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

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

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

Imperative

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

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

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

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

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

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

Observable time-varying variables

Dependent Reactive expressions

V1 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

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

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

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

V1

V2
Background and Motivation

• Variant 2: Models aggregator
  – Requires glitch freedom
  – Assume initially \( f_1: 110, f_2: 95, f_3: 99 \) with \( \text{MAX}: 100 \)
  – New \text{marketIndex}: \text{all} models recalculate.
  – Model \( f_1 \) finishes first with \( f_1: 90 \)
    \( \rightarrow \text{STOCKS DECREASED (GLITCH!)} \)
  – Other models finish: \( f_2: 111, f_3: 103 \)

```java
var financialAlert := ((f1+f2+f3)/3) < \text{MAX}
if (financialAlert) decrease(stockOpts)
```

V2
Background and Motivation

• Variant 3: Multiple aggregators
  – \( f_1, f_2, f_3 \) 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 \( f_1, f_2 \) and \( f_3 \) change in the same order

\[
\begin{align*}
\text{var} \ & \text{financialAlert}_n := \text{computeAlert}_n(f_1,f_2,f_3) \\
\text{if (financialAlert}_n) & \text{adjust}_n(\text{stockOpts})
\end{align*}
\]
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:
  \[ f3 := \text{Model3.compute(marketIndex, news)} \]
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 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* (→) 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 \( e_1 = w_x(v_i) \) 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:

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

![Diagram of DREAM architecture](image)
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
      \[ \rightarrow \text{subscribe}(c, \text{obj}, \text{obm}) \text{ with CommunicationManager} \]
    • 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:

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

<table>
<thead>
<tr>
<th></th>
<th>Delay (ms)</th>
<th>Traffic (KB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causal</td>
<td>4.77</td>
<td>4.76</td>
</tr>
<tr>
<td>Glitch free</td>
<td>29.53</td>
<td>17.18</td>
</tr>
<tr>
<td>Atomic</td>
<td>53.41</td>
<td>26.75</td>
</tr>
</tbody>
</table>
Evaluation

• Locality of expressions
  – General trend: locality cuts costs

![Graphs showing delay and traffic vs degree of locality]

(a) Delay
(b) Traffic

Completely local + causal = 0 costs
Completely remote + causal = glitches!
Evaluation

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

Long chains of reactive vars + causal = glitches!
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