On the Semantics of Distributed Reactive Programming: the Cost of Consistency

Alessandro Margara, Guido Salvaneschi

Abstract—
The reactive programming paradigm aims to simplify the development of reactive systems. It provides abstractions to define time-changing values that are automatically updated by the runtime according to their dependencies. The benefits of reactive programming in distributed settings have been recognized for long. Yet, existing solutions for distributed reactive programming enforce the same semantics as in single processes, introducing communication and synchronization costs that hamper scalability.
Establishing suitable abstractions for distributed reactive programming demands for a deeper investigation of the semantics of change propagation. This paper takes a foundational approach and defines precise propagation semantics in terms of consistency guarantees that constrain the order and isolation of value updates. We study the benefits and costs of these consistency guarantees both theoretically and empirically, using case studies and synthetic benchmarks.
We show that different applications require different levels of consistency and that manually implementing the required level on a middleware that provides a lower one annuls the abstraction improvements of reactive programming. This motivates a framework that enables the developers to select the best trade-off between consistency and overhead for the problem at hand. To this end, we present DREAM, a distributed reactive programming middleware with flexible consistency guarantees.

Index Terms—Distributed Reactive Programming, Consistency Guarantees, Reactive Programming Middleware, DREAM

1 Introduction

Many modern software systems are *reactive*: they respond to the occurrence of events of interest by updating values and performing computations that 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. Reactive code is asynchronously triggered by event occurrences, which makes it hard to trace and understand the control flow of the entire system [1] and opens room for subtle errors. In fact, in 2008 a half of the bugs reported for Adobe's desktop applications was generated in code for event handling [1].

The *reactive programming* (RP) paradigm [2] has been proposed to mitigate these issues and simplify the development of reactive systems. RP aims to make the propagation of events and the management of updates implicit by building on three key concepts: time-changing values, tracking of dependencies, and automated propagation of changes. To explain the core principles of RP, let us consider the following pseudocode snippets that define a variable a and a variable b that depends on a:

- *Alessandro Margara is with the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) at Politecnico di Milano, Milan, Italy.*
  E-mail: alessandro.margara@polimi.it
- *G. Salvaneschi is with the Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany.*
  E-mail: salvaneschi@cs.tu-darmstadt.de

```
a: int = 10            a: int = 10
b: int = a + 2         b: int := a + 2
print(b) // 12         print(b) // 12
a = 11                 a = 11
print(b) // 12         print(b) // 13
```

In conventional imperative programming (left), any future change to the value of a does not impact on the value of b. In RP (right), the second line defines a *constraint* (denoted by :=) rather than a statement, thus ensuring 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 Observer design pattern adopted in traditional event-based architectures [3]. In particular, developers do not need to *program* the update logic. Instead, they *declare* the dependencies between variables and entirely delegate the update process to the runtime. This results in more compact and readable code, and reduces the possibility of subtle errors [4], [5]. Most significantly, the runtime takes care of ensuring the *correctness* of the propagation. For instance, it can enforce certain order guarantees in the propagation of changes to avoid the occurrence of *glitches* – temporary violations of data flow invariants [6].

The need to extend the benefits of RP to the distributed setting is well recognized [7], [2], [8], [9], [10] as many reactive applications, such as Web applications, monitoring systems, and mobile apps, IoT software, are distributed. Yet, most existing RP frameworks do not support distribution in the sense that they only propagate changes inside each single process, while cross-process changes must be propagated manually. As a result, the benefits of RP are exploited only half way: software design of each process improves, but dropping out of the principled propagation guaranteed by RP at the process boundaries comes at the cost of relinquishing consistency properties otherwise enforced by RP [2].

Conversely, the few frameworks that support RP in a distributed setting typically incur high costs for change propagation since they rigidly try to impose the same semantics as in single process RP. For instance, they typically consider each propagation as an atomic operation, introducing high communication and synchronization costs that limit scalability.

We argue that different applications demand for different semantics of propagation, with different trade-offs between the consistency properties offered and the cost to implement them. On the one hand, a framework that provides a strong consistency might impose costs that are prohibitive for certain applications. On the other hand, as we show in this work, manually enforcing some desired consistency properties on top of a framework that does not offer them is a complex and error prone task that nullifies the design advantages RP is adopted for.

In this paper, we define p recise s emantics a nd desired properties for distributed reactive programming (DRP), identifying a number of *consistency guarantees for DRP* that constrain the order and the interleaving of value updates in a reactive application. We study the benefits and costs of the proposed consistency guarantees both theoretically and empirically, using case studies as well as synthetic benchmarks. Finally, we present DREAM (Distributed REActive Middleware), a DRP framework that offers multiple levels of consistency guarantees. When considering the same propagation semantics, DREAM outperforms the efficiency of state-of-the-art solutions.

In summary, this work makes the following contributions:

- We define multiple consistency guarantees for the propagation of changes in DRP and describe how to implement them;
- We analytically inspect the benefits and the overhead to ensure the proposed consistency guarantees;
- We propose novel algorithms for the propagation of changes in a distributed setting that outperform the state-of-the-art solutions in terms of efficiency;
- We show how the propagation algorithms can be efficiently implemented on top of a distributed publish-subscribe middleware;
- We present the design and implementation of the DREAM framework, which offers DRP with flexible consistency guarantees for Java programs;
- We empirically evaluate the benefits and overhead of the proposed consistency guarantees using both case studies and synthetic workloads.

This work is based on our previous paper on the design and implementation of DREAM [11]. DREAM has been entirely reimplemented based on native Java code and supports the newly defined consistency guarantees. In addition, this paper (i) provides a new formal definition for the consistency guarantees; (ii) introduces new levels of consistency guarantees; (iii) discusses the implementation strategies to enforce the proposed consistency guarantees and their cost; (iv) presents various case studies that we concretely implemented on top of DREAM; (v) presents a wide evaluation campaign that empirically studies the overhead of the proposed consistency guarantees in a real distributed deployment on Amazon EC2 and in simulation; (vi) compares the propagation algorithms implemented in DREAM with Distributed REScala, a state-of-the-art propagation algorithm for DRP with fixed and predefined consistency guarantees [9].

The rest of the paper is organized as follows. Section 2 discusses the motivations behind our work. Section 3 presents a model for DRP and introduces various consistency guarantees. Section 4 discusses their technical realization and cost. Section 5 introduces the design and implementation of DREAM. Section 6 evaluates DREAM with benchmarks and Section 7 presents a qualitative evaluation based on case studies. Section 8 discusses the assumptions underneath our work and outlines possible solutions to relax them. Section 9 surveys related work and Section 10 concludes.

2 Background and Motivation

In this section, we introduce examples of distributed reactive applications that can benefit from an implementation based on RP, and we overview their requirements. In doing so, we show how different applications demand for different propagation semantics.

Navigation application

We first consider a navigation application as in Fig. 1. Variable n_1 stores the current position of a GPS sensor and variable n_2 is defined on a remote display that shows the current position of the GPS on a map. Variable n_2 is defined through the constraint $n_2 := n_1$, meaning that whenever the value of n_1 changes, the value of n_2 should change accordingly. As explained in Section 1, the runtime is responsible for the propagation of changes.

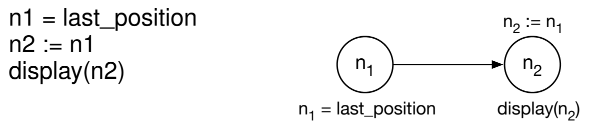


Fig. 1: Navigation application.

In this scenario, the developer expects that the order of the updates to variable n_2 matches the order of updates to variable n_1, such that the remote display always shows positions that match the real trajectory of the GPS. If the runtime does not enforce this property, n_2 might receive updates in any order and the position on the remote display may not correspond to the trajectory of the GPS.
We address this requirement with FIFO consistency, as defined in Section 3.4.1.

AI engine

Fig. 2 models the components of a game implemented with RP. Variable n holds the current move of the player. The value of n is propagated to an AI module, which computes variable n_1 – the reaction of the AI to the player move. Finally, a display module combines the player move and the AI reaction in variable n_2 and displays them on screen.

```
n = player_move
n1 := AI_react(n)
n2 := (n, n1)
display(n2)
```

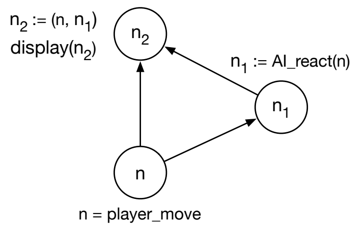


Fig. 2: AI engine.

In this application, the developer might desire that n_2 reflects the new move from the player *before* the corresponding reaction from the AI. Indeed, the reaction *is caused* by the player's move, and displaying the reaction first would represent an undesired behavior.
We address this requirement with causal consistency, as defined in Section 3.4.2.

Financial application

Fig. 3 presents a financial application, where variable s reflects the latest values from the financial market, f_1 and f_2 compute forecasts using two different models, and a averages the contribution of the two models. If the average of the two models is greater than 10, the application automatically performs a sell operation.

```
s = marketIndex
f1 := model1(s)
f2 := model2(s)
a := (f1 + f2) / 2
if (a > 10) sell()
```

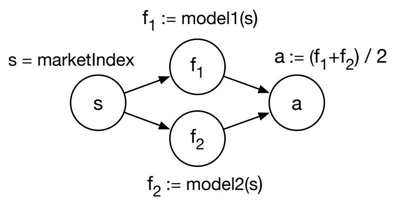


Fig. 3: Financial application.

Now assume that an update to s triggers a recomputation of f_1 and f_2: f_1 was 4 and becomes 6, while f_2 was 5 and becomes 4. If the new value of f_1 is propagated and a is recomputed taking into account the new value of f_1 but the *old* value of f_2, then a becomes 6+5=11 and triggers a sell operation. Even if a is later updated with the correct value of f_2, the sell operation cannot be undone. In this application, this is an undesired behavior since the developer expects a to reflect the latest changes in *both* models.
We address this requirement with single-source glitch freedom, as defined in Section 3.4.3.

Online game

Fig. 4 models an online game. Variables s_1 and s_2 are the actions of two players. Variables n_1 and n_2 collect the actions of the two players and display them from two different perspectives.

Depending on how the propagation of changes in s_1 and s_2 interleave, n_1 and n_2 might process the changes coming from s_1 and s_2 in different orders and the two

```
s1 = ...
s2 = ...
n1 := (s1, s2)
display(n1)
n2 := (s1, s2)
display(n2)
```

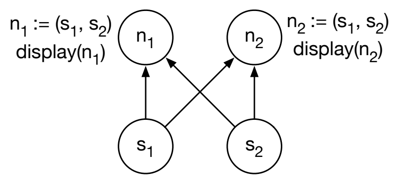


Fig. 4: Online game.

displays might show inconsistent views of the game, which is undesirable for this application.
We address this requirement with complete glitch freedom, as defined in Section 3.4.4.

Load balancer

Fig. 5 shows the architecture of a replicated Web server with a load balancer. Variables n_1 and n_2 hold the percentage of requests the load balancer redirects to server1 and server2, respectively. Variable s controls the balancing strategy holding the fraction of requests for server1. An actuator component enforces the balancing policy by sampling variables n_1 and n_2.

```
s = ...
n1 := s
n2 := 1 − s
```

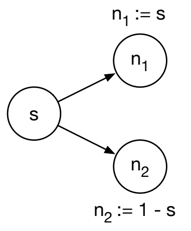


Fig. 5: Load balancer.

In case of an update of s from 0.5 to 0.6, the change is propagated to n_1 and n_2, which become 0.6 and 0.4. The developer desires that the actuator cannot read the new value of n_1, 0.6, *and* the old value of n_2, 0.5, which do not sum up to 100.
We address this requirement with atomic consistency, as defined in Section 3.4.5.

Executive Summary

The previous examples highlight that there is no *one size fits all* definition of correctness in DRP. RP frameworks need to offer different degrees of consistency and developers should be able to select the best trade-off between performance and consistency for their application. This is especially true in presence of distribution, where ensuring consistency may introduce a significant overhead, for instance, to obtain a global agreement on the order of updates.

To satisfy these needs, in the remainder of this paper we precisely define different consistency guarantees and we study the cost for ensuring them in a distributed environment.

3 Model and Consistency Guarantees

This section introduces a model for DRP and the notation that we use in the rest of the paper. Then, it precisely defines the consistency guarantees that satisfy the requirements in Section 2. Table 1 provides a summary of our notation.

G	Reactive graph
$n \in N$	Graph node, Var or Signal
e	Graph edge
$\rightarrow$	Dependency relation
$\xrightarrow{*}$	Transitive closure of $\rightarrow$
$S \subseteq N$	Sources in the graph
$N_s \subset N$	Nodes affected by a change to a source $s \in S$
$F_s \in N_s$	Sinks in the graph
E_s	Edges involved in a propagation from source s
$p_{(s,n)}$	Paths from s to any node $n \in N_s$
$pc_{(s,n)}$	Cardinality of paths $p_{(s,n)}$
E^i	Internal edges connecting nodes in the same process
E^e	External edges connecting nodes in different processes
E^i_s	Internal edges in E_s
E^e_s	External edges in E_s
c	Traffic cost for sending a message
c_{lock}	Cost for locking or releasing a set of resources
$l_{(i,j)}$	Latency for propagating a message across edge (i,j)
l_p	Latency of a path p

TABLE 1: Notation used in the paper.

3.1 Basic Definitions

We assume the program logic to be distributed across multiple *processes*, deployed on one or more hosts. We adopt two fundamental RP abstractions: *signals* and *vars*. Each var and signal is defined within a single process.

Vars are sources of changes that produce a stream of values over time and can be used to build reactive expressions. For instance, in the financial application of Fig. 3, s is a var used in the definition of f_1 and f_2. A signal is defined via a reactive expression e and gets automatically updated: whenever the value of one of the vars or signals that appear in e changes, e is recomputed to determine the new value of the signal. In the financial application scenario of Fig. 3, signal f_1 is defined by the reactive expression $model1(s)$. When s changes, the value of f_1 gets automatically updated by recoputing $model1(s)$. Within an application running on a process p, developers can read the current value of any subset of the signals defined in p. They are not allowed to directly change the value of a signal or the expression that defines that signal. A fundamental feature for composability is that signals can be used into reactive expressions to create new signals. For example, in the financial application of Fig. 3, signal a is defined from signal f_1 and signal f_2.

3.2 System Model

We model a reactive application using a directed acyclic graph $G = (N, E)$ called *dependency graph*, where a node $n \in N$ represents a var or a signal and an edge $e = (n_1, n_2) \in E$ indicates that the value of n_2 depends on the value of n_1 according to the dependency relation $\rightarrow$. We call $e = (n_1, n_2)$ an *internal* edge if n_1 and n_2 are defined in the same process, *external* otherwise. We denote with $\xrightarrow{*}$ the transitive closure of $\rightarrow$: $n_1 \xrightarrow{*} n_2$ iff there is a (possibly zero-length) path in the graph from n_1 to n_2. $S \subseteq N$ is the set of vars – also referred to as *sources* of changes in the rest.

Our model assumes liveness: when the value of a source s changes, the change (also called update) is propagated to all the nodes n such that $s \rightarrow n$ and their value is recomputed based on the new value of s. The update process is recursive: an update u_i on a node n_i triggers an update u_j of the value of any node n_j such that $n_i \rightarrow n_j$. We say that there is a *causal relation* between u_i and u_j, or that u_i (directly) causes u_j. Since the graph is acyclic, the propagation of updates is guaranteed to terminate. We further refer to the transitive closure of the causal relation between updates by saying that an update u_i directly or indirectly causes update u_j.

The *dependency graph* abstracts over the real architecture where signals and vars are located on different processes – in the same or in different hosts – and exchange updates through a *communication infrastructure*.

During the execution, the program running on a process p can read any subset of vars and signals that are located within that process. We say that a read operation on a node n_j observes the effects of an update u_i on a node n_i, $n_i \xrightarrow{*} n_j$, if the read value is computed by taking into account the update u_i.

Our model assumes that the communication infrastructure is reliable, meaning that it delivers each message exactly once. Processes can fail: if a process p fails, all the nodes stored on p are removed from the dependency graph and other processes are notified about the failure. More complex mechanisms for fault tolerance in DRP are out of the scope of this work and are object of current research [12], [13]. We discuss in Section 8 possible strategies to integrate fault tolerance in our DRP design.

A naive propagation algorithm that follows the schema above only guarantees that in absence of failures an update from a source s is eventually propagated to all nodes that directly or indirectly depend on s (due to the liveness assumption in our model). Based on the application scenario, stronger guarantees are desirable (Section 3.4).

3.3 Notation for the Cost Analysis

We consider two costs: (i) *traffic*, the number of messages required to propagate a change and (ii) *latency* to receive an update. In this section, we introduce the notation required for cost analysis.

We call $N_s \subset N$ the set of nodes that are directly or indirectly affected by a change to a source $s \in S$:

$$N_s = \{n \in N \mid s \xrightarrow{*} n\}$$

We call $F_s \in N_s$ the set of *sinks* for s, that is to say, the nodes in N_s that do not have any outgoing edge:

$$F_s = \{n \in N_s \mid \forall i \in N \nexists (n, i) \in E\}$$

We call E_s the set of edges involved in a propagation from source s:

$$E_s = \{(i, j) \in E \mid i \in N_s,\ j \in N_s\}$$

We call $p_{(s,n)}$ the paths from s to any node $n \in N_s$, and $pc_{(s,n)} = |p_{(s,n)}|$ the number of such paths. We call E^i the set of internal edges that connect nodes belonging to the same process, and E^e the set of external edges that connect nodes belonging to different processes. $E^i_s = E_s \cap E^i$ is the set of internal edges in E_s and $E^e_s = E_s \cap E^e$ is the set of external edges in E_s. Propagating a change across an external edge involves sending a message through the communication infrastructure. We denote c the traffic cost to send such message. Change propagation across internal edges has zero traffic cost – it uses only local invocations.

Since propagation may involve coordination among processes to guarantee global order or mutual exclusion, DREAM enforces a node locking discipline. We denote c_{lock} the cost for locking or releasing a set of resources.

Finally, we call $l_{(i,j)}$ the latency for propagating a message across the edge (i, j). The latency l_p of a path p is the sum of the delays of all the edges in p.

3.4 Consistency Guarantees

Using the notation presented above, we now introduce different definitions of correctness for change propagation in DRP, starting from the ones that offer weaker consistency guarantees to the ones that offer stronger consistency guarantees, but higher performance overhead.

We assume that each update to a node n_1 is eventually propagated to all the nodes n_2 such that $n_1 \to n_2$ and we define the consistency guarantees based on the values that a process can observe when reading from one or more nodes. Whenever possible, we relate the consistency levels to classic ones in the field of data replication and database transactions [14], [15]. Single-node RP systems typically identify correctness with glitch freedom [1], [16], [17]. Our levels include two forms of glitch freedom – single-source and complete – as well as weaker and stronger levels.

3.4.1 FIFO Consistency

For any two nodes n_1 and n_2 such that n_2 depends on n_1, $n_1 \overset{*}{\Rightarrow} n_2$, and for any two updates u_1 and u_2 to n_1 such that u_1 occurs before u_2, a propagation algorithm is FIFO consistent if a reader of n_2 cannot see the effects of u_2 before the effects of u_1.

Intuitively, FIFO consistency ensures that a signal reflects the changes to a single variable in the order in which they occur. Thus, FIFO consistency satisfies the requirement of the navigation application presented in Fig. 1.

3.4.2 Causal Consistency

A propagation algorithm is causally consistent if it is FIFO consistent and, for any two updates u_1 and u_2 such that u_1 causes u_2, a reader of a node n cannot observe the effects of u_2 on n before the effects of u_1 on n (if any). Causal consistency satisfies the requirements of the AI engine presented in Fig. 2.

This consistency level is analogous to causal consistency in replicated data stores, which guarantees that operations that are causally related with each other take place in the same order in all the replicas, and this order reflects the causal dependency [14].

3.4.3 Single-source Glitch Freedom

Given two nodes n_1 and n_2 such that n_2 depends on n_1, $n_1 \overset{*}{\Rightarrow} n_2$, and given an update u to n_1, a propagation algorithm provides single-source glitch freedom if it is FIFO consistent and further ensures that the value returned by a read on a node n_2 reflects *all* the effects on u on n_2 or *none* of them. Intuitively, under single-source glitch freedom all the effects caused by an update on a node become visible at the same time. This means that a reader cannot observe the effects of an update on a node in an order that violates causality. As a consequence single-source glitch freedom implies causal consistency.

Single-source glitch freedom satisfies the requirements of the financial application presented in Fig. 3, ensuring that any recomputation of a triggered by an update of s considers *both* the new value of f_1 and the new value of f_2.

3.4.4 Complete Glitch Freedom

FIFO, causal, and single-source glitch freedom focus on the effects of a single update on a single node. Complete glitch freedom targets the interleaving of the effects of multiple updates on multiple nodes.

Given two nodes n_1 and n_2 and two updates u_1 and u_2, a propagation algorithm provides complete glitch freedom if it provides single-source glitch freedom and further ensures that any read that involves both n_1 and n_2 observes the effects of u_1 and u_2 on n_1 in the same order as the effects of u_1 and u_2 on n_2. Intuitively, complete glitch freedom ensures that the results of two propagations are the same as if the propagations took place in some sequential order, without interleaving at any node. Thus, complete glitch freedom satisfies the requirements of the online game presented in Fig. 4.

Complete glitch freedom shares some analogy with sequential consistency in replicated data stores and serializable isolation of transactions [15]. Sequential consistency ensures that the results of reading from any replica are the same as if all the updates to the data store were executed in some sequential order. Similarly, serializable isolation ensures that the results of transactions are the same as if they were executed in some sequential order.

3.4.5 Atomic Consistency

A propagation algorithm provides atomic consistency if it provides complete glitch freedom and further ensures that, for any update u to a node n and for any pair of nodes n_1 and n_2 such that $n \overset{*}{\to} n_1$, $n \overset{*}{\to} n_2$, a read operation that involves both n_1 and n_2 observes the effects of u on both n_1 and n_2, or on none of them.

Intuitively, atomic consistency ensures that a read operation cannot observe only *some* of the effects of the change in a source: either it observes all of them, or none of them. Thus, it satisfies the requirements of the load balancer presented in Fig. 5.

4 Consistency Realization and Cost

This section discusses the implementation of the consistency levels introduced in Section 3.4 and their cost. We adopt the DRP model from Section 3.2: nodes in the dependency graph are separate processes that propagate updates. Updates are implemented as messages delivered through the communication infrastructure. Nodes process each message atomically and have a complete view of the graph topology.

A message `m` holds a value (`m.val`), an identifier of the sender node (`m.sender`), an identifier of the var – the source – that triggered `m` (`m.source`), and a timestamp (`m.ts`). Timestamps are logical vector clocks [18], [19] used to reorder messages based on consistency requirements. Each timestamp stores a counter c_n for each node n in the dependency graph. When a node sends a new message, it increments its own counter and attaches it to the message. It also attaches a counter for all nodes it transitively depends on. For each node n, the counter is the maximum among the counters c_n stored in the messages the node has received and already successfully processed.

```
class Node {
  self
  currentVal
  children[]
  propagate(source, ts) {
    for child in children
      send(child, new msg{currentVal, self, source, ts})
  }
}
class Var extends Node {
  counter = 0
  modify(val) {
    waitModify()
    currentVal = val
    ts = [0, .., 0]
    ts.self = counter
    counter = counter + 1
    propagate(self, ts)
  }
}
class Signal extends Node {
  val[]
  lastTS[]
  compute() { ... }
  receive(m) { ... }
}
read(nodes) {
  waitRead(nodes)
  return nodes.values
}
```

Listing 1: Algorithm for the consistency guarantees.

Listing 1 shows the algorithm used by the nodes to propagate changes. Each node – class `Node` – stores a unique node identifier `self`, its current value `currentVal`, and the identifiers `children` of all its children in the dependency graph, that is, all the nodes that depend on it. In the `propagate()` function, the node propagates its value to all children sending a message with the current value, the node identifier as sender, the original source of the change, and the timestamp discussed above.

A `Var` extends a `Node` and implements a function `modify()` to update its value. The function propagates the current value using the identifier of the `Var` as the source of message and a timestamp where all the counters are set to zero except the counter of the `Var`, which is increased at each propagation. Similarly, a `Signal` extends `Node` and implements a `receive()` function to receive update messages. We assume that each signal stores a value for each of the nodes `d` it directly depends on, that we call `val[d]`. With these values, function `compute()` recomputes the reactive expression of the signal `currentVal`. Each signal stores in `lastTS[d]` the timestamp of the last message sent by `d` that it has successfully processed. Finally, external components can invoke the utility function `read()` to retrieve the current value of the set of input nodes.

The concrete propagation algorithm depends on the implementation of the `receive()`, `waitModify()`, and `waitRead()` functions, which define the behavior upon receiving a message, trying to modify the value of a `Var`, and trying to read the value of one or more nodes. Table 2 shows such functions for different consistency levels.

4.1 FIFO and Causal Consistency

Under FIFO and causal consistency, nodes use the timestamps to enforce constraints on the order of evaluation of messages in the `receive()` function.

In the case of FIFO consistency, the `receive(m)` function checks that all previous messages from the same sender have been received, referring only to the sender timestamp `m.ts[m.sender]` and comparing it with the timestamp of the last processed message `lastTS[m.sender]`. This guarantees that all the updates from a given node are processed in the order in which they occurred, thus satisfying the definition of FIFO consistency.

For causal consistency, the `receive(m)` function checks that all the messages that causally precede `m` have been received. Specifically, message `m'` causally precedes `m` if it caused the propagation of `m`, that is if there is a node `s` such that $\mathtt{s} \rightarrow \mathtt{self}$, $\mathtt{s} \xrightarrow{*} \mathtt{m.sender}$, $\mathtt{m'.sender} = \mathtt{s}$, and $\mathtt{m'.ts[s]} < \mathtt{m.ts[s]}$. Thus, the algorithm checks that for all nodes `s` that satisfy the properties above $\mathtt{lastTS[s]} \geq \mathtt{m.ts[s]}$. This ensures that updates are processed in causal order, thus satisfying the definition of causal consistency.

As Table 2 shows, both in the case of FIFO and causal consistency, if message `m` is received out of order, the `receive(m)` function stores it into the `store` variable for future evaluation. Otherwise, the function updates the value and timestamp of the sender, `val[m.sender]` and `lastTS[m.sender]`, using the values contained in `m`, `m.val` and `m.ts[m.sender]`, recomputes its own value `currentVal`, and propagates the new value to all the dependent nodes. Since the processing of `m` might enable the processing of previously received and stored messages, the `signal.receive()` function is invoked recursively.

FIFO and causal consistency do not impose a total order among the updates of different vars and do not enforce atomicity of propagation. As a consequence, the functions `waitModify()` and `waitRead()` immediately return.

The implementation of FIFO and causal consistency in Table 2 makes no assumptions on the behavior of the communication infrastructure. The protocol resembles classic approaches to causal consistency based on vector clocks [20] and applied in the context of distributed shared memory [21]. Alternative solutions can exploit properties of the communication infrastructure. For FIFO consistency, a connection between nodes implemented using direct TCP channels ensures reliable FIFO communication. If the communication infrastructure includes intermediate processors, FIFO communication is preserved if the processors preserve message order and use FIFO channels. Similarly, causal consistency is guaranteed by the communication infrastructure if (i) the messages between any two processes are delivered in FIFO order; (ii) there exists a single, acyclic path (sequence of links) between any two processes [22].

Cost Analysis

With FIFO and causal consistency each node n outputs one message on each of its outgoing edges in the dependency graph for each input message received. As a consequence, for each change to a source s, a node n receives an update message from any of the paths from s to $n - p_{(s,n)}$. Recalling that there are $pc_{(s,n)}$ paths from s to n, the overall number of messages delivered for each update to s is:

	FIFO	Causal	Single Source GF / Complete GF / Atomic

signal.receive(m)	**if** lastTS[m.sender] != m.ts[m.sender] $-$ 1 {
	  store = store ++ {m}
	} **else** {
	  val[m.sender] = m.val
	  currentVal = compute()
	  lastTS[m.sender] = m.ts[m.sender]
	  propagate(m.sender, lastTS)
	  for msg $\in$ store {
	    store = store $--$ msg
	    self.receive(msg)
	  }
	}
		if $\exists$ s, s $\rightarrow$ self $\wedge$ s $\xrightarrow{*}$ m.sender
		  $\wedge$ lastTS[s] $<$ m.ts[s] {
		  store = store ++ { m }
		} **else** {
		  val[m.sender] = m.val
		  currentVal = compute()
		  lastTS[m.sender] = m.ts[m.sender]
		  propagate(m.sender, lastTS)
		  for msg $\in$ store {
		    store = store $--$ msg
		    self.receive(msg)
		  }
		}
			// This algorithm assumes that
			// FIFO consistency is guaranteed
			if $\exists$ p, s, s $\xrightarrow{*}$ p $\wedge$ s $\xrightarrow{*}$ m.sender $\wedge$ p $\rightarrow$ self $\wedge$
			  $\nexists$ m′ $\in$ store,
			  m′.sender == p $\wedge$ m′.ts[s] == m.ts[s] {
			  store = store ++ {m}
			} **else** {
			  $\forall$ p, s, s $\xrightarrow{*}$ p $\wedge$ s $\xrightarrow{*}$ m.sender $\wedge$ p $\rightarrow$ self $\wedge$
			  $\forall$ m′ $\in$ store,
			  m′.sender == p $\wedge$ m′.ts[s] == m.ts[s] {
			    store = store $--$ m′
			    val[m′.sender] = m′.val
			  }
			  val[m.sender] = m.val
			  currentVal = compute()
			  propagate(m.sender, lastTS)
			}

	FIFO	Causal	Single Source GF	Complete GF	Atomic

var.waitModify()	**return**	**return**	**return**	writeLock({n | n $\in$ conflict(var)})
				
				where
				shared(v1, v2) = { n | v1 $\xrightarrow{*}$ n, v2 $\xrightarrow{*}$ n }
				conflict(v1, v2) = $\begin{cases} \emptyset & \text{if } \lvert\text{shared(v1, v2)}\rvert < 2 \\ \text{shared(v1, v2)} & \text{otherwise} \end{cases}$
				conflict(var) = $\bigcup_{\text{v} \in \text{vars}}$ conflict(v, var)
					writeLock({n | var $\xrightarrow{*}$ n})

waitRead(nodes)	**return**	**return**	**return**	**return**	readLock(nodes)

TABLE 2: Implementation of the `receive()`, `waitModify()`, and `waitRead()` functions for different consistency levels.

$$msgs = \sum_{(i,j) \in E_s} pc_{(s,i)}$$

The traffic cost is the cost for traversing the external edges E_s^e, which connect nodes in different processes:

$$c_{msgs} = \sum_{(i,j) \in E_s^e} c \cdot pc_{(s,i)}$$

A node $n \in N_s$ receives information about an update to a source s from any of the paths $p \in p_{(s,n)}$. The latency $l_{(s,n)}$ for receiving this information is the maximum latency over all such paths:

$$l_{(s,n)} = \max_{p \in p_{(s,n)}} l_p$$

The propagation terminates when it reaches all the sinks F_s. The overall propagation latency l is:

$$l = \max_{f \in F_s} l_{(s,f)}$$

In the rest of this section, we present the realization of stronger consistency guarantees assuming that messages are received in FIFO order.

4.2 Single-source Glitch Freedom

We implement single-source glitch freedom by updating the value of a node `n` only once for each update to a node `n` depends on. To do so, we temporarily buffer messages within nodes until we are ready to apply the changes they carry. As Table 2 shows, when a signal receives a message `m` – function `receive(m)` –, it stores the message and checks if it has received an update for the same change from all nodes it depends on. The signal checks if there exist some nodes `s` and `p`, such that `s` $\xrightarrow{*}$ `m.sender`, `s` $\xrightarrow{*}$ `p`. In this case, it checks that a message `m′` has been received from `p` with the same timestamp of `m` for `s`: `m.ts[s] == m′.ts[s]`. When all such messages are received, the function `signal.receive(m)` updates the values of the dependent nodes accordingly, recomputes the value of the signal, `currentVal`, and propagates it to dependent nodes.

Similar to FIFO and causal consistency, single-source glitch freedom does not constrain the order of accesses to vars and signals. Thus, the functions `waitModify()` and `waitRead()` shown in Table 2 immediately return.

Cost Analysis

In contrast to FIFO and causal consistency, single-source glitch freedom requires that a node n processes all messages originating from the same update of a source s in one step. Hence, n always propagates *a single* message in response to each update to a source s and the overall number of messages delivered for each update to a source s is:

$$msgs = |E_s|$$

The cost of this propagation is the cost for traversing the external edges E_s^e:

$$c_{msgs} = c \cdot |E_s^e|$$

The analysis shows that although single-source glitch freedom offering a higher level of consistency with respect to FIFO and causal consistency, it exhibits potential savings in terms of traffic, since each node changes only once in response to a source change.

4.3 Complete Glitch Freedom

Complete glitch freedom requires coordinating potentially conflicting propagations from different sources. The updates from two sources `v1` and `v2` conflict if there are at least two nodes `n1` and `n2` that depend on both `v1` and `v2` – `v1` $\xrightarrow{*}$ `n1`, `v1` $\xrightarrow{*}$ `n2`, `v2` $\xrightarrow{*}$ `n1`, `v2` $\xrightarrow{*}$ `n2`. We indicate the set of nodes that depend on both `v1` and `v2` as `conflict(v1, v2)`. `conflict(var)` is the union of all conflicting nodes that involve `var` and any other source.

Complete glitch freedom can be conceptually achieved via distributed locking, associating exclusive locks to each node. When the application changes the value of a var `var`, the `waitModify()` function acquires exclusive lock on all nodes in `conflict(var)`. If it succeeds, the update is performed and propagated. Else, the caller is blocked until the lock is available.

Such mechanism assumes globally ordered evaluation of lock acquisitions and releases, which can be implement using a centralized coordinator node or total order message broadcasting [23]. After lock acquisition, complete glitch freedom implements the same propagation algorithm as single-source glitch freedom. A lock is released when all the nodes in `conflict(var)` have received and processed the update. This is achieved by sending an unlock message from each node in `conflict(var)`.

Cost Analysis

A source s acquires a lock on conflicting n odes before starting a propagation. Only groups of more than one node conflicting with a different source s' need to be locked. We define the group of nodes to lock for source s with respect to source s':

$$lock_{(s,s')} = \begin{cases} N_s \cap N_{s'} & \text{if } s \neq s' \ \wedge \ |N_s \cap N_{s'}| > 1 \\ \emptyset & \text{otherwise} \end{cases}$$

We define the set of all the nodes to lock for a source s:

$$lock_s = \bigcup_{s' \in (S \setminus \{s\})} lock_{(s,s')}$$

A node in $lock_s$ releases the lock after it processes an update from s. To ensure total order of lock acquisitions and releases, we model them as messages to a locking component, consistently with the DREAM implementation (Section 5). The messages required to propagate a change from s to all the dependent nodes include: (i) the update propagation messages, which are identical to the case of single-source glitch freedom; (ii) one lock request message from the source to the locking component; (iii) one lock grant message from the locking component to the source, to notify when the lock has been granted; (iv) one lock release message for each locked node. Hence, the total number of messages for complete glitch freedom is:

$$msgs = |E_s| \ + \ 2 \ + \ |lock_s|$$

The total communication cost depends on the cost to transfer messages through external links plus the cost c_{lock} of communication with the locking component to transmit lock requests, lock grants, and lock releases.

$$c_{msgs} = c \cdot |E^e_s| \ + \ c_{lock} \cdot (2 \ + \ |lock_s|)$$

The latency for propagating a change from a source s comprises the following components: (i) submitting a request to the locking component; (ii) obtaining a lock grant from the locking component, which includes waiting that locked nodes are released; (iii) receiving a lock grant; (iv) propagating the update to all dependent nodes.

Assuming that the communication latency between a node and the locking component is l_{lock} and the (average) latency to obtain the grant for a lock is l_{grant}, then the overall latency of propagation with complete glitch freedom is:

$$l = 2 \cdot l_{lock} \ + \ l_{grant} \ + \max_{f \in F_s} l_{(s,f)}$$

Since it is difficult to analytically estimate the latency l_{grant} to obtain a lock, we empirically study its effect for different system configurations and workloads in Section 6.

Computing the $lock_s$ for source v follows the next steps. (i) Compute the set of nodes reachable from each source. For each source, we can compute the set of reachable nodes using breadth-first search (BFS), which complexity is $O(|N| + |E|)$, where N is the set of nodes and E the set of vertices. Considering Ns sources, the total complexity becomes $O(|Ns|(|N| + |E|))$. (ii) Compute the intersection of the set of v with any other set and keep only the intersections with at least two elements. Assuming an implementation with hash sets, each intersection costs $O(N)$ and the overall operation costs $O(N^2)$. (iii) Merge all remaining sets. Assuming that all sets contain all nodes, and that insertion into the result set costs $O(1)$, the overall operation costs again $O(N^2)$. Computing the $lock_s$ can be considered an expensive operation. In practice, however, we expect a limited number of dependencies in real applications, hence a limited number of edges and small intersection sets; moreover, the sets for each source are computed only when the topology changes and then reused for the delivery of each message.

4.4 Atomic Consistency

Complete glitch freedom ensures isolation of concurrent propagations. On top, atomic consistency ensures that any propagation is seen as an atomic operation by any component that observes the values of multiple nodes in the graph. As for complete glitch freedom, atomic consistency can be achieved via locking. However, in the case of atomic consistency, locking involves *all* nodes affected by the change of a var `var`, not only conflicting ones. Thus, the `waitModify()` function locks all nodes `n` such that `var` $\xrightarrow{*}$ `n`. A lock is released when all the nodes affected by a change of `var` have received and processed the update. This is achieved by sending an unlock message from each of these nodes. Also, every access to the value of a set of nodes needs to be protected, which is modeled with the request of a read lock in the `waitRead()` function. In summary, lock acquisition is necessary for the update process, as well as for every signal access. As with complete glitch freedom, total order of messages, achieved with a centralized entity or with a distributed algorithm, ensures total order of lock acquisitions and releases.

Cost Analysis

Atomic consistency requires locking *all* nodes in N_{var} before starting and update propagation from s. Locks are released when the propagation ends, i.e., all sinks F_s are updated. The messages required to propagate a change from s include: (i) the update propagation messages, identical to the case of single-source and complete glitch freedom; (ii) one lock request message from the source to the locking component; (iii) one lock grant message from the locking component to the source; (iv) one lock release message for each sinks in F_s. The total number of messages in the case of atomic consistency is:

$$msgs = |E_s| + 2 + |F_s|$$

The total communication cost depends on the cost to transfer messages through external links plus the cost c_{lock} of communication with the locking component for lock requests, grants, and releases:

$$msgs = c \cdot |E_s^e| + c_{lock} \cdot (2 + |F_s|)$$

The overall latency of propagation includes transmitting the change to all dependent nodes, communicating with the locking component, and obtaining a lock grant. As for complete glitch freedom, the latency amounts to:

$$l = 2 \cdot l_{lock} + l_{grant} + \max_{f \in F_s} l_{(s,f)}$$

However, l_{grant}, i.e., the time to obtain a lock grant, can be different – typically higher – than in complete glitch freedom for two reasons. (i) The set of nodes to lock is larger: for complete glitch freedom, it includes only nodes $lock_s$ shared with other sources; in the case of atomic consistency, it includes all nodes N_s transitively dependent on s. (ii) To obtain a lock grant, the source competes with other conflicting sources, but also with program threads that acquire a non-exclusive lock on nodes to read them.

Atomic consistency introduces a cost to access a node, which involves five steps: 1) submit a read lock request to the locking component; 2) obtain a lock grant; 3) receive the lock grant from locking component; 4) read the value; 5) submit a lock release to the locking component. The communication cost to read a variable includes sending a lock request, receiving a grant, and sending a release.

$$c_{read} = 3 \cdot c_{lock}$$

Atomic consistency also introduces a latency when reading a variable, which includes the latency to send a request, to obtain a grant, and to receive such grant:

$$l_{read} = 2 \cdot l_{lock} + l_{grant}$$

5 DREAM

This section describes DREAM's API and how DREAM implements consistency. In a nutshell, DREAM provides DRP with multiple levels of consistency guarantees by relying on a distributed publish-subscribe infrastructure for efficient propagation of changes. To the best of our knowledge, DREAM represents the first work that bridges the gap between the high-level programming entities studied in the domain of reactive programming and the propagation algorithm implemented in event-based middleware.

5.1 The DREAM API

To promote interoperability, DREAM is designed as a Java library, with vars and signals being Java objects.

5.1.1 *Local Dependencies*

DREAM defines a generic class `Var<T>` to represent vars – time-changing values of type `T`. `Var` objects are instantiated with an initial value and an optional name. The name identifies the `Var` in a distributed setting, and must be unique within the process defining `Var`.

```
Var<String> a = new Var<>("aVar", "Dream");
Var<List<Integer>> b = new Var<>("bVar", new ArrayList<>());

Signal<String> c = new Signal<>("cSig",
        () -> a.get() + "\t" + b.get().size(), a, b);

a.set("newVal");
b.modify(val -> val.add(10));

System.out.println(c.get());
c.addValueChangeListener(val -> System.out.println("c: " + val));
```

Listing 2: Definition of Vars and Signals

Listing 2 shows the definition of two vars, `a` and `b`, of type `String` and `List<Integer>`, respectively. The initial value of `a` is `"Dream"` and its local unique name is `aVar`. The initial value of `b` is an empty `ArrayList` and its local unique name is `bVar`.

In DREAM, the class `Signal<T>` defines signals. `Signal` objects are instantiated with a `Supplier<T>` object, which implements a `get()` method returning a `T` value. The concrete implementation of `get()` represents the actual definition of the signal and contains references to the time-changing values (vars or signals) the signal depends on. The `Signal` constructor receives the list of the objects the signal depends on, and an optional name. As in the case of vars, the name needs to be unique within the defining process.

Listing 2 shows the definition of a signal `c` of type `String`. The signal has a name `cSig`, and depends on the two vars `a` and `b` passed to the constructor. Whenever the value of `a` or `b` changes, DREAM invokes the `get()` method of the `Supplier` object passed as the second parameter of the constructor using Java 8 lambdas. Within the definition of the signal, one can refer to the value of the dependent vars and signals by invoking their `get()` method. Vars can be updated in two ways: using the `set()` method, which overwrites the value of the encapsulated object, or using the `modify()` method, which invokes a function on the encapsulated object. Listing 2 shows both approaches: it updates the var `a` with the new value `newVal` and it updates var `b` adding `10` to the list of integers.

Finally, as shown in the last two lines of Listing 2, developers can access the value of signals by invoking the `get()` method, which returns the current value, or by registering a `ValueChangeListener`, which gets notified whenever a new value is computed. The last line of Listing 2 shows the definition of a `ValueChangeListener` that prints the value of signal `c` upon changes.

5.1.2 *Remote Dependencies*

To support distributed scenarios, DREAM provides a `RemoteVar` class that enables a Java program to refer to a time-changing value defined in a separate process.

A `RemoteVar` behaves as a local proxy for a remote object. For instance, Listing 3 exemplifies the definition of a `RemoteVar` of type `String` that reflects the value of the

```
RemoteVar<String> remA = new Var<>("aProc", "aVar");
Signal<Integer> d = new Signal<>("d", () -> remA.get().size(), remA);
```

Listing 3: Definition of Remote Dependencies

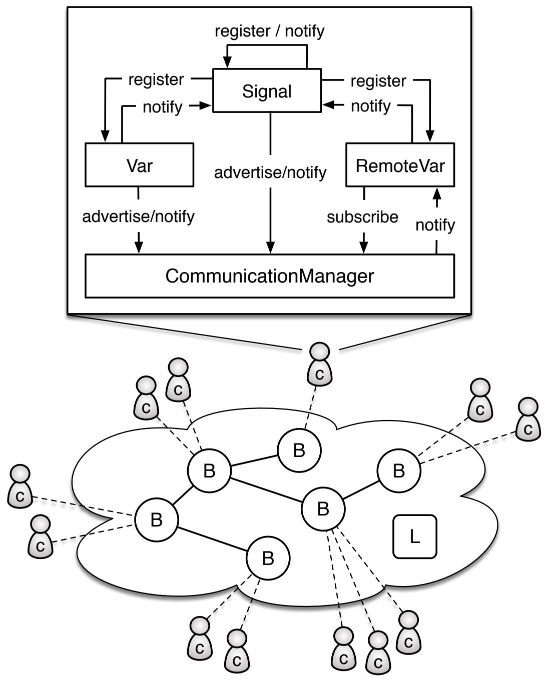


Fig. 6: The architecture of DREAM.

var or signal named `aVar` and defined in the process `aProc`. The process name and the var name uniquely identify a time-changing value – var and signal names are unique within a process and processes in a distributed application must have a unique name defined in a configuration file.

`RemoteVar`s can be used to define signals in the same way as local `Var`s. DREAM is responsible for updating the value of each `RemoteVar` with the value of the corresponding remote var or signal it. For instance, in Listing 3, `remA` defines the integer signal `d` as if `remA` was a local var.

DREAM offers a `DreamClient.listVariables()` static utility method to access the list of all variables defined within the distributed application.

5.2 Architecture and Algorithms

Fig. 6 shows the high level architecture of DREAM. DREAM consists of two parts: a *client library*, adopted by each client – denoted as `c` in Fig. 6 – and a distributed event-based communication infrastructure, consisting of one or more *brokers* – `B` circles in Fig. 6 – connected to form an acyclic undirected overlay network. The client library is responsible for propagating changes between local vars and signals, and to send and receive notifications of changes from remote clients, via the broker infrastructure. The broker infrastructure implements the communication channel between clients. An optional lock manager – `L` in Fig. 6 – handles lock acquisition and release for complete glitch freedom and atomic consistency.

5.2.1 Client Library

The DREAM client library implements the `Var`, `Signal` and `RemoteVar` classes – Section 5.1. Fig. 6 (top) shows how instances of these classes interact with each other and with the `CommunicationManager` component that acts as an interface to the event-based communication infrastructure.

First, we consider the interaction among vars and signals defined within the same process. When a `Signal` s is created, its constructor takes in input a `Supplier` and the list of time-changing variables s depends on – Section 5.1. Time-changing variables are `Var`, `RemoteVar`, or `Signal` objects.

We use the Observer pattern to update a signal when one of the variables it depends on changes. In the constructor, a `Signal registers` as a listener to the time-changing variables it depends on. Each time-changing variable stores a list of registered `Signal`s and notifies them by invoking the `notify` method whenever its value changes. Upon receiving a notification of change, a signal recomputes its value by invoking the `get()` method on the `Supplier` object received during initialization.

Remote change propagation is implemented using a publish-subscribe paradigm, offered by the `CommunicationManager` component. Remote interaction works in three steps: (i) `Var` and `Signal` objects `advertise` their existence enabling remote processes to address them using their unique name and process identifier. (ii) `RemoteVars subscribe` to remote time-changing variables addressing them through their unique name and process identifier. (iii) `Vars` and `Signals` publish notifications of their changes. The communication infrastructure distributes them to all the subscribed `RemoteVar`s, which receive them through the `CommunicationManager`.

5.2.2 Event-Based Infrastructure

The event-based infrastructure is implemented using the REDS event dispatching system [24], which builds an acyclic overlay network of brokers connected with each other through bidirectional TCP links.

```
Map<Node, List<Subscription>> subTable;

void processAdv(Advertisement adv, Node sender) {
  sendToAllNodesExcept(sender, adv);
}
void processSub(Subscription sub, Node sender) {
  subTable.add(sub, sender);
  forwardTowardsNode(sub.process);
}
void processEvent(Event e, Node sender) {
  for (Node n : subTable.keySet()) {
    if (!n==sender) {
      for (Subscription s : subTable.get(n)) {
        if (s.process==e.process && s.name==e.name) {
          sendToNode(n, e);
  } } } } }
```

Listing 4: Packets forwarding algorithm in the brokers.

Each client connects to the event-based infrastructure by establishing a TCP connection with a broker. When a client advertises a var or signal v, its `CommunicationManager` generates an advertisement packet adv containing the name and process of v and forwards it to its connected broker.

When a broker b receives an advertisement adv, it broadcasts adv to all clients and brokers connected to b, excluding the one b is receiving adv from. In this way, advertisements reach all the processes of the distributed system, both clients and brokers. The processing of advertisements in brokers is exemplified by the method `processAdv()` in Listing 4.

When a client in a process h_1 creates a `RemoteVar` that subscribes to a remote variable v on a process h_2, the `CommunicationManager` generates a subscription packet sub that contains the name and process of the remote variable and forwards it to its connected broker. Subscriptions are forwarded from broker to broker following the path that connects h_1 to h_2. Each broker maintains a subscription table that maps each connected client or broker n with the set of subscriptions the broker has received from n. Intuitively, these subscriptions indicate the set of variables node n is interested in. Processing of subscriptions in brokers is shown in method `processSub()` in Listing 4.

When a var or signal v in a client c changes, the `ConnectionManager` of c generates an event notification packet e containing the new value of v. Brokers distribute the event e to all their connected clients and brokers that expressed an interest in the changes of v through a subscription, that is to say, to all the clients and brokers having a subscription that refers to v in the subscription table. In this way, an event e that notifies a change to a variable v is distributed from the producing process to all and only the processes that contain a `RemoteVar` that refers to v. The processing of events at brokers is exemplified by the method `processEvent()` in Listing 4.

In DREAM, the communication infrastructure that delivers change notifications is independent from the way notifications are processed at clients. For instance, the communication infrastructure can be replaced with remote method invocations from a time-changing variable to all `RemoteVar`s that depend on it. Yet, a distributed event-based infrastructure supporting multicast one-to-many dispatching of event notifications reduces network traffic and better exploits data locality compared to multiple point-to-point communications. For example, in a scenario in which multiple processes located in North America declare `RemoteVar`s that refer to a var located in Europe. If the event-based infrastructure includes one broker in Europe and one broker in North America, every change to the var is delivered only once from Europe to North America, where the latter broker dispatches it to all the interested processes.

5.2.3 Ensuring Consistency Guarantees

This section discusses how DREAM implements the consistency guarantees presented in Section 3.4.

FIFO and Causal Consistency

In DREAM, both client-to-broker communication and 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 clients, 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.

An acyclic topology that preserves end-to-end FIFO ordering is sufficient to ensure that messages are delivered in causal order [22]. Thus, the architecture of DREAM always ensures both FIFO and causal consistency.

Single-source Glitch Freedom

DREAM implements single-source glitch freedom within clients, which are responsible for detecting and avoiding glitches based on the complete knowledge of the dependency graph that they gain through advertisement packets.

The approach follows the abstract algorithm presented in Section 4. DREAM annotates each event notification e with unique identifiers of the source and with the change that triggered e. When a signal receives an event notification from a source s with change identifier id, it analyzes the dependency graph to check whether it needs to receive additional notifications for the same change. In this case, it stores the event notification in a queue, and processes it only when all the expected notifications for id are received. Note that the analysis on the dependency graph is computed only once, when the dependency graph is created. Hence the overhead of this step does not impact communication operations.

Complete Glitch Freedom

In the case of complete glitch freedom, DREAM enforces ordering constraints using a centralized `LockManager` process, as in the abstract algorithm of Section 4. When a source s needs to propagate a change, it first analyzes the dependency graph to compute the set of nodes N involved in the propagation that can conflict with concurrent propagations from other sources. Then, it contacts the `LockManager` to request a lock on all nodes in N. The `LockManager` stores lock requests in a queue, until all the requested nodes become available. At this point, it assigns the lock on N to s and notifies s with a lock grant packet. After a node $n \in N$ has successfully processed the change notification from s, it contacts the `LockManager` to release the lock, thus making the node available for other sources.

DREAM optimizes this process by recognizing when the nodes to be locked only depend on sources defined in a single process. In this case, the locking discipline is implemented with local Java locks, to avoid the burden of remote communication with the `LockManager`.

Atomic Consistency

DREAM implements atomic consistency using the same locking approach as in complete glitch freedom. Since atomic consistency needs to ensure total order among both read and write accesses to time-changing values, the locking algorithm presents two substantial differences with respect to complete glitch freedom: (i) A source locks *all* the nodes that directly or indirectly depend on it. (ii) Nodes are not only locked in exclusive mode by propagation processes that change their value, but also in non exclusive mode by program threads that want to access their value.

Number of brokers	10
Number of clients	50
Distance between nodes (network hops)	1
Distance to lock manager (network hops)	2
Topology of the network	Scale-free
Percentage of pure forwarders	30%
Distribution of clients	Uniform
Link latency	1 ms–5 ms
Number of vars (sources)	10
Depth of dependency graphs	5
Number of signals per level (avg)	2
Number of dependencies per signal (avg)	2
Degree of nodes sharing	0.1
Degree of nodes locality	0.5
Frequency of var updates (avg)	0.5 change/s
Frequency of read access to signals (avg)	0.2 read/s
Duration of a read access	100 ms

TABLE 3: Parameters used in the default scenario.

6 Performance Evaluation

Our evaluation assesses the cost of ensuring different levels of consistency guarantees. We evaluate DREAM in a cloud Amazon EC2 deployment as well as in simulation for parameter sensitivity. We also compare the propagation algorithms implemented in DREAM with the state of the art algorithm SID-UP, used in Distributed REScala [9]. To the best of our knowledge, SID-UP is the most efficient algorithm for change propagation: a recent comparison [9] shows its advantage with respect to the algorithms adopted in ELM [25], Scala.React [1], and Scala.Rx [26]. SID-UP provides complete glitch freedom by allowing a *single* propagation to occur at any moment in time. Conversely, DREAM support parallel propagation by selectively locking only the nodes that may lead to conflicts.

The code of DREAM as well as all the datasets used in the experiments presented in this section is publicly available.[1]

6.1 Experiment Setup

We compare the cost of propagation for different levels of consistency, namely causal (`causal`), single-source glitch freedom (`single`), complete glitch freedom (`complete`), and `atomic`. We also consider SID-UP (`sid up`), which offers complete glitch freedom. We do not consider FIFO consistency since DREAM always ensures by design at least causal consistency and since FIFO consistency does not differ from causal consistency in terms of network traffic.

We measure network traffic and propagation latency. For network traffic, we consider both client-broker communication and broker-broker communication and we measure the average traffic flowing in the entire network per second. For the propagation latency we compute the average difference between the time when an event is produced, i.e., when the value of a var object v changes, and the time when an event is processed, i.e., when the notification of the change is used to update the value of a signal s that depends, directly or indirectly, on v.

To isolate the cause of the propagation cost, we define a default scenario with the values in Table 3 and we vary a single parameter in each experiment. We run the default scenario both on a real cloud deployment on Amazon EC2 and on a simulated environment based on the Protopeer network emulator [27]. We then use the simulated environment to perform parameter sensitivity analysis.

Given the unavailability of empirical studies that characterize large production-quality RP programs, our approach is to start from values in Table 3 that are typical in medium-scale systems [28] and perform a sensitivity analysis on these values. Our default scenario consists of 10 brokers connected in a randomly-generated scale-free overlay network and serving 50 clients. We assume that 30% of the brokers are pure forwarders – they have no directly connected client. Clients are uniformly distributed among remaining brokers.

On Amazon EC2, we deploy each broker and the lock manager on a separate t2.micro instance[2], while all clients run on a t2.2xlarge instance.[3] With this approach, we can precisely measure the propagation delay using the wall clock time of the instance where the clients run. Brokers are connected to each other and to clients via TCP connections.

In the simulated environment, we assume that the brokers are directly connected to each other and to their clients using a single network link. Link latency is uniformly distributed between 1 ms and 5 ms. Clients can reach the lock manager within two network hops, on average. We assume that packet's processing time is negligible with respect to link latency[4]. We consider changes from ten vars propagating across a dependency graph of depth five, with an average of two nodes per level. On average, each signal in a level depends on two nodes of the previous levels. Given two vars v_1 and v_2, there is a 0.1 probability that at least one signal depends on both v_1 and v_2. More formally, there is a 0.1 probability that $N_{v_1} \bigcap N_{v_2} \neq \emptyset$.

On Amazon EC2, each var and signal is stored on a separate client. In the simulated environment, to reproduce different application loads, we introduce a *locality* parameter – set to 0.5 in the default scenario – for the probability that a node n in the propagation graph lays on the same client as one of the nodes it depends on. On average, vars change once every two seconds and the value of each signal is accessed once every five seconds. Access frequency is relevant for the `atomic` protocol that requires the acquisition of a read lock. Each read operation lasts for 100 ms and then the read lock is released.

Our experiments consist of two phases: (i) clients define and deploy vars and signals, (ii) the value of each var is repeatedly updated, causing a propagation in the dependency graph. Each experiment lasts 1000 (simulated) seconds. We measure the average value as well as the 90% confidence interval of the overall network traffic and propagation latency over ten repetitions using different random seeds in each repetition to generate the network topology and the dependency graphs.

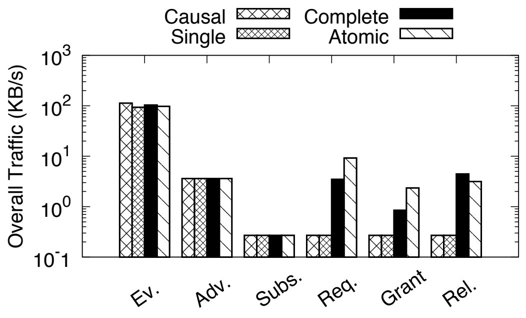
Since the paper on SID-UP does not prescribe a specific communication infrastructure between processes [9], we adopt the same event dispatching infrastructure used for DREAM, enabling a fair comparison between the protocols.

[1]Url provided for review, it will be updated to a dedicated address in the final version. https://github.com/allprojects/dream-java

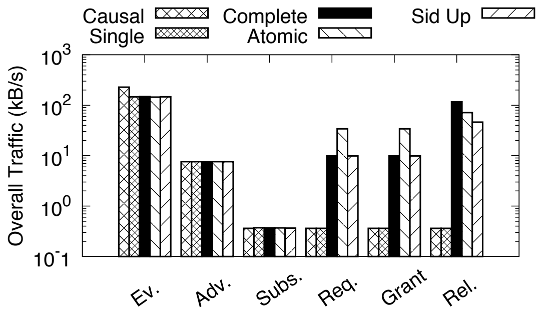
[2]Equipped with 1 core of an Intel Xeon family processor running at up to 3.3 GHz, 1 GB or RAM, 8 GB of disk on an SSD drive.

[3]Equipped with 8 cores of an Intel Xeon family processor running at up to 3.0 GHz, 32 GB or RAM, 8 GB of disk on an SSD drive.

[4]This assumption has also been validated in the Amazon EC2 deployment. Several algorithms have been proposed for efficient event dispatching in brokers [29], [30], [31].



(a) Amazon EC2 deployment



(b) Simulation

Fig. 7: Default scenario: overall traffic.

6.2 Experimental Analysis and Results

First we present the results in the default scenario. Then, we investigate the influence of the parameters in Table 3.

6.2.1 Default Scenario

Fig. 7 shows the average traffic in the default scenario for various types of packets exchanged. Fig. 7a shows the results we observed on the Amazon EC2 deployment, Fig. 7b shows the results we measured in the Protopeer network emulator. In Protopeer, we also implemented the SID-UP protocol for comparison. The differences between the traffic in Amazon EC2 and in Protopeer are due to slightly different topologies in the experiments. Despite these differences, they both confirm the key findings below.

For the advertisement and subscription packets, all the protocols exhibit the same traffic amount. These packets are used in the first phase of each experiment to build the dependency graph. Advertisements announce the definition of a var or signal, and are broadcast to all the brokers and to all the clients. Subscriptions define a dependency between two nodes, and are distributed over the unicast path that connects the subscribing node to the node it depends on. Since in the experiments `sid up` adopts DREAM's mechanism to distribute advertisements and subscriptions, we observe the same amount of advertisement and subscription packets.

The traffic of event packets – used to propagate change notifications – is almost identical for all protocols that guarantee glitch freedom – `single`, `complete`, `atomic`, and `sid up`. The small differences are due to the protocol-related information carried by event packets, which slightly changes in size based on the protocol, as we further discuss in the following. In contrast, the absence of glitch freedom

Protocol	Amazon EC2 Deployment Avg. latency	Simulation Avg. latency	Simulation 90% c.i.
`causal`	56.7 ms	18.6 ms	1.3 ms
`single`	55.1 ms	20.6 ms	1.5 ms
`complete`	65.1 ms	33.8 ms	2.0 ms
`atomic`	77.5 ms	40.4 ms	1.7 ms
`sid up`	n.a.	41.1 ms	1.9 ms

TABLE 4: Default scenario: average propagation latency.

causes a larger number of events in `causal`, because events are replicated when multiple paths converge in one node.

Lock request, lock grant, and lock release packets must be exchanged to guarantee complete glitch freedom in the `complete`, `atomic`, and `sid up` protocols. `complete` and `sid up` adopt only exclusive locks, acquired before the propagation of a change. `atomic` also exploits non exclusive read locks to control the read accesses to the value of signals. This explains the higher traffic of lock request and grant packets in the `atomic` case. Such three protocols also differ in locks release: `complete` releases the lock on a node n as soon as n has been updated, `sid up` and `atomic` release the lock only when the entire propagation terminates, i.e., when all nodes that directly or indirectly depend on the source of change have been updated. `Complete` sends a separate lock release packet for each locked node. Conversely, `sid up` and `atomic` send a lock release from each node that has no outgoing edges. In the default scenario, the number of nodes to lock surpasses, on average, the number of nodes with no outgoing edges, motivating the larger number of lock release packets in `complete`. The difference of lock release packets between `atomic` and `sid up` is again explained by the presence of additional read locks in `atomic`.

We also evaluated the size of individual packets. In our default scenario, the average size of events is 0.55 kB in the case of `causal` and `single`, and increases to about 0.7 kB in the case of `complete`, since events also carry the identifier of the nodes that need to release the lock after propagation. The cost of events is affected by the complexity of the Java objects used in vars and signals and, in the case of `complete`, by the number of nodes that need to release the lock after propagation. Concerning locking, lock grant and release packets have a fixed size of about 0.17 kB, and lock requests have a fixed size of about 0.65 kB. Advertisements and subscriptions are exchanged only when new vars and signals are defined: the size of the former is about 0.73 kB and the latter requires 0.56 kB in our default scenario, but it can change with the number of dependencies encoded into the subscription.

Table 4 considers the latency between the time when a change occurs in a var and the time when the change is processed in a dependent node. In the simulated environment, we performed ten executions and we also report the 90% confidence interval.

In the `causal` protocol, each node processes event notifications as soon as they are received. `causal` and `single` present the smallest latency among all the considered protocols. The `complete`, `sid up`, and `atomic` protocols exhibit a significantly larger latency because all of them need to request and obtain a lock before starting change propagation.

The `complete` and `sid up` protocols offer the same consistency guarantees, yet `complete` produces a smaller

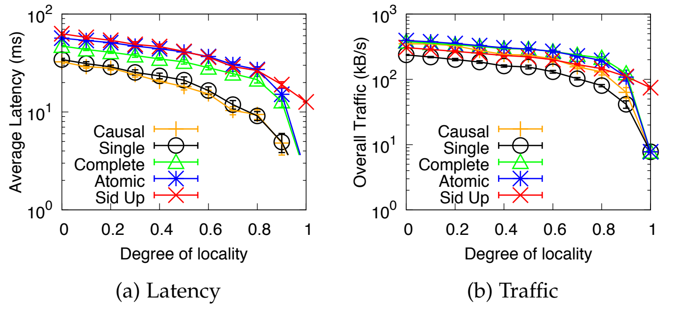


Fig. 8: Degree of locality.

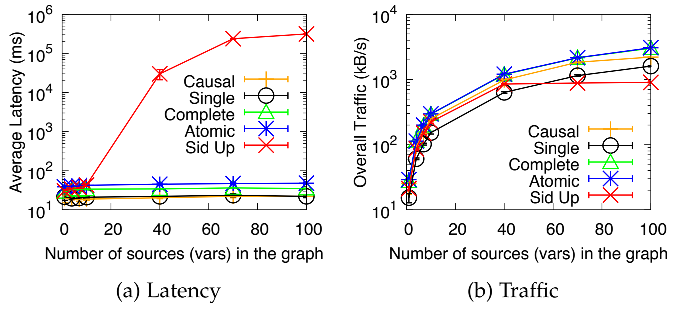


Fig. 9: Number of Vars.

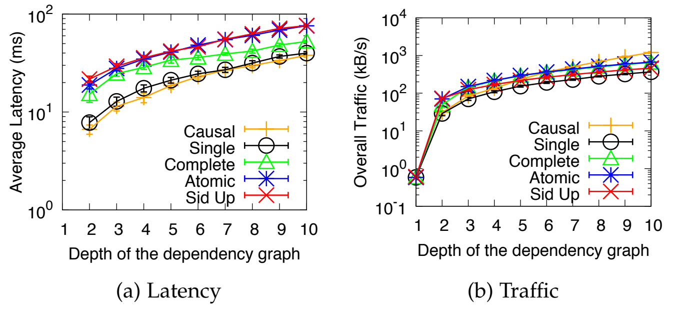


Fig. 10: Depth of the dependency graph.

latency because `sid up` does not support concurrent propagations, not even across different sets of nodes. For this reason, a propagation starts only after all the previous ones have terminated, increasing the overall latency. Conversely, `complete` allows concurrent non-conflicting propagations, reducing the waiting time before lock acquisition, and thus also reducing the overall latency.

Finally, `atomic` exhibits larger latency than `complete`, since acquiring exclusive locks for change propagation also conflicts with the non exclusive locks to access signals value.

The larger network latency in the Amazon EC2 deployment leads to larger propagation times than in the simulated environment. Nevertheless, the absolute difference between protocols is comparable in the two scenarios.

6.2.2 Parameter Analysis

Additional experiments performed in the Protopeer network emulator assess the impact of specific factors that influence performance.

Locality

Fig. 8 shows the effect of changing nodes locality, i.e., the probability that a node is on the same process of the nodes it depends on. Higher locality means that more event notifications are delivered with local method invocation instead of being dispatched through the brokers network. As a consequence, upon increasing locality, both propagation latency (Fig. 8a) and overall traffic (Fig. 8b) decrease.

Similar to the default scenario, the `causal` and `single` protocols exhibit the smallest latency. Since their latency only depends on the time required for events delivery, they are highly affected by locality. In the extreme case locality=1, the latency drops to 0 as events are always delivered via local method calls. A similar consideration explains the impact of locality on the overall traffic, which drops by more than an order of magnitude when locality increases from 0 to 1.

As it does not support concurrent propagations, `sid up` also benefits from higher locality. When locality increases, the propagation latency for each change decreases, hence waiting time for subsequent changes decreases as well. In case of a locality=1, the latency of `complete` and `atomic` drops to 0. This result is explained by an optimization in DREAM: in absence of potential conflicts that involve multiple processes, DREAM adopts local locking to avoid expensive communication with the `LockManager`. `sid up` does not support this optimization and, with high locality, it exhibits the highest propagation latency and network traffic.

Number of vars

Fig. 9 shows the behavior of the protocols when increasing the number of vars. A higher number of vars does not impact on the latency of the protocols implemented in DREAM (Fig. 9a). Conversely, this scenario shows a major limitation of `sid up`: since its locking mechanism does not allow concurrent propagations of changes from multiple sources, the lock manager can become a bottleneck, with more and more lock requests waiting for acceptance.

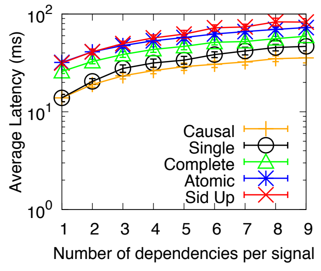
This issue also has an impact on the overall network traffic. In the protocols implemented in DREAM, traffic increases linearly with the number of vars. In `sid up`, propagations need to wait for lock acquisition and fewer events can access the network in a given time window. Hence locking is a bottleneck for the throughput, which does not increase with the number of vars.

Depth of the Dependency Graphs

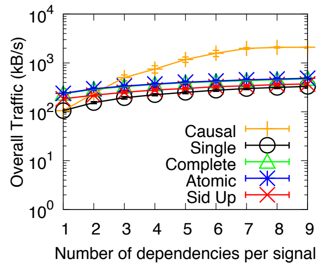
Fig. 10 shows the influence of the depth of each dependency graph on the latency and traffic. As expected, a larger graph produces more traffic and leads to a higher propagation latency. All protocols consistently exhibit the same behavior.

Number of Dependencies per Signal

Next we inspected the effect of the number of dependencies per signal on propagation latency and on network traffic. The propagation latency (Fig. 11a) increases for all the protocols, since more events need to be delivered via the broker network. A larger number of dependencies also increases the overhead of single source glitch freedom: each

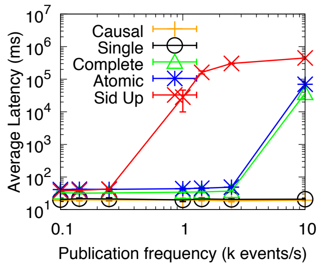


(a) Latency

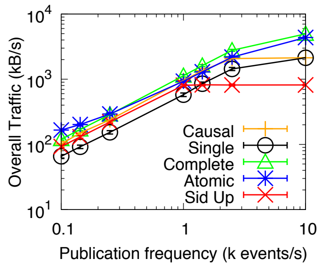


(b) Traffic

Fig. 11: Number of dependencies per signal.

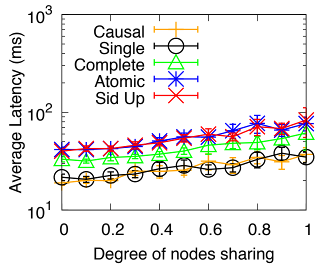


(a) Latency

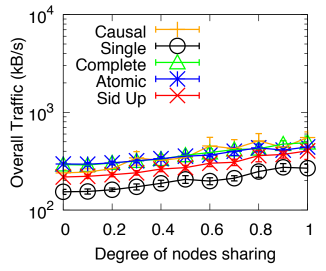


(b) Traffic

Fig. 13: Frequency of updates.

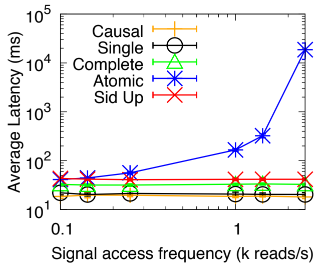


(a) Latency

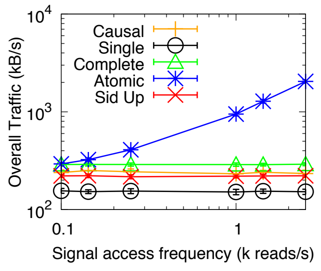


(b) Traffic

Fig. 12: Degree of nodes sharing.



(a) Latency



(b) Traffic

Fig. 14: Frequency of read accesses.

node waits for an event notification from all the nodes it depends on before processing any input. As a consequence, the average latency for `single` increases much faster than the average latency for `causal`. Also, increasing the propagation latency impacts on the waiting time for lock acquisition. This is especially visible for `sid up`, which does not allow concurrent propagations of changes: as the number of dependencies increases, the latency of `sid up` overcomes the latency of `atomic`. The overall network traffic (Fig. 11b) increases for all protocols, since more edges require more event propagations. This factor influences the `causal` protocol the most because a higher number of edges in the graph increases the probability of event duplication.

Degree of Nodes Sharing

Fig. 12 shows how the probability of sharing nodes among the dependency graphs originating from multiple vars impacts on the cost of the protocols. We control node sharing by building a separate dependency graph for each var, and connecting such graphs with varying probability (x axis, Fig. 12). A connection is established adding two edges that connect the first graph to the second graph.

Adding new edges impacts both on propagation latency (Fig. 12a) and on the overall network traffic (Fig. 12b) for all protocols, as discussed previously. Shared nodes between sources influence the number of locks acquired in the `complete` protocol: in the extreme case of share probability=1, `complete` exhibits almost the same latency as `sid up` because fewer propagations can occur concurrently. Similarly, with high probability of node sharing, traffic for `complete` increases due to increasing lock request, grant, and release packets, and becomes similar to the network traffic of `atomic`.

Frequency of Updates

Fig. 13 shows the impact of vars change frequency. This parameter significantly influences the latency for protocols that exploit node locking: `complete`, `atomic`, and `sid up` (Fig. 13a). The behavior is similar to the case of increasing the number of vars. With higher updates frequency, the locking system becomes a bottleneck: events waiting for lock grant accumulate, increasing the latency between the occurrence of a change and its propagation to all dependent nodes. Again, `sid up` does not allow concurrent propagations and suffers the most from high frequency updates.

As expected, the overall traffic increases with the frequency of updates (Fig. 13b) for all the protocols. Interestingly, `sid up`'s traffic flattens after 1000 events per second because the protocol reaches its maximum throughput and event propagation is limited by the locking system.

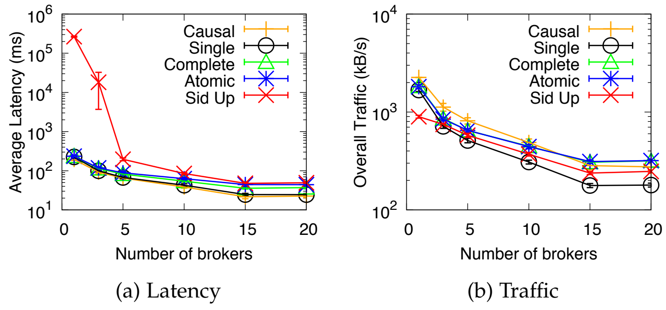
Frequency of Read Accesses

Fig. 14 shows the effect of changing the frequency of read accesses to signals. As expected, this parameter influences only the `atomic` protocol, which is the only one requiring a lock acquisition to access a signal value.

Since the acquisition of read locks can conflict with the acquisition of the exclusive locks that control the propagation of changes, the latency of propagation for `atomic` increases with the frequency of read accesses (Fig. 14a). The overall traffic increases as well (Fig. 14b) since a higher number of read accesses generates a higher traffic of read lock request, grant, and release packets.

Number of Brokers

Fig. 15 shows the effect of exploiting a distributed infrastructure for change propagation. We consider a fixed network



(a) Latency (b) Traffic

Fig. 15: Number of brokers.

topology with 20 physical hosts (we disregard client hosts), and we vary the number of brokers deployed on top of this physical network from one to 20. The case of a single broker is the centralized scenario: a single component processes and dispatches all packets exchanged among clients. This approach reduces the chances of exploiting locality and forces the delivery of all packets – through multiple physical hops – to the single broker. As a consequence, the lower the number of nodes, the higher is the average propagation latency (Fig. 15a) and the overall network traffic (Fig. 15b). In the case of SID-UP, the increased latency creates a bottleneck with less than five brokers. Notifications get accumulated, waiting for dispatching, and limit the maximum throughput (Fig. 15b) significantly increasing latency (Fig. 15a).

6.3 Summary of the Performance Evaluation

We summarize the results on the cost to ensure different consistency guarantees with the protocols implemented in DREAM as follows.

First, the `causal` protocol propagates changes with the smallest latency, but it produces a larger traffic compared to all other protocols, due to possible events duplication. This means that single source glitch freedom is not only semantically important [6], but also beneficial for traffic.

Second, introducing order constraints among propagations that originate in different vars comes at a significant price for propagation latency and overall network traffic. It requires distributed consensus on the order of certain messages, which is implemented in DREAM via a centralized component, the `Lock Manager`. In our analysis, the overhead of such locking mechanism is visible in the `complete` and `sid up` protocols (complete glitch freedom), and in the `atomic` protocol (atomic consistency).

Third, DREAM's locking mechanisms for the `complete` and `atomic` protocols are more efficient than `sid up` 's: `sid up` locks the entire graph before the propagation, hence not supporting concurrent propagations from different sources. Conversely, DREAM locks at node granularity: a propagation involving a set of nodes is allowed to take place if none of them is locked, enabling non-conflicting concurrent propagations. In some cases, `sid up`'s traffic can be smaller than the `complete` case. This depends on the shape of the graph, since `sid up` releases a lock when the propagation terminates, but `complete` delivers a lock release for each locked node n, after its value has been updated. Such approach, however, enables `complete` to release locked resources earlier, achieving higher concurrency.

Finally, atomic consistency introduces a high overhead both in terms of propagation latency and in terms of network traffic when the values of signals are accessed frequently. Indeed, atomic consistency is the only level that enforces order constraints on read operations.

7 Qualitative Evaluation

This section evaluates the effect of ensuring multiple levels of consistency guarantees within a DRP middleware on development effort and program design.

7.1 Research Questions

First, we assess the flexibility of DREAM in providing multiple levels of consistency, targeting the research question:

RQ1. What is the advantage of DREAM in terms of development effort versus a middleware with a fixed consistency level?

To this end, we evaluate the effort to manually ensure adequate consistency guarantees if they are not offered by the middleware, that is, the burden on the developer in case she has to manually implement the logic to ensure the right consistency level for the application at hand.
Second, we assess the advantage of DREAM in terms of software design compared to a scenario where the consistency guarantees are implemented manually, targeting the research question:

RQ2. What is the impact of implementing the required consistency guarantees manually on the design of a distributed reactive application?

The purpose of this research question is also to provide a qualitative evaluation about the design experience and effort of developing reactive applications with different levels of consistency guarantees. In the rest, we describe the applications that we implemented to answer the research questions above, and present the results of our evaluation.

7.2 Applications

We implemented each application both by relying on DREAM and by manually achieving the desired level of consistency on top of a middleware – a simplified version of DREAM – that does not offer it.

Distributed Form

In the distributed form application, users access the fields of a form on a remote server to fill them. Different users can access different parts of the form. For example, a secretary can enter the office hours and the management can change the per-hour salary. Similar to a spreadsheet, some fields depend on the values entered in other fields and are automatically updated. For instance, the monthly employees salary depends on the per-hour salary. Also, an alert may become active when some values do not meet the application constraints. For example, if the number of working hours for an employee is too high.

Single-source glitch freedom is required to avoid spurious recomputations in case a field depends on multiple other fields, and those depend on a common value.

Management Application

This application consists of three components, C1, C2, C3. C1 allows managers to enter working tasks. Each task includes detailed information, such as a description of that task and its subtasks. C2 assigns tasks to workers based on schedule optimizations. C3 depends on C1 and C2 and displays task information and assignments. In this setting, causal consistency ensures that C3 always displays task information when an assignment for the task is computed at C2. It is instead acceptable for C3 to display a task and its related information even when an assignment is not available for that task.

Scrum Board

The scrum board is a distributed application for test-driven development. Each client holds a list of task pairs (development-task, testing-task). A server keeps track of all the development tasks and all testing tasks in two separate signals. User applications can run a query to read the development tasks and the testing tasks. When queried, the two signals should be consistent: for each development task in one signal the corresponding testing task should be available in the other signal and vice versa.

This behavior requires atomic consistency since each change must propagate atomically to the two signals stored in the server.

Chat Application

The chat application is based on a P2P architecture. A server component is only used for bootstrapping and handles the list of online users. The application supports chat rooms with multicast communication: in a chat room each client writes messages to a var and all other clients hold a signal that depends on the var. Causal consistency ensures proper visualization of the messages within a room. For instance, consider a room with three clients, C1, C2, C3. C1 sends a message M1: “Hi”. C2 reads M1 and replies with M2: “Hi back”. Without causal consistency, C3 might receive M2 before M1 and read “Hi back” *before* “Hi”.

7.3 Design and Development Effort

We analyze the steps that are necessary to implement a desired level of consistency on top of a middleware that does not provide that consistency level.

For the **distributed form**, implementing single-source glitch freedom includes (i) considering which changes depend on the same input and (ii) temporarily accumulating changes until all required values become available for the recomputation.

A possible approach is to collect the values from different paths from the same input form into queues, and use them for the recomputation only when all queues contain at least one value. Such solution requires: (i) Imperative management of the queues. (ii) Knowledge of the topology of the dependency graph: if the topology changes, the developer needs to re-consider the design of the glitch freedom mechanism and possibly add new queues. (iii) Additional information attached to the propagated values, to uniquely identify the node that triggered the propagation and add the value to the correct queue. (iv) If the middleware does not guarantee FIFO consistency, values need also be re-ordered in the queues.

If clients can concurrently access the shared form, complete glitch freedom is needed. This guarantee can be implemented by locking conflicting nodes (Section 3.4.4). If such feature is not available in the middleware, the developer needs to either: (i) lock only conflicting nodes manually, (ii) implement an equivalent protocol to ensure total order of propagations, or (iii) entirely avoid concurrent propagations. The first approach require again complete knowledge of the dependency graph, and leads to a new design in case of topology changes. The latter two choices might incur in the same performance issues discussed for SID-UP.

In the **management application**, manual implementation of causal consistency presents difficulties similar to single-source glitch freedom. C3 needs to store changes from C2 until the corresponding information from C1 is available. This requires knowledge of the dependency graph and additional information associated to the propagated values to uniquely order them and relate them to the node that triggered the change. A more general solution that does not require knowledge of the specific topology associates vector clocks to each propagation of values to capture causal relations among updates. Finally, since FIFO consistency is a necessary condition for causal consistency, if the middleware does not ensure FIFO propagation of updates, the developer also needs to manually and imperatively re-order the values received in the queues before using them.

The **scrum board** application presents similar issues as those described for the distributed form when it comes to implement atomicity. Indeed, atomicity requires distributed consensus on the order of messages to avoid that accesses to multiple signals interleave with the propagation of changes from some source.

From the **chat application** we learn that ensuring causal consistency for the propagation of updates is not sufficient to ensure causal consistency at the application level – that is, between chat messages. Indeed, in the chat application we use a separate var for each user to represent the list of messages published by that user. Then, each user displays the entire chat conversation as a signal that composes messages from all the users. Unfortunately, causal consistency in the middleware only targets the propagation of changes from a *single* var, and cannot capture the dependency between the update of such var – publishing a message – and *previous* updates of another signal – receiving a message.

There are two solutions to ensure causality of message delivery: (i) Implement causality on top of the distributed middleware, e.g., using vector clocks associated to messages and showing a new message M in the UI only when all the messages that causally precede M have been received. (ii) Adopt atomic consistency in the middleware, to guarantee that each propagation process is atomic, meaning that all users see the same total order of messages within a room.

Finally, we attempt to provide an intuition of the *size* of each solution as a measure of development effort. Table 5 shows the size of the modules that constitute the propagation logic of each application (we intentionally omit the parts that are irrelevant for the analysis, e.g., the GUI) comparing lines of code for the DREAM implementation and the case in which the consistency guarantees are im-

	DREAM	Manual consistency
Distributed Form		
Single-source glitch freedom	297	343
Complete glitch freedom	314	607
Management application		
Causal consistency	392	519
Scrum Board		
Atomic consistency	155	213
Chat application		
FIFO consistency	516	798

TABLE 5: Size of the propagation logic in the case studies.

plemented manually. The latter requires significant effort with an increase in the lines of code that implement the propagation logic between 15% and 93%.

The considerations above answer research question RQ1 showing that DREAM significantly reduces design and development effort compared to a middleware that offers weak consistency.

7.4 Software Design

Based on our case studies, we now discuss the impact of DREAM on software design.

Lower the Level of Abstraction

Signals abstract over single changes. For example, from the value of a signal there is no way to say if it never changed or if it is the result of switching to a different value and switching back. In contrast, implementing consistency guarantees at the application level always requires to reason about changes and their order. In our case studies, this involves: (i) Decorating the propagated values with meta-information, such as the producer of the change. (ii) Temporarily storing propagated values into queues to enforce ordering relation: this is needed for FIFO, causal, and single-source glitch freedom. In practice, signals are converted into discrete events relinquishing the level of abstraction they provide.

We conclude that a middleware that offers a lower level of consistency than required by the application often forces the application code to lower the level of abstraction and reason about individual changes rather than time-changing values and their dependencies, which cancels the main benefit or RP.

Fragile Solutions

Implementing a consistency guarantee at the application level forces the developer to reason about the individual changes and the order between them. The knowledge of the topology of the dependency graph can sometimes simplify the implementation or make it more efficient: for instance, only some nodes are affected by single-source glitch freedom, and complete glitch freedom applies only when multiple concurrent propagations are allowed. Yet, solutions based on these assumptions are very fragile. If more components are added or the structure of the graph changes, an ad-hoc solution may become unsound.

Lack of Composabilty and Modularity

Implementing ad-hoc solutions that rely on assumptions on the dependency graph topology also results in a lack of composability and modularity. The lack of composability manifests because composing two reactive applications that individually support a certain level of consistency might not result in an application that ensures the same level of consistency. This issue conflicts with one of the main goals of reactive programming, which is intended to increase composability [16]. Similarly, ad-hoc solutions also conflict with modularity, since the developer of a component needs to know the dependencies that exist in the whole application.

Unnecessary Use of State

Implementing causal consistency and glitch freedom when not natively offered by the middleware requires storing values before processing them, for instance into queues. This approach complicates the application because it introduces mutable state, hence leading to potential synchronization issues and non-local updates, even when it is not needed for the functional requirements of the application. Also, abstraction over state is among the motivations for RP [6], and reintroducing state because of consistency conflicts with the adoption of RP in the first place.

The considerations above answer research question RQ2 showing that DREAM improves software design compared to manually implementing consistency guarantees.

8 Assumptions and Discussion

This section discusses aspects that are orthogonal to the core contributions of this work and represent assumptions to it. We discuss their implications and possible research strategies to tackle these aspects in future research.

Fault-Tolerance

Our work does not consider failures of nodes or links. Nevertheless, partial failures are a primary challenge in distributed systems, especially when applications are executed on clusters of commodity hardware, as common in industrial practice today [32].

In the context of DREAM, we distinguish two types of failures: failures of the nodes and links that form the communication infrastructure – that is, the brokers of the distributed event dispatching system and their connections – and failures of the clients.

Techniques that ensure fault-tolerance and exactly-once delivery of messages in distributed event-based infrastructures have been proposed in the early 2000s, for instance in Hermes [33] and Gryphon [34]. These solutions build on the concept of *soft state*, meaning that the internal state of the brokers – the routing tables – is transient and needs to be updated with periodic messages. In the presence of failures, the state expires and triggers a reconfiguration of the broker topology. These techniques incur low overhead during the normal activity. They require that messages are stored at the sources or intermediate brokers and replayed upon failures. As a consequence, in the presence of failures, the clients temporarily experience higher latency in message propagations [34].

In the DRP model, clients store vars and signals and the updates propagate through the dependency graph. Modern distributed stream processing systems, that rely on a similar dataflow model, propose different fault-tolerance mechanisms. For instance Flink adapts and optimizes the classic distributed snapshot algorithm by Chandy and Lamport [35] to its dataflow model [36]. In this approach, special system messages flow within the same channels as application messages and mark the boundaries of global consistent snapshots. In the case of failures, the last available snapshot is restored and all subsequent messages are replayed. This approach requires that the state is persisted on some durable data store that survives the failure and that the sources can replay their messages starting from the last snapshot. The requirement to periodically store the state of a node may incur some overhead with respect to what we measured in Section 6.

An additional issue is related to the presence of side effects. While the fault-tolerance mechanisms discussed above ensure that the state of the nodes in the dependency graph is correctly restored, they do not take into account possible side effects that the propagation of a change might have triggered. As a consequence, the replay of messages in the case of failures might trigger the side effect more than once and lead to undesired program states.

Dynamic Changes

Currently, DREAM assumes that the dependency graph is fixed and does not change over time. Since the presence of dynamic changes to the dependency graph significantly complicates enforcing glitch freedom – especially in presence of concurrent propagation – some languages, like ELM [25], support only a static graph. Existing languages that support glitch freedom (e.g., REScala [9] and Flapjax [37]) forbid concurrent update propagations in the reactive system. Combining concurrent propagation and glitch freedom – let alone other consistency guarantees – is an open research problem. In this work we support concurrent propagation and leave studying the interaction of concurrency and consistency levels to future research.

Protocols for Isolation

Complete glitch free and atomic consistency require some form of isolation between updates that propagate concurrently. Although DREAM currently relies on a centralized lock manager to order concurrent updates, other solutions are possible.

Two types of protocols exist to ensure isolation. Lock-based protocols, like the one adopted in DREAM, avoid conflicts through mutual exclusions. A distributed consensus protocol would be a viable alternative to a centralized lock manager [38]. However, it would require a group communication between all the nodes, thus making them more tightly coupled and complicating the configuration and start-up phase. The second type of protocols are referred to as *optimistic protocols* [39]. They do not prevent conflicts, but detect them, and cancel and restart the computations involved in the conflict. Optimistic protocols provide better performance if the conflicts are rare. However in our case, computations might trigger side effects: in the case of conflicts, restarting a computations might trigger the side effects more than once, leading to incorrect program state.

Integrating Multiple Levels of Consistency

In this paper we do not consider the presence of different levels of consistency within the same program. We assume the level of consistency to be a configuration property of the middleware that all DRP programs running on it inherit. Supporting multiple levels of consistency within the same program would require (i) to support multiple consistency levels in the middleware and (ii) to prevent undesired interactions between values at different levels in the same program.

Supporting multiple levels of consistency in disjoint dependency graphs would be a straightforward extension to DREAM. Defining the semantics and the propagation mechanisms in the case multiple levels of consistency can co-exist in the same dependency graph is an open research issue.

Preventing undesired interactions between values in the same program has been explored by Holt et al. [40] who introduce a typing discipline that statically detects and prevents wrong combinations of values at different levels of consistency. Similarly, DCCT is a data-oriented language that mixes multiple consistency levels [41] where *actions* – for example, queries – access a distributed storage, and annotations define consistency of values. Developing a language and system that supports multiple consistency levels is ongoing work [42].

9 Related Work

Reactive Programming

RP refers to language abstractions to model time-changing values. It was originally proposed in the context of strictly functional languages to support interactive animations [43] in Haskell. Also, this line of work has been strongly influenced by asynchronous dataflow languages like Lustre [44]. Recently, RP has gained increasing attention, and several implementations have been developed for various programming languages. We report only the most prominent examples; a complete overview is available in a survey [2].

In the context of functional languages, Fran [43], [45] and Yampa [46] extend Haskell to express time-varying values: *behaviors*, to model continuous values, and *events*, to model discrete changes. Similar to DREAM vars and signals, behaviors are composable and various Haskell operators have been adapted to work with behaviors (more recently, support for streams has been also added to Haskell[5]). Analogous extensions have been proposed for Scheme (FrTime [16]) and Scala (Scala.React [1] and REScala [17]). Similar to DREAM, Frappe [47] is a reactive programming library for Java. It extends the JavaBeans component model [48] introducing event sources (analogous to DREAM vars) and behaviors (analogous to DREAM signals).

Salvaneschi and Mezini [49] discuss the integration of reactive and object-oriented programming. As future work, we plan to explore this integration in more detail within DREAM, focusing on solutions for automated tracking of

[5]https://wiki.haskell.org/Library/Streams

dependencies between method invocations, inheritance, and strategies to optimize the recomputation of values.

Elliott [45] proposes hybrid push/pull-based propagation. The former reevaluates a reactive expression as soon as a change is detected, making the new updated value always available, as in DREAM. The latter postpones the evaluation until it is strictly required, potentially reducing the computational effort. Investigating the benefits of pull-based propagation in combination with consistency guarantees is also a promising direction for future work.

Ramson and Hirschfeld recently proposed Active Expressions [50] as a fundamental primitive to implement different flavours of RP. Other current RP research directions include debugging [51], [52], [53] and application to new domains such as IoT/edge computing and autonomous vehicles [54], [55], [56].

RP was influenced by other paradigms based on dataflow and synchronous propagation of change. Synchronous programming [57], [58] is one of the earliest approaches proposed for the development of reactive systems. It is based on the synchronicity assumption, meaning that reactions are instantaneous and atomic. This assumption simplifies the program, which can be compiled into finite-state automata. Dataflow programming [59] represents entire programs as directed graphs, where nodes represent computations and arcs are dependencies between them. ReactiveML [60] is an extension of a strict ML language with synchronous parallelism to program reactive applications.

Languages for incremental computation [61], [62] allow programs to efficiently re-execute after a small input change and share with RP the problem of maintaining dependencies and propagating updates over the dependency graph.

In contrast to the work presented in this paper, all these solutions target only the local setting.

Consistency Guarantees

Glitch freedom has been introduced in FrTime [6] and implemented in most of the approaches for local RP. This paper further distinguishes single-source and complete glitch freedom and compares the mechanisms to ensure them. Furthermore, it relates glitch freedom to weaker consistency guarantees, such as causal consistency, as well as orthogonal concerns, such as atomicity of propagations.

The most widely adopted glitch free update propagation algorithm [6], [63], [1], [37] separates the dependency graph into layers. The source of changes belongs to layer 0 and all the dependent nodes belong to one layer above their highest layer incoming dependency. During the propagation, all dependent nodes are first added to a priority queue, and then dequeued in layer order to re-compute their value. Concurrent propagations initiated from different sources are not allowed. This approach guarantees complete glitch freedom – the value of a node is re-computed only after all the nodes it depends on (those that belong to a lower layer) have been re-computed. Yet it is not suitable for distributed settings, because it requires a coordination between the nodes to manage the shared priority queue.

In most RP approaches the evaluation order of the dependency graph is non deterministic. SignalJ [64], a reactive extension of Java takes a different approach, defining a precise evaluation oder specified by a formal semantics and enabling a more precise interaction between imperative and reactive code.

Scala.Rx [26] does not offer flexible consistency guarantees but it enables the developers to define their own propagation algorithm programmatically. By default, Scala.Rx ensures glitch freedom by implementing the priority queue mechanism discussed above, but it also offers a parallel version that evaluates all the nodes in the same level concurrently. The mechanisms to ensure single-source and complete glitch freedom that we propose in this paper and implement in DREAM offer a higher degree of parallelism, since they enable nodes at different levels to be evaluated concurrently.

Elm [25] is an FRP language to design GUIs. It ensures glitch freedom while enabling for a higher degree of parallelism during the propagation. In particular, it relies on a central coordinator that broadcasts a message to all sources to start the update process. Each node waits for a message from all the incoming edges before processing any of them. While this approach enables multiple concurrent propagations from different sources, it forces all nodes, including the ones that are not affected by any change, to propagate messages to their dependent nodes. Also, the presence of a central coordinator makes the approach unsuitable for distributed settings.

Rx [65] is a library originally developed for .NET and recently ported to various platforms, including Java and Scala. Rx has received great attention after its adoption in large-scale projects, including content distribution in the Netflix streaming media provider. In contrast to most other approaches to RP, Rx does not provide glitch freedom and events are propagated as soon as they are processed in a node of the dependency graph.

Distributed Reactive Programming Frameworks

The need for DRP with consistency guarantees has been widely recognized in the literature [8], [2], [7], [9] as a way to deal with the complexity of programming distributed systems [66]. Nevertheless, only a few frameworks have addressed the problem of implementing RP abstractions in distributed settings.

In the context of Web programming, Flapjax [37] has been proposed as a reactive programming library for JavaScript. The design of Flapjax is mostly based on FrTime. Flapjax addresses only client side code in the client-server model. The server side of the application can potentially be implemented in a reactive language, but the reactive system on the client and on the server are not aware of each other. As a result there is not guarantee of glitch freedom spanning the whole application [9].

AmbientTalk/R [63] is a reactive extension to AmbientTalk [67], an actor-based language for mobile applications. Similarly to Flapjax, AmbientTalk/R supports DRP but does not ensure consistency guarantees or glitch avoidance. Interestingly, this is a design choice: indeed, since AmbientTalk/R targets mobile peer-to-peer networks where the network topology changes frequently, the cost to ensure consistency guarantees can become unfeasibly high.

Distributed REScala [9] was the first framework to ensure glitch freedom in distributed settings. Since the authors of Distributed REScala compare the SID-UP algorithm used in such framework with Elm, Scala.rx, and Scala.React, we use SID-UP as a baseline to compare the algorithms we implemented in DREAM. As shown in Section 6, the main limitation of SID-UP consists in the impossibility to execute concurrent propagation processes concurrently. Yet, SID-UP supports dynamic updates in the dependency graph, a problem that we plan to address as future work.

Reynders et al. [68] propose a programming model for Web applications that combines DRP and multitier programming, enabling (distributed) signals that belong to the client and to the server to coexist inside the same compilation unit.

Myter et al. [69] propose a DSL for distributed applications that combines DRP reactive abstractions and replication via CRDTs for shared distributed signals. The resulting DRP framework supports applications that may temporarily disconnect from the network ensuring eventual consistency in the value propagation over the dependency graph.

Proença and Baquero [70] provide a formalization of a DRP language for IoT that guarantees glitch freedom as well as more relaxed notions like "glitch freedom with an error margin" and supports decentralized, *pipelined* [9] propagation across the dependency graph.

Event-based Systems

Event-based systems [71] 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.

The problem of efficiently event dispatching in large scale distributed systems has received significant attention in the last decades and led to the development of various infrastructures for event delivery [72], [73], [74].

DREAM builds on top of the REDS distributed dispatching system, which offers a publish-subscribe communication model [75] to *forward* events within an overlay broker network. In this model, clients express interest in events by subscribing to them and publish events that are automatically distributed to all the interested subscribers. The process of associating events to relevant subscriptions is called *matching* and is performed based on a type associated to each events or on the entire event content [75].

In addition to pure event matching and forwarding, Complex Event Processing (CEP) systems [76], [77], [78] provide operators for event *composition* [79], [80]. In particular, they enable the detection of patterns on the content and time relations among events. This approach is complementary to the composition of time-changing values offered in RP. Our previous work provides a detailed comparison [8]. Consistency – specifically, event ordering – guarantees have been studied both in the context of publish-subscribe systems [81] and CEP systems [82]. RP has been used to design the API of *adaptive* CEP systems [83], which provide ways to programmatically configure operator placement at runtime [84]. Several libraries and language extensions have been proposed to support events and event composition as first class language constructs. The most notable examples are C# events, EventJava [85], Ptolemy [86], EScala [87], and JEScala [88].

Stream Processing Systems

Stream Processing (SP) systems [89] provide processing abstractions to deal with streaming data. The typical processing model adopted in SP was first defined in the Stanford STREAM system [90]. It comprises stream-to-relation primitives (windows) to isolate relevant portions of a stream, relational operators to operate on the content of each window, and relation-to-stream operators to convert the result of processing back into a data stream. For instance, a window can isolate the last 100 elements in two incoming streams, a join operator merges the two streams, and a relation-to-stream operators streams only newly generated data.

Another approach is to specify the processing task as a graph of operators [91]: each operator consumes the input streams and produces one or more output streams for other operators. Operators can be either standard or defined programmatically. This model shares some similarities with the dependency graph in RP. However, to the best of our knowledge, consistency guarantees involving glitch freedom and atomicity have not been addressed in this field.

Concerning ordering guarantees, most SP systems assume that the elements in a stream are always received in order and comply with the expected semantics only if this assumption holds. Also, in the case of multiple processing steps, SP systems often require intermediate operators to preserve the ordering when they emit new results [90], [91], [92], [93]. In some cases, SP systems expose the management of ordering guarantees to the developers, forcing them to specify when the incoming data can be safely processed because no more out-of-order information is expected [94].

More recently, several stream processing technologies have been developed for cluster and Cloud environments [95], [96] – a field where scalability and fault tolerance become the key goal. These systems combine task and data parallelism by deploying different operators on different processing cores or machines and by splitting the processing of each stream across multiple operator instances. In this context, Affetti et al. [97] recently studied the consistency for accessing and updating the internal state of operators.

10 Conclusion

Despite the increasing popularity of RP, precise semantics of update propagation in distributed settings has received little attention. In this work, we define the semantics for the propagation of changes in DRP introducing various levels of consistency that constrain the order and the visibility of updates. We propose techniques to ensure such consistency guarantees and we analytically study their overhead in terms of network traffic and propagation latency.

We design and implement DREAM, a DRP framework that supports different consistency levels. Our evaluation compares the empirical overhead of each level in DREAM and shows that supporting multiple consistency levels is a fundamental feature for a DRP framework: Manually implementing a higher level of consistency breaks the design benefits that make RP desirable in the first place.

Acknowledgment

We thank the anonymous reviewers for their detailed analysis of the work and their useful suggestions during the revision process. We also thank Prof. Gianpaolo Cugola for his comments and insights. They significantly helped us to improve the quality of the paper. This research has been partially funded by the European Research Council (grant No. 321217), by the German Research Foundation (DFG) as part of project C2 within the Collaborative Research Center 1053 (MAKI), by DFG grant SA 2918/2-1, and by the AWS Cloud Credits for Research program.

References

[1] I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.react," EPFL, Tech. Rep., 2012.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, 2013.

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

[4] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini, "An empirical study on program comprehension with reactive programming," in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE '14. ACM, 2014, pp. 564–575.

[5] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1125–1143, Dec. 2017.

[6] G. H. Cooper and S. Krishnamurthi, "Embedding dynamic dataflow in a call-by-value language," in *Proceedings of the European Conference on Programming Languages and Systems*, ser. ESOP '06. Springer-Verlag, 2006, pp. 294–308.

[7] G. Salvaneschi, J. Drechsler, and M. Mezini, "Towards distributed reactive programming," in *Proceedings of the International Conference*, ser. COORDINATION '13. Springer-Verlag, 2013, pp. 226–235.

[8] A. Margara and G. Salvaneschi, "Ways to react: Comparing reactive languages and complex event processing," in *In Proceedings of the Workshop on Reactivity, Events and Modularity*, ser. REM '13, 2013.

[9] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini, "Distributed REScala: An update algorithm for distributed reactive programming," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. ACM, 2014, pp. 361–376.

[10] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse, "Reactive control of autonomous drones," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 207–219.

[11] A. Margara and G. Salvaneschi, "We have a DREAM: Distributed reactive programming with consistency guarantees," in *Proceedings of the International Conference on Distributed Event-Based Systems*, ser. DEBS '14. ACM, 2014, pp. 142–153.

[12] F. Myter, C. Scholliers, and W. De Meuter, "Handling partial failures in distributed reactive programming," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2017. New York, NY, USA: ACM, 2017, pp. 1–7.

[13] R. Mogk, L. Baumgärtner, G. Salvaneschi, B. Freisleben, and M. Mira, "Fault-tolerant Reactive Programming," in *32nd European Conference on Object-Oriented Programming (ECOOP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 18:1–18:26.

[14] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys*, vol. 49, no. 1, pp. 19:1–19:34, 2016.

[15] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly, 2017.

[16] K. Burchett, G. H. Cooper, and S. Krishnamurthi, "Lowering: A static optimization technique for transparent functional reactivity," in *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '07. ACM, 2007, pp. 71–80.

[17] G. Salvaneschi, G. Hintz, and M. Mezini, "REScala: Bridging between object-oriented and functional style in reactive applications," in *Proceedings of the International Conference on Modularity*, ser. MODULARITY '14. ACM, 2014, pp. 25–36.

[18] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[19] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[20] M. Raynal, "About logical clocks for distributed systems," *SIGOPS Operating System Review*, vol. 26, no. 1, pp. 41–48, 1992.

[21] D. Mosberger, "Memory consistency models," *SIGOPS Operating System Review*, vol. 27, no. 1, pp. 18–26, 1993.

[22] X. Jia, "A total ordering multicast protocol using propagation trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 617–627, 1995.

[23] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[24] G. Cugola and G. P. Picco, "Reds: A reconfigurable dispatching system," in *Proceedings of the International Workshop on Software Engineering and Middleware*, ser. SEM '06. ACM, 2006, pp. 9–16.

[25] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM, 2013, pp. 411–422.

[26] "Scala.rx." https://github.com/lihaoyi/scala.rx.

[27] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer, "Protopeer: From simulation to live deployment in one step," in *Proceedings of the International Conference on Peer-to-Peer Computing*, ser. P2P '08. IEEE, 2008, pp. 191–192.

[28] P. R. Pietzuch, "Hermes: A scalable event-based middleware," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-590, Jun. 2004. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-590.pdf

[29] A. Margara and G. Cugola, "High performance content-based matching using GPUs," in *Proceedings of the International Conference on Distributed Event-based System*, ser. DEBS '11. ACM, 2011, pp. 183–194.

[30] ——, "High-performance publish-subscribe matching using parallel hardware," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 126–135, 2014.

[31] M. Sadoghi, H. Singh, and H.-A. Jacobsen, "Towards highly parallel event processing through reconfigurable hardware," in *Proceedings of the International Workshop on Data Management on New Hardware*, ser. DaMoN '11. ACM, 2011, pp. 27–32.

[32] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[33] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *Proceedings of the International Conference on Distributed Computing Systems*, ser. ICDCSW '02. IEEE, 2002, pp. 611–618.

[34] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '02. IEEE, 2002, pp. 7–16.

[35] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[36] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015.

[37] 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 *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 1–20.

[38] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[39] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, 1981.

[40] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze, "Disciplined Inconsistency with Consistency Types," in *Proceedings of the Symposium on Cloud Computing*, ser. SoCC '16. ACM, 2016, pp. 279–293.

[41] N. Zaza and N. Nystrom, "Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency," in *Workshop on Programming Models and Languages for Distributed Computing*, ser. PMLDC '16. ACM, 2016, pp. 3:1–3:4.

[42] A. Margara and G. Salvaneschi, "Consistency types for safe and efficient distributed programming," in *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, ser. FTFJP '17. ACM, 2017, pp. 8:1–8:2.

[43] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the International Conference on Functional Programming*, ser. ICFP '97. ACM, 1997, pp. 263–273.

[44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[45] C. M. Elliott, "Push-pull functional reactive programming," in *Proceedings of the Symposium on Haskell*, ser. Haskell '09. ACM, 2009, pp. 25–36.

[46] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming*, ser. AFP '03. Springer, 2003.

[47] A. Courtney, "Frappé: Functional reactive programming in Java," in *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, ser. PADL '01. Springer-Verlag, 2001, pp. 29–44.

[48] D. Blevins, "Overview of the enterprise Javabeans component model," *Component-based Software Engineering*, pp. 589–606, 2001.

[49] G. Salvaneschi and M. Mezini, "Reactive behavior in object-oriented applications: An analysis and a research roadmap," in *Proceedings of the International Conference on Aspect-oriented Software Development*, ser. AOSD '13. ACM, 2013, pp. 37–48.

[50] S. Ramson and R. Hirschfeld, "Active expressions: Basic building blocks for reactive programming," *CoRR*, vol. abs/1703.10859, 2017.

[51] G. Salvaneschi and M. Mezini, "Debugging for reactive programming," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 796–807.

[52] I. Perez and H. Nilsson, "Testing and debugging functional reactive programming," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 2:1–2:27, Aug. 2017.

[53] H. Banken, E. Meijer, and G. Gousios, "Debugging data-flows in reactive programs," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018.

[54] K. Sawada and T. Watanabe, "Emfrp: A functional reactive programming language for small-scale embedded systems," in *Companion Proceedings of the 15th International Conference on Modularity*, ser. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 36–44.

[55] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Vehicle platooning simulations with functional reactive programming," in *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, ser. SCAV'17. New York, NY, USA: ACM, 2017, pp. 43–47.

[56] B. Calus, B. Reynders, D. Devriese, J. Noorman, and F. Piessens, "FRP IoT Modules as a Scala DSL," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2017. New York, NY, USA: ACM, 2017, pp. 15–20.

[57] 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*, vol. 91, no. 1, pp. 64–83, 2003.

[58] F. Sant'Anna, R. Ierusalimschy, and N. Rodriguez, "Structured synchronous reactive programming with céu," in *Proceedings of the 14th International Conference on Modularity*, ser. MODULARITY 2015. New York, NY, USA: ACM, 2015, pp. 29–40.

[59] P. G. Whiting and R. S. Pascoe, "A history of data-flow languages," *Annals of the History of Computing*, vol. 16, no. 4, pp. 38–59, 1994.

[60] L. Mandel, C. Pasteur, and M. Pouzet, "ReactiveML, ten years later," in *Proceedings of 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, Siena, Italy, Jul. 2015.

[61] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster, "Adapton: Composable, demand-driven incremental computation," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 156–166.

[62] D. C. Harkes, D. M. Groenewegen, and E. Visser, "IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 11:1–11:26.

[63] A. L. Carreton, S. Mostinckx, T. Van Cutsem, and W. De Meuter, "Loosely-coupled distributed reactive programming in mobile ad hoc networks," in *Proceedings of the International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Springer-Verlag, 2010, pp. 41–60.

[64] T. Kamina and T. Aotani, "Harmonizing signals and events with a lightweight extension to java," *The Art, Science, and Engineering of Programming*, vol. 2, 2018.

[65] E. Meijer, "Your mouse is a database," *Queue*, vol. 10, no. 3, pp. 20:20–20:33, 2012.

[66] I. Kuraj and A. Solar-Lezama, "Leveraging Sequential Computation for Programming Efficient and Reliable Distributed Systems," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 71. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 7:1–7:15.

[67] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming in Ambienttalk," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP'06. Springer-Verlag, 2006, pp. 230–254.

[68] 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*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 55–68.

[69] F. Myter, T. Coppieters, C. Scholliers, and W. De Meuter, "I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming," in *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2016. New York, NY, USA: ACM, 2016, pp. 1–8.

[70] J. Proença and C. Baquero, "Quality-aware reactive programming for the internet of things," in *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Dastani and M. Sirjani, Eds., vol. 10522. Springer, 2017, pp. 180–195.

[71] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Springer, 2006.

[72] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *Proceedings of the International Conference on Distributed Computing Systems*, ser. ICDCSW '02. IEEE, 2002, pp. 611–618.

[73] E. Fidler, H. A. Jacobsen, G. Li, and S. Mankovski, "The padres distributed publish/subscribe system," in *Proceedings of the International Conference on Feature Interactions in Telecommunications and Software Systems*, ser. FIW '05, 2005, pp. 12–30.

[74] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "A peer-to-peer approach to content-based publish/subscribe," in *Proceedings of the International Workshop on Distributed Event-based Systems*, ser. DEBS '03. ACM, 2003, pp. 1–8.

[75] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[76] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, 2012.

[77] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[78] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Co., 2010.

[79] P. R. Pietzuch, B. Shand, and J. Bacon, "A framework for event composition in distributed systems," in *Proceedings of the Interna-*

tional Conference on Middleware, ser. Middleware '03. Springer-Verlag, 2003, pp. 62–82.

[80] G. Cugola and A. Margara, "Tesla: A formally defined event specification language," in *Proceedings of the International Conference on Distributed Event-Based Systems*, ser. DEBS '10. ACM, 2010, pp. 50–61.

[81] K. Zhang, V. Muthusamy, and H.-A. Jacobsen, "Total order in content-based publish/subscribe systems," in *Proceedings of the International Conference on Distributed Computing Systems*, ser. ICDCS '12. IEEE, 2012, pp. 335–344.

[82] R. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *Biennial Conference on Innovative Data Systems Research*, ser. CIDR '07, 2007, pp. 363–374.

[83] P. Weisenburger, M. Luthra, B. Koldehofe, and G. Salvaneschi, "Quality-aware runtime adaptation in complex event processing," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 140–151.

[84] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "TCEP: adapting to dynamic user environment by enabling transitions between operator placement mechanisms," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '18. New York, NY, USA: ACM, 2018, pp. 378–381.

[85] P. Eugster and K. R. Jayaram, "EventJava: An extension of Java for event correlation," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 570–594.

[86] H. Rajan and G. T. Leavens, "Ptolemy: A language with quantified, typed events," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP '08. Springer-Verlag, 2008, pp. 155–179.

[87] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé, "EScala: Modular event-driven object interactions in scala," in *Proceedings of the International Conference on Aspect-oriented Software Development*, ser. AOSD '11. ACM, 2011, pp. 227–240.

[88] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé, "Jescala: Modular coordination with declarative events and joins," in *Proceedings of the International Conference on Modularity*, ser. MODULARITY '14. ACM, 2014, pp. 205–216.

[89] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Symposium on Principles of Database Systems*, ser. PODS '02. ACM, 2002, pp. 1–16.

[90] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: The stanford stream data manager (demonstration description)," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '03. ACM, 2003, pp. 665–665.

[91] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[92] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '03. ACM, 2003, pp. 647–651.

[93] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," in *Proceedings of the International Conference on Data Engineering*, ser. ICDE '08. IEEE Computer Society, 2008, pp. 804–813.

[94] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[95] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[96] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bullettin*, vol. 38, no. 4, pp. 28–38, 2015.

[97] L. Affetti, A. Margara, and G. Cugola, "Flowdb: Integrating stream processing and consistent state management," in *Proceedings of the International Conference on Distributed and Event-based Systems*, ser. DEBS '17. ACM, 2017, pp. 134–145.