We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees

Alessandro Margara, Guido Salvaneschi

Presented by Wilfried Daniels

Introduction

- Designing, implementing and maintaining reactive systems is difficult
  - Asynchronous callbacks
  - Hard to trace/understand control flow

➔ Solution: Reactive Programming

Introduction

- Key concepts:
  - time-varying values
  - tracking of dependencies
  - automatic propagation of changes

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

Imperative

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

Reactive

Introduction

- Advantages vs. classic event-based arch:
  - No explicit update logic
  - Declarative specification of dependencies
  - Runtime manages correct propagation (e.g. glitch freeness/consistency)

- This work focuses on distributed reactive programming (DRP)

Introduction

- Previous DRP solutions do not guarantee distributed consistency (only local)

- This paper presents DREAM , a **Distributed REActive Middleware** with three different levels of consistency guarantees

Background and Motivation

- Motivation for different levels of consistency
- Running example: financial application system

```
var marketIndex = InputModule.getMarketIndex()
var stockOpts = InputModule.getStockOpts()
var news = InputModule.getNews()

// Forecasts according to different models
var f1 := Model1.compute(marketIndex,stockOpts)
var f2 := Model2.compute(marketIndex,stockOpts)
var f3 := Model3.compute(marketIndex,news)

var gui := Display.show(f1,f2,f3)

var financialAlert := ((f1+f2+f3)/3) < MAX
if (financialAlert) decrease(stockOpts)

var financialAlert_n := computeAlert_n(f1,f2,f3)
if (financialAlert_n) adjust_n(stockOpts)
```

V1

V2

V3

Observable time-varying variables

Dependent **Reactive expressions**

Reactive expressions resulting in 3 alternative outputs, each requiring different consistency guarantees

Background and Motivation

- Variant 1: Smartphone app
  - Just displays output of 3 models
  - No consistency required

```
var gui := Display.show(f1,f2,f3)
```

V1

- Variant 2: Models aggregator
  - Aggregates output of 3 models
  - Undertakes action when below threshold

```
var financialAlert := ((f1+f2+f3)/3) < MAX
if (financialAlert) decrease(stockOpts)
```

V2

Background and Motivation

- Variant 2: Models aggregator
  - Requires glitch freedom
  - Assume initially **f1**:110, **f2**:95, **f3**:99 with **MAX**:100
  - New **marketIndex**: *all* models recalculate.
  - Model **f1** finishes first with **f1**: 90
    - → STOCKS DECREASED (GLITCH!)
  - Other models finish: **f2**:111, **f3**:103

```
var financialAlert := ((f1+f2+f3)/3) < MAX
if (financialAlert) decrease(stockOpts)
```

V2

Background and Motivation

- Variant 3: Multiple aggregators
  - **f1**, **f2**, **f3** are dispatched to *n* aggregators, that work autonomously
  - In case of deviating behaviour, any aggregator can adjust stockOpts
  - No glitch freedom required, but every single aggregator needs to see **f1**, **f2** and **f3** change in the same order

```
var financialAlert_n := computeAlert_n(f1,f2,f3)
if (financialAlert_n) adjust_n(stockOpts)
```

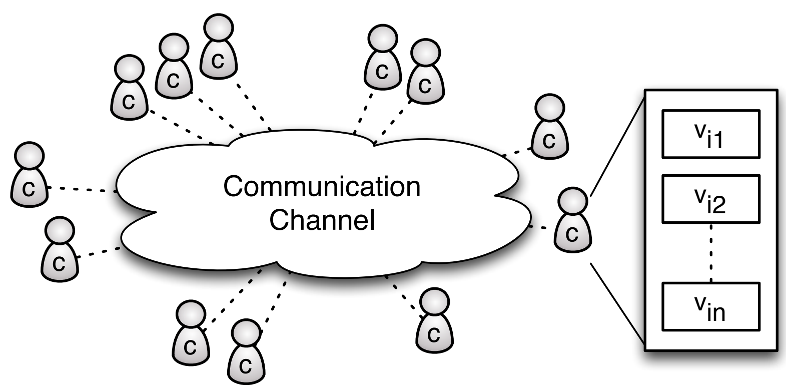
V3

A model for DRP

- Formal definition of DRP system architecture/ consistency guarantees
- **Components**: networked nodes in system

$$c_1 \ \ldots \ c_n$$

- **Variables**: state of component c_i is represented by $V_i = \{v_{i1} : \tau_{i1} \ \ldots \ v_{im} : \tau_{im}\}$



A model for DRP

- Besides traditional *imperative* variables, *reactive* and *observable* variables are defined
- **Reactive**: variable that is automatically updated based on reactive expression
- **Observable**: continuously changing var that is used to build expressions. Local or Global.
- e.g. stock market:

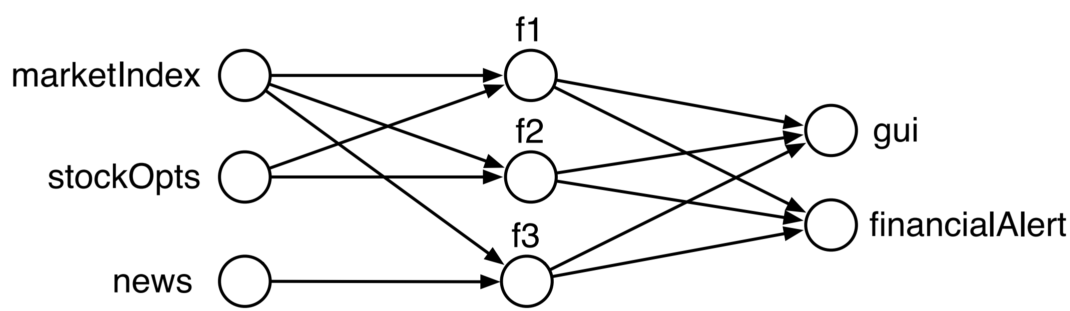
```
f3 := Model3.compute(marketIndex,news)
```

Reactive (+ observable) variable

Observable variables

A model for DRP

- **Dependency Graph:**
  - Directed graph $D = \{V, E\}$, where V is the set of all observable/reactive variables and E is the set of all edges that connect directly depending variables
  - E.g. stock market for Variant 1 + 2:



A model for DRP

- **Events:**
  - *Write* event: $w_x(v)$
    - Occurs when value x is written to variable v
  - *Read* event: $r_x(v)$
    - Occurs when value x is read from variable v
  - *Update* event: $u(S, w_x(v)),\ S = \{w_{y1}(v_1) \ldots w_{yn}(v_n)\}$
    - Depending variable v is reactively update with value x due to the write events contained in the set S

A model for DRP

- **Consistency Guarantees**
  - **Exactly once delivery:** ensures that, in absence of failure, the communication channel does not lose or duplicate an update. More formally:

If $w_x(v)$ occurs, then $u(S_i, w_y(v_i)),\ w_x(v)\ \in\ S_i$ occurs exactly once.

A model for DRP

- **Consistency Guarantees**
  - **FIFO ordering:** changes to a a variable v in a component c are propagated to depending reactive expressions in the same order they occur in c. More formally:

> $\forall v_i, v_j$, such that v_j depends on v_i, if $w_{x1}(v_i)$ occurs before $w_{x2}(v_i)$, then $u(S_1),\ w_{x1}(v_i) \in S_1$ occurs before $u(S_2), w_{x2}(v_i) \in S_2$

A model for DRP

- **Consistency Guarantees**
  - **Causal ordering:** ensures that events that are causally connected occur in every component in the same order. More formally:

> We define a *happened before* ($\longrightarrow$) partial order relation:
> - If two events e_1, e_2, occur in the same process, then $e_1 \rightarrow e_2$ if and only if e_1 occurs before e_2
> - If $e_1 = w_x(v_i) \text{ and } e_2 = u(S_i, w_y(v_j)),\ w_x(v_i) \in S_i$, then $e_1 \rightarrow e_2$ (a write happens before an update depending on it)
> - If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$ (transitivity)

  - No guarantees are made for events that are not causally connected!

A model for DRP

- **Consistency Guarantees**
  - **Glitch freedom:** no partial updates due to propagation delays. More formally:

Consider the set V_d, containing all observable variables a reactive variable v depends on. Let us call $V_{d1} \subseteq V_d$ the set of variables that depend directly or indirectly from a variable v_1. The update $u(S, w_x(v))$ is a *partial* update if $S \subset V_{d1}$. A glitch free system does not have partial updates.

A model for DRP

- **Consistency Guarantees**
  - **Atomic consistency:** ensures that: (i) the system provides FIFO ordering, and (ii) every write event to an observable variable is atomically propagated to all (in)directly depending reactive variables. More formally:

> All the update events $u(S_i, w_y(v_i))$ triggered (directly or indirectly) by $w_x(v)$ are executed as a single operation

  - This is stricter than glitch freedom

DREAM: API

- DREAM is entirely written in Java
- Observable variables → observable objects
  - Inherit from `Observable` abstract class
  - All non-void methods: *observable* methods
  - Generic method *m* that potentially changes return value of observable method *obm*: *m* impacts *obm*
  - Impacts should be known by runtime

    → Java Annotations

DREAM: API

- Example of observable class representing an integer:

```java
public class ObservableInteger extends Observable {
  private int val;

  // Constructors ...

  @ImpactsOn(methods = { "get" })
  public final void set(int val) {
    this.val = val;
  }

  public final int get() {
    return val;
  }
}
```

DREAM: API

- Reactive variables → Reactive objects
- Created by using the `ReactiveFactory` class
  - Parses reactive expressions (strings with ANTLR)
  - Reactive objects can be observable (optional)
- Naming space:
  - Unique name: `c.obj.obm` for observable method `obm` of object `obj` in component `c`
  - For local objects: `obj.obm`

DREAM: API

- Example:

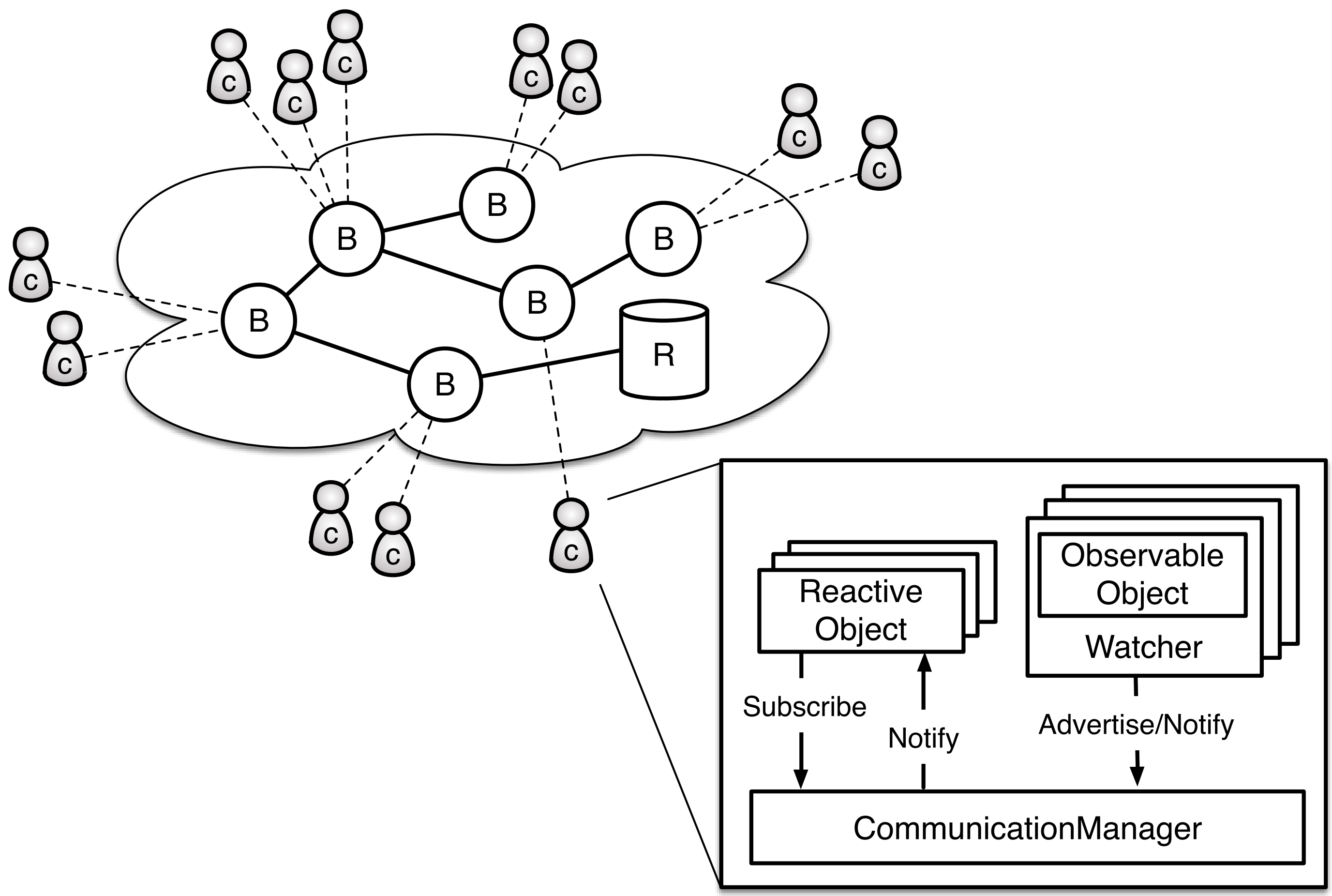
```
// Component c1
ObservableInteger obInt =
  new ObservableInteger("obInt1", 1, LOCAL);
ObservableString obStr1 =
  new ObservableString("obStr1", "a", GLOBAL);
ObservableString obStr2 = ...

// Component c2
ReactiveInteger rInt = ReactiveFactory.
  getInteger("obInt.get()*2");
ReactiveString rStr = ReactiveFactory.
  getString("obStr1.get()+obStr2.get()");
while(true){
  System.out.println(rStr.get())
  Thread.sleep(500)
}

// Component c3
ReactiveInteger strLen =
  ReactiveFactory.getObservableInteger
  ("c1.obString1.get().length()", "obString1Len");
```

DREAM: Implementation

- Architecture consists of two parts:
  - A client library on every component
  - A distributed event-based infrastructure, consisting of *brokers*
- Brokers form an acyclic overlay network, offering communication between components
- Optional registry for persistence

DREAM: Implementation

- Architecture overview



DREAM: Implementation

- **Pub-Sub Communication:**

  Clients register with brokers through 3 primitives:

  – `advertise(c,obj,obm)`: used by c if it has a globally observable method `obj.obm()`

  – `subscribe(c,obj,obm)`: used to register a component that has a reactive expression containing `c.obj.obm()`

  – `notify(c,obj,obm,val)`: used by c when `obj.obm()` has a new value `val`

DREAM: Implementation

- **Clients**
  - `CommunicationManager`:
    - Proxy for global communication
    - Manage local communication
  - Observable objects:
    - Have `Watcher` code woven in through AOP
    - `Watcher` interacts with `CommunicationManager to`:
      1. Advertise new objects through `advertise(c,obj,obm)`
      2. Detect changes to observables and propagate them out through `notify(c,obj,obm,val)`

DREAM: Implementation

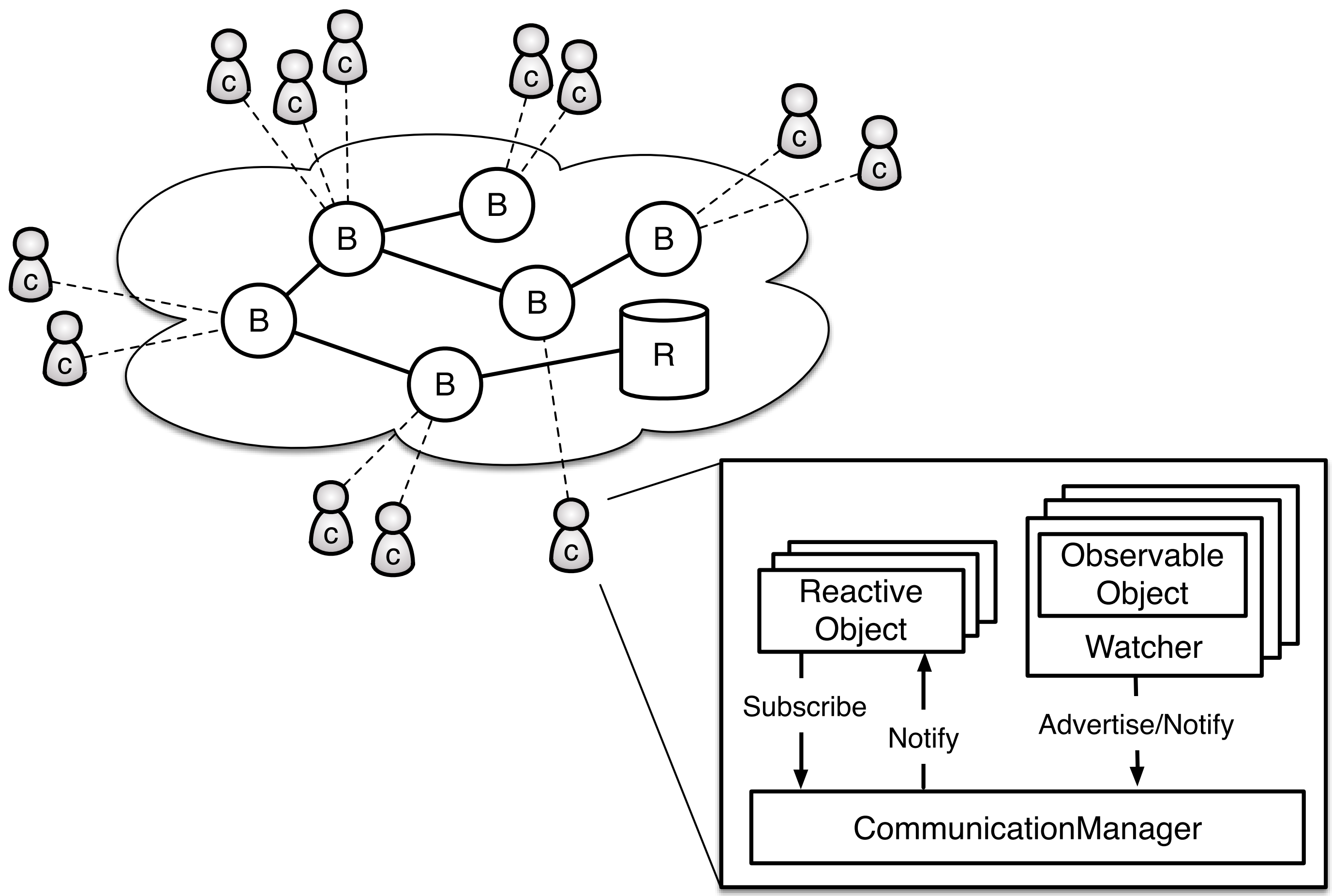
- **Clients**
  - Reactive Objects:
    - When instantiated, for all relevant observable methods

      → `subscribe(c,obj,obm)` with `CommunicationManager`

    - When new values available, notification from `CommunicationManager`

DREAM: Implementation

- Architecture overview



DREAMS: Implementation

- **Brokers**

  Run REDS event dispatching

  – Brokers are connected in acyclic graph

  – Advertisements are propagated through graph + stored by all brokers, remembering next hop

  – When a broker receives a subscription, store in table and forward to next hop (retrace path of advertisements)

DREAMS: Implementation

- **Consistency Guarantees**
  - Causal ordering:
    - Use point to point TCP for broker-broker and client-broker communication
    - Use single thread for FIFO event processing

→ These 2 properties with an acyclic topology are sufficient for causal ordering

DREAMS: Implementation

- **Consistency Guarantees**
  - Glitch freedom:
    - New reactive object: push propagate expression to *all* brokers → each broker has dependency graph
    - When a chain of operations is triggered, always include the original write event that caused it in communications
    - Local communication *has* to go through a broker as well to ensure glitch freedom

    → This information is enough for the brokers to schedule propagation in a way that avoids partial updates

DREAMS: Implementation

- **Consistency Guarantees**
  - Atomic ordering:
    - Adds centralized Ticket Granting Service (TGS)
    - When a write event occurs, *all* it's directly and indirectly dependent reactive expressions are reevaluated atomically (no other write operations)
    - On write: get ticket, wait in line and be served one at a time

    → This entails glitch freedom and is an even stronger consistency guarantee

Evaluation

- Twofold:
  1. Large scale emulation: Cost of DRP protocols with different levels of consistency guarantees/ varying parameters. KPIs:
     - Average propagation delay (ms)
     - Network wide traffic throughput (KB/s)
  2. Real-world runtime overheads

Evaluation

- Default values for emulation:

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

Evaluation

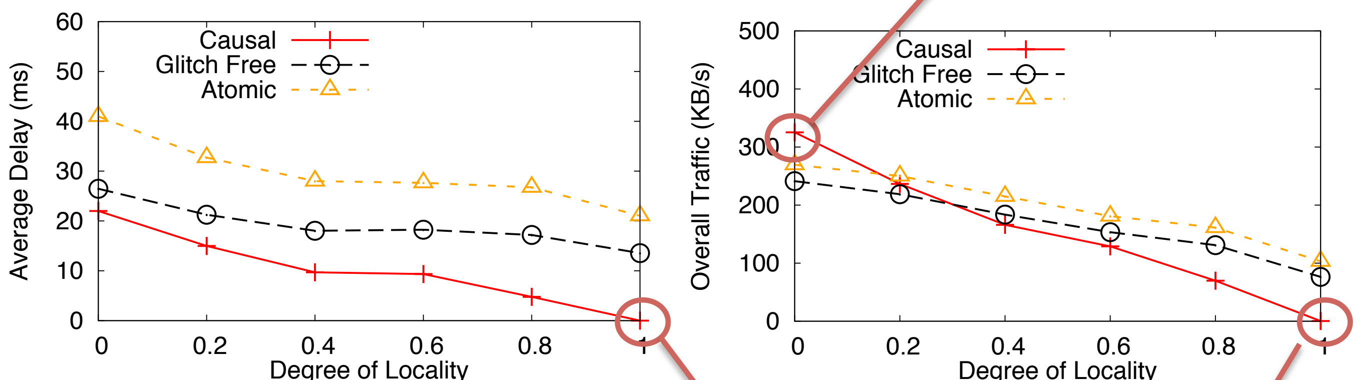
- **Advantages of distribution**
  – 1 broker vs. 10 brokers
  – Causal: no big impact – mainly due to locality
  – Glitch free: *all* propagation through broker
    → Having multiple brokers helps
  – Atomic: adds TGS delay + traffic
    → Same advantages when multiple brokers

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

Evaluation

- **Locality of expressions**
  - General trend: locality cuts costs

Completely remote + causal = glitches!



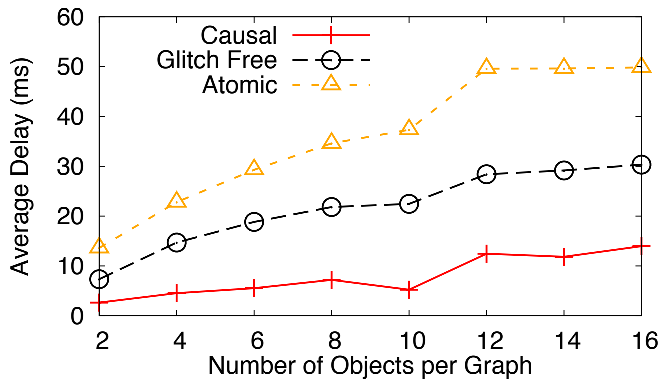
(a) Delay (b) Traffic

Completely local + causal = 0 costs

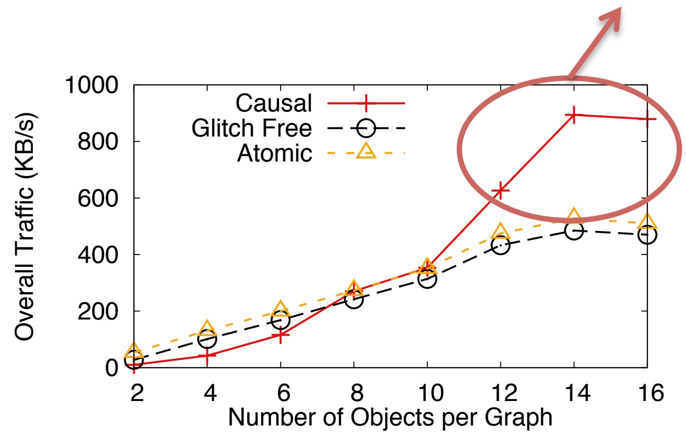
Evaluation

- **Size of reactive graphs**
  – General trend: large reactive graphs increase costs

Long chains of reactive vars + causal = glitches!



(a) Delay

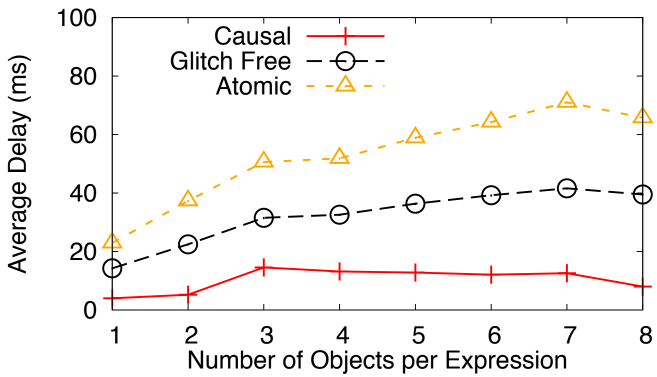


(b) Traffic

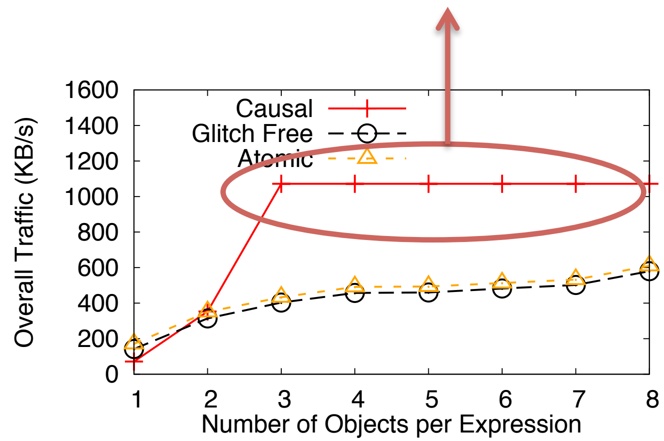
Evaluation

- **Size of expressions**
  - General trend: bigger expressions increase costs

More vars/expression + causal = glitches!



(a) Delay



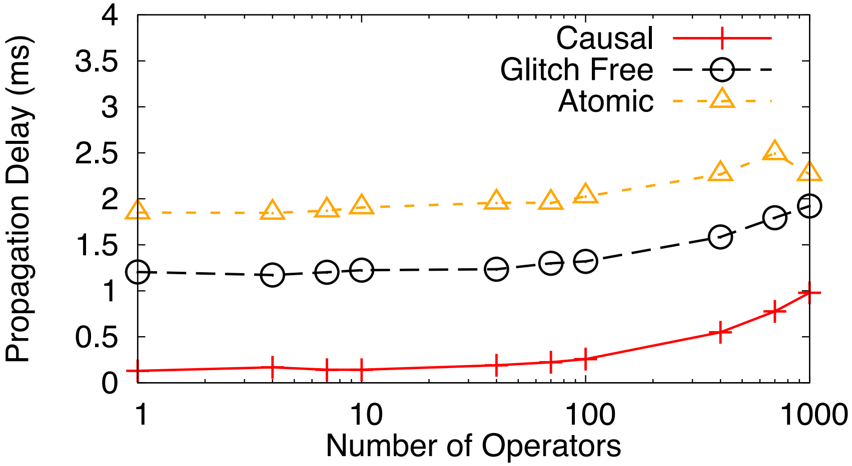
(b) Traffic

Evaluation

- **Runtime overheads**
  - Overheads consisting of:
    - Intercepting a method call
    - Serializing/deserializing
    - Propagating the change
    - Evaluating reactive expression
  - Local scenario: two clients and a broker on 1 machine, with increasing expression length

Evaluation

- **Runtime overheads**
  – Conclusion: runtime overheads are minimal



Conclusion

- Key contributions:
  - First abstract model of DRP/formalizing consistency constraints
  - DREAM: a first DRP middleware supporting 3 propagation semantics
  - A thorough evaluation of the costs

Conclusion

- Future work:
  - A glitch free protocol that takes advantage of locality
  - Robustness in case of node failure
  - More complex expressions (time series and sequence of changes)
  - Different evaluation strategies (lazy, incremental) to improve efficiency
  - More real applications