QUARP: Quality Aware Reactive Programming for the IoT

José Proença & Carlos Baquero

FSEN 2017
Quality Aware Reactive Programming for the IoT

1. Context: WSN applications

2. Motivation: RP challenges

3. Solution: use thresholds

slides: http://quarp.proenca.org
IoT application
(a WSN perspective)
IoT application
(a WSN perspective)

smart office

Coffee Control
- interval
- actuatorPin

RFID Reader
- UARTChannel
- baudrate

Coffee Manager
- webservice
- timeout

RFID Checker
- webservice
- ACL

Screen Manager
- webservice
- timeLock

Motion Det.
- sampleRate
- listenPin

Motion Agg.
- interval
- timeout

Screen Control
- interval
- actuatorPin

Motion Reporter
- webservice

Button Sensor
- listenPin

Motion Reporter
- webservice

Screen Manager
- webservice
- timeLock

Screen Control
- interval
- actuatorPin

Coffee Manager
- webservice
- timeout

Coffee Control
- interval
- actuatorPin

Motion Agg.
- interval
- timeout

Motion Det.
- sampleRate
- listenPin

Motion Reporter
- webservice

Button Sensor
- listenPin

RFID Checker
- webservice
- ACL

slides:  http://quarp.proenca.org
IoT application (a WSN perspective)

smart office

 Middleware

<events>

Actuators

Sensors

Coffee Control
- interval
- actuatorPin

RFID Reader
- UARTChannel
- baudrate

RFID Checker
- webservice
- ACL

Motion Agg.
- interval
- timeout

Motion Det.
- sampleRate
- listenPin

Motion Reporter
- webservice

Screen Manager
- webservice
- timeLock

Screen Control
- interval
- actuatorPin

Button Sensor
- listenPin

Coffee Manager
- webservice
- timeout

Screen Manager
- webservice
- ACL

Middleware

slides: http://quarp.proenca.org
IoT application (a WSN perspective)

Need to manage the growing complexity of the Internet of Everything
Quality Aware Reactive Programming for the IoT

1. Context: WSN applications

2. Motivation: RP challenges

3. Solution: use thresholds

(Distributed)
Reactive programming

- for event-driven and interactive applications
- express time-varying values
- automatically manage dependencies between such values
- abstract over time management
- like spreadsheets: change 1 cell => others are recalculated

e.g., GUls, web-apps
Example

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

...
Example

\[
\begin{align*}
\text{var1} &= 1 \\
\text{var2} &= 2 \\
\text{var3} &= \text{var1} + \text{var2}
\end{align*}
\]

Stream \textbf{s1} = \text{new Stream(“1”)};
Stream \textbf{s2} = \text{new Stream(“2”)};
Stream \textbf{s3} = \text{Stream.add(s1,s2)};
Challenges

1. **Push vs. Pull behaviour**
2. **Order of evaluation**
3. **“Lifting” operations**
4. **Distribution**

Avoid “glitches”
Glitches

“Momentary view of inconsistent data”

![Diagram of dataflow graph]

```
var1 = 1
var2 = var1 * 1
var3 = var1 + var2
```
Glitches

Also, an efficient reactive implementation should avoid unnecessary recomputations. Most recent reactive implementations achieve glitch avoidance in reactive programs running on a single computer, but not in distributed reactive programs. Avoiding glitches in a distributing setting is not straightforward because of network failures, delays and lack of a global clock. This is a potential sweet spot for future research.

3.4. Lifting Operations

Lifting serves a dual purpose: it transforms a function's type signature (both the types of its arguments and its return type) and it registers a dependency graph in the behaviours is known as lifting.

A Survey on Reactive Programming:7

Glitches: Momentary view of inconsistent program state and recomputation.

1st Calculate ‘+’

2

var1

1

* var2

1

+ var3

2

Glitches

Change to “2”

1

* var2

1

+ var3

2

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3

var1

1

* var2

1

+ var3

3
Glitches

2

var1

1

*

1

var2

+

2

var3

2nd Calculate *

2

var1

1

*

2

var2

+

3

var3

Change to “2”

WRONG!

slides:  http://quarp.proenca.org
Glitches

1. Change to “2”

2
var1

* 

1
var2

+ 

2
var3

3rd REcalculate +

2
var1

* 

2
var2

+ 

4
var3

OK (now!)
that take multiple arguments is trivial. To take a single behaviour argument for the sake of brevity, generalising to functions, we assume the application's dataflow graph. In the following definitions, we assume functions that are lifted, i.e., they take a single behaviour argument and register a dependency graph in the application's dataflow graph. A lifted function is known as a behaviour function.

In the reactive programming literature, the conversion of an ordinary operator to a variant that can operate on behaviours is known as lifting. When reactive programming is embedded in host languages (either as a library or as a language extension), existing language operators (e.g., +, *) and user-defined functions must be converted to operate on behaviours. When reactive programming is embedded in host languages, existing language operators (e.g., +, *) and user-defined functions must be converted to operate on behaviours. In the reactive programming language extension, existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours. When reactive programming is embedded in host languages, existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours.

3.4. Lifting Operations

Lifting serves a dual purpose: it transforms a function's type signature (both the types of its arguments and its return type) and it registers a dependency graph in the application's dataflow graph. In such a case, the values for var1, var2, and var3 are

1. Change to “2”
2. Calculate

value they depend on is updated to a new value that is the same as the previous value. Taking the same example above, suppose the value of var1 is changed to 2. The values for var2 and var3 are subsequently updated to 1 and 2, respectively. Dependent computations need not be recomputed if the value of var1 remains unchanged. Dependent computations need not be recomputed if the value of var1 remains unchanged.

When reactive programming is embedded in host languages (either as a library or as a language extension), existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours. When reactive programming is embedded in host languages, existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours. In such a case, the values for var1, var2, and var3 are

Also, an efficient reactive implementation should avoid unnecessary recomputations. Most recent reactive implementations achieve glitch avoidance in reactive programs. A glitch is a momentary view of inconsistent program state and recomputation. Glitches in distributed systems are not straightforward because of network failures. This is a potential sweet spot for future research.

When reactive programming is embedded in host languages (either as a library or as a language extension), existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours. When reactive programming is embedded in host languages, existing language operators (e.g., +, *) and user-defined functions are converted to operate on behaviours. In such a case, the values for var1, var2, and var3 are

Lifting serves a dual purpose: it transforms a function's type signature (both the types of its arguments and its return type) and it registers a dependency graph in the application's dataflow graph. In such a case, the values for var1, var2, and var3 are

Glitches: Momentary view of inconsistent program state and recomputation. Glitches in distributed systems are not straightforward because of network failures. This is a potential sweet spot for future research.
Lifting serves a dual purpose: it transforms a function’s type signature (both the types of its arguments and its return type) and it registers a dependency graph in the literature the conversion of an ordinary operator to a variant that can operate on behaviour methods must be converted to operate on behaviours. In the reactive programming

### 3.4. Lifting Operations

Glitches: Momentary view of inconsistent program state and recomputation.

1. Change to “2”

2. Calculate

3. Recalculate

The diagram illustrates an example of how lifting operations work in reactive programming. When reactive programming is embedded in host languages (either as a library or as a language extension), glitch avoidance in reactive programs running on a single computer, but not in distributed reactive programs. Avoiding glitches in a distributed setting is not straightforward because of network failures, required robustness, and the need to ensure glitch freedom.

Some recent reactive implementations achieve glitch avoidance in reactive programs, thus ensuring that an expression is always evaluated after all its dependents have been evaluated.

When reactive programming is embedded in host languages (either as a library or as a language extension), avoiding glitches in a distributed setting is not straightforward because of network failures, required robustness, and the need to ensure glitch freedom.

The distributed view of the program state is shown in the diagram, with the values for `var1`, `var2`, and `var3` updated as follows:

- `var1 = 1` is initially.
- `var2 = 1` is initially.
- `var3 = 3` is initially.

**Sequence of Events**:

1. **Change to “2”**
   - `var1` changes to `2`.
2. **Calculate**
   - The expression `var1 * var2 + var3` is evaluated.
3. **Recalculate**
   - The expression `var1 * var2 + var3` is recalculated, but it is wrong because the value of `var1` was changed without updating the expression.
4. **Wrong**
   - The correct value should be recalculated after all dependents have been evaluated.

The diagram illustrates the correct process of updating the values and recalculating the expression after all dependents have been evaluated.

### Slides

[http://quarp.proenca.org](http://quarp.proenca.org)
Distributed Reactive Programming

Distributed REScala: An Update Algorithm for Distributed Reactive Programming

Joscha Drechsler, Guido Salvaneschi
Technische Universität Darmstadt, Germany
<lastname>@cs.tu-darmstadt.de

Ragnar Mogk
Technische Universität Darmstadt, Germany
ragnar.mogk@stud.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt, Germany; Lancaster University, UK
mezini@cs.tu-darmstadt.de

Abstract
Reactive programming improves the design of reactive applications by relocating the logic for managing dependencies...
DRP: minimise overall time

Figure 7. Visual comparison of update propagation with different algorithms.

4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronisation overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do

count steps

count messages

Scala.React

Scala.Rx

ELM

SID-UP

slides: http://quarp.proenca.org
DRP: minimise overall time

4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not have any bold edges, indicating that they are not updating at that moment.
DRP for the IoT

- Operates in rounds (no pipelining)
- Extra coordination

- Minimum coordination
- Maximum concurrency
- Data *can* be lost

Avoid glitches (and similar probs.)
Quality Aware
Reactive Programming
for the IoT
Reactive Programming with Failure

Fig. 1. Application that reacts to sensor values to either notify to close the window or to produce a feels-like value.

Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs. Section 5 discusses the key advantages and disadvantages of our approach with respect to existing approaches to distributed reactive programming. Finally, Sections 6 and 7 present related work and main conclusions, respectively.

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of: data sources, observers, and mixed components. The data sources \( t_1, t_2 \) represent temperature sensors, \( h_1, h_2 \) represent humidity sensors, \( \text{wind} \) represents a wind sensor, \( \text{wdw} \) the open/closed status of a window, and \( w_1, w_2 \) produces weights used to average sensor values. The \( \text{avg}_t \) and \( \text{avg}_h \) components calculate the weighted averages of temperature and humidity, respectively. Finally, the observers \( \text{closeWindow} \) and \( \text{feelsLike} \) are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

Glitches. A glitch can occur, for example, if \( w_1, w_2 \) produces a value, triggering \( \text{avg}_t \) and \( \text{avg}_h \) to recalculate the averages, and later \( \text{feelsLike} \) updates its value after receiving a new value from \( \text{avg}_t \) but before receiving from \( \text{avg}_h \).

Timestamps. Alternatively, the \( \text{feelsLike} \) observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.
“Good enough” inputs

1. Data arrives
2. Input data is “good enough”?
3. Publish if “yes”
“Good enough” inputs

Fig. 1. Application that reacts to sensor values to either notify to close the window or to produce a feels-like value.

Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs. Section 5 discusses the key advantages and disadvantages of our approach with respect to existing approaches to distributed reactive programming. Finally, Sections 6 and 7 present related work and main conclusions, respectively.

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:

- Data sources ($t_1$, $t_2$, $h_1$, $h_2$)
- Wind sensor ($w$)
- Sensor weights ($w_{1,2}$)
- Wind sensor status ($wdw$)
- Temperature and humidity component ($avg_t$, $avg_h$)
- Observers ($closeWindow$, $feelsLike$)

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

Glitches. A glitch can occur, for example, if $w_{1,2}$ produces a value, triggering $avg_t$ and $avg_h$ to recalculate the averages, and later $feelsLike$ updates its value after receiving a new value from $avg_t$ but before receiving from $avg_h$.

Timestamps. Alternatively, the $feelsLike$ observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.

"Good enough" inputs

- $t_1$
- $t_2$
- $w_{1,2}$
- $wdw$
- $avg_t$
- $avg_h$

Not enough input data

- $25°$
- $0.35, 0.65$
“Good enough” inputs

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:
- data sources ($t_1, t_2$),
- observers ($h_1, h_2$),
- and mixed components ($\text{wind}$, $w_1, 2$).

The data sources $t_1, t_2$ represent temperature sensors, $h_1, h_2$ represent humidity sensors, $\text{wind}$ represents a wind sensor, and $w_1, 2$ produces weights used to average sensor values.

The $\text{avg}_t$ and $\text{avg}_h$ components calculate the weighted averages of temperature and humidity, respectively.

Finally, the observers $\text{closeWindow}$ and $\text{feelsLike}$ are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

Glitches. A glitch can occur, for example, if $w_1, 2$ produces a value, triggering $\text{avg}_t$ and $\text{avg}_h$ to recalculate the averages, and later $\text{feelsLike}$ updates its value after receiving a new value from $\text{avg}_t$ but before receiving from $\text{avg}_h$.

Timestamps. Alternatively, the $\text{feelsLike}$ observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.

"Good enough" inputs...?
“Good enough” inputs

- $t_1$
- $t_2$
- $w_{1,2}$
- $w_{dw}$
- $avg_t$

Values and weights:
- $25^\circ$
- $14^\circ$
- $0.35, 0.65$

Decisions:
- not enough input data
- seems ok...?
- no glitches?
- values up-to-date?

---

Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs. Section 5 discusses the key advantages and disadvantages of our approach with respect to existing approaches to distributed reactive programming. Finally, Sections 6 and 7 present related work and main conclusions, respectively.
Need **Context**

Fig. 1. Application that reacts to sensor values to either notify to close the window or to produce a feels-like value.

Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs.

Section 5 discusses the key advantages and disadvantages of our approach with respect to existing approaches to distributed reactive programming. Finally, Sections 6 and 7 present related work and main conclusions, respectively.

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:

- **Sources**:
  - $t_1$, $t_2$ represent temperature sensors,
  - $h_1$, $h_2$ represent humidity sensors,
  - wind represents a wind sensor,
  - $w_1$, $w_2$ produces weights used to average sensor values.

- **Components**:
  - $avg_t$ and $avg_h$ calculate the weighted averages of temperature and humidity, respectively.

- **Observers**:
  - closeWindow and feelsLike are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

- **Glitches**.
  - A glitch can occur, for example, if $w_1$, $w_2$ produces a value, triggering $avg_t$ and $avg_h$ to recalculate the averages, and later feelsLike updates its value after receiving a new value from $avg_t$ but before receiving from $avg_h$.

- **Timestamps**.
  - Alternatively, the feelsLike observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.
Avoiding glitches

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:

- **Data sources**: \( t_1, t_2 \) represent temperature sensors, \( h_1, h_2 \) represent humidity sensors, wind represents a wind sensor, and \( w_1, w_2 \) produces weights used to average sensor values.
- **Mix components**: \( \text{avg}_t \) and \( \text{avg}_h \) calculate the weighted averages of temperature and humidity, respectively.
- **Observers**: \( \text{closeWindow} \) and \( \text{feelsLike} \) are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

**Glitches.** A glitch can occur, for example, if \( w_1, w_2 \) produces a value, triggering \( \text{avg}_t \) and \( \text{avg}_h \) to recalculate the averages, and later \( \text{feelsLike} \) updates its value after receiving a new value from \( \text{avg}_t \) but before receiving from \( \text{avg}_h \).

**Timestamps.** Alternatively, the \( \text{feelsLike} \) observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.
Avoiding glitches

Fig. 1. Application that reacts to sensor values to either notify to close the window or to produce a feels-like value.

Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs. Section 5 discusses the key advantages and disadvantages of our approach with respect to existing approaches to distributed reactive programming. Finally, Sections 6 and 7 present related work and main conclusions, respectively.

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:

- **Data sources** (t₁, t₂, h₁, h₂, w₁, w₂, w₁₂),
- **Observers** (feelsLike),
- **Mixed components** (avgₜ, avgₜ, avgₜ, avgₜ)

The data sources t₁, t₂ represent temperature sensors, h₁, h₂ represent humidity sensors, wind represents a wind sensor, w₁, w₂ the open/closed status of a window, and w₁₂ produces weights used to average sensor values. The avgₜ and avgₜ components calculate the weighted averages of temperature and humidity, respectively. Finally, the observers closeWindow and feelsLike are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources. Glitches. A glitch can occur, for example, if w₁, w₂ produces a value, triggering avgₜ and avgₜ to recalculate the averages, and later feelsLike updates its value after receiving a new value from avgₜ but before receiving from avgₜ.

Timestamps. Alternatively, the feelsLike observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.
Generalising contexts

2 Motivation: composition of reactive IoT components

We use as a running example a simple distributed reactive application in the context of the Internet of Things (IoT), where different sensors produce values that are aggregated and displayed by different services. This example motivates our approach and helps explaining the design choices that influenced our framework.

The reactive application in Figure 1 is composed of:

- data sources ($t_1$), $t_2$,
- observers ($h_1$, $h_2$), $w_1$, $w_2$,
- and mixed components ($avg_t$, $avg_h$, feelsLike, closeWindow, feelsLike).

The data sources $t_1$, $t_2$ represent temperature sensors, $h_1$, $h_2$ represent humidity sensors, $w$ represents a wind sensor, and $w_1$, $w_2$ produce weights used to average sensor values. The $avg_t$ and $avg_h$ components calculate the weighted averages of temperature and humidity, respectively. Finally, the observers closeWindow and feelsLike are capable of producing side effects, namely to send a warning to close a window and to display a feels-like temperature value, respectively.

This IoT example illustrates some possible challenges that can occur when managing dataflows triggered by new values being produced by data sources.

- Glitches. A glitch can occur, for example, if $w_1$, $w_2$ produces a value, triggering $avg_t$ and $avg_h$ to recalculate the averages, and later feelsLike updates its value after receiving a new value from $avg_t$ but before receiving from $avg_h$.

- Timestamps. Alternatively, the feelsLike observer may choose to inspect the timestamps for when the original data sources produced the readings, and decide on whether these are within an acceptable time window.

\[
\text{ctx-FL} = \text{ctx-t} + \text{ctx-h}
\]

\[
\text{qual(\text{ctx-FL})} \geq \text{threshold}
\]

\[
\text{ctx-t}
\]

\[
\text{ctx-h}
\]

\[
\text{avg}_t
\]

\[
\text{avg}_h
\]

\[
\text{feelsLike}
\]

\[
\text{w_1,2}
\]

\[
\text{t_2}
\]

\[
25^\circ
\]

\[
38\%
\]
Generalising contexts

\[ \text{ctx-FL} = \text{ctx-1} + \text{ctx-2} \]

\[ \text{Qual}(\text{ctx}) \geq \text{threshold} \]
Beyond glitches

$(\text{ctx-1} + \text{ctx-2})$

$\text{qual}(\text{ctx}) \geq \text{threshold}$

- Geographic location
  - $\text{ok} = \text{close-by enough}$
- Relaxed glitch-freedom
  - $\text{ok} = \text{small counter difference}$
- Wall-clock Difference
  - $\text{ok} = \text{small time-stamp difference}$
Wrapping up

Distributed Reactive Programming: not optimal for the IoT

Add context to messages
combine and measure contexts
“discard” instead of “wait”

Thank you!

slides: http://quarp.proenca.org