DREAM. On the one hand, we want to investigate more complex reactive expressions (e.g., to capture time series and sequences of changes). On the other hand, we want to explore different strategies for evaluating expressions (e.g., incremental evaluation, lazy evaluation). Finally, we plan to assess the benefits of DREAM in real applications.

We hope that the content of this paper, and in particular our analysis of the consistency guarantees and our evaluation of their costs, will represent the base for future research efforts, to provide flexible and powerful abstractions and ease the development of distributed reactive systems.

Acknowledgment

This research has been funded by the Dutch national program COMMIT, by the European Research Council, grant No. 321217 “PACE” and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 01IC12S01V.

9. REFERENCES

[1] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 2012.

[2] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Procs. of the IEEE*, 91(1):64–83, 2003.

[4] D. Blevins. Overview of the enterprise JavaBeans component model. In *Component-based software engineering*, pages 589–606. Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *PEPM*, pages 71–80, 2007.

[6] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[7] A. Courtney. Frappe: Functional reactive programming in Java. In *PADL*, pages 29–44, 2001.

[8] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Soft. Eng.*, 27(9):827–850, 2001.

[9] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):1–62, 2012.

[10] G. Cugola and G. P. Picco. REDS: a reconfigurable dispatching system. In *SEM*, pages 9–16, 2006.

[11] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[12] C. M. Elliott. Push-pull functional reactive programming. In *Haskell*, pages 25–36, 2009.

[13] O. Etzion and P. Niblett. *Event processing in action*. Manning Publications Co., 2010.

[14] P. Eugster and K. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP*, pages 570–594, 2009.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[16] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer. Protopeer: From simulation to live deployment in one step. In *P2P*, pages 191–192, 2008.

[17] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. In *AOSD*, pages 227–240, 2011.

[18] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[19] X. Jia. A total ordering multicast protocol using propagation trees. *IEEE Trans. on Parallel Distrib, Syst.*, 6(6):617–627, 1995.

[20] R. Johnson, R. Helm, J. Vlissides, and E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[23] A. Lombide Carreton, S. Mostinckx, T. Cutsem, and W. Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Objects, Models, Components, Patterns*, volume 6141, pages 41–60, 2010.

[24] D. C. Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[25] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.

[26] A. Margara and G. Cugola. High Performance Content-Based Matching Using GPUs. In *DEBS*, pages 183–194, 2011.

[27] A. Margara and G. Cugola. High performance publish-subscribe matching using parallel hardware. *IEEE Trans. on Parallel Distrib. Syst.*, 2014.

[28] A. Margara and G. Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. In *REM*, 2013.

[29] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[30] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *OOPSLA*, pages 1–20, 2009.

[31] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed event-based systems*. Springer, Heidelberg, 2006.

[32] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[33] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[34] M. Sadoghi, H. Singh, and H.-A. Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *DaMoN*, pages 27–32, 2011.

[35] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In *Coordination Models and Languages*, pages 226–235. Springer, 2013.

[36] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *AOSD*, 2014.

[37] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *AOSD*, 2013.

[38] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular coordination with declarative events and joins. In *AOSD '14*, 2014.

[39] P. G. Whiting and R. S. Pascoe. A history of data-flow languages. *Annals of the History of Computing*, 16(4):38–59, 1994.

[40] K. Zhang, V. Muthusamy, and H.-A. Jacobsen. Total order in content-based publish/subscribe systems. In *ICDCS*, pages 335–344, 2012.