Quality-Aware Reactive Programming for the Internet of Things

José Proença, Carlos Baquero

Quality-Aware Reactive Programming for the Internet of Things

José Proença* and Carlos Baquero**

HASLab, INESC TEC and Univ. of Minho, Portugal
{jose.proenca,cbm}@di.uminho.pt

Abstract. The reactive paradigm recently became very popular in user-interface development: updates – such as the ones from the mouse, keyboard, or from the network – can trigger a chain of computations organised in a dependency graph, letting the underlying engine control the scheduling of these computations. In the context of the Internet of Things (IoT), typical applications deploy components in distributed nodes and link their interfaces, employing a publish-subscribe architecture. The paradigm for *Distributed Reactive Programming* marries these two concepts, treating each distributed component as a reactive computation. However, existing approaches either require expensive synchronisation mechanisms or they do not support *pipelining*, i.e., allowing multiple "waves" of updates to be executed in parallel.
We propose *Quarp* (Quality-Aware Reactive Programming), a scalable and lightweight mechanism aimed at the IoT to orchestrate components triggered by updates of data-producing components or of aggregating components. This mechanism appends meta-information to messages between components capturing the context of the data, used to dynamically monitor and guarantee useful properties of the dynamic applications. These include the so-called glitch freedom, time synchronisation, and geographical proximity. We formalise Quarp using a simple operational semantics, provide concrete examples of useful instances of contexts, and situate our approach in the realm of distributed reactive programming.

Keywords: Reactive programming, component-based systems, pervasive systems, distributed systems, failure.

1 Introduction

Reactive programming is a paradigm that uses functions defined over streams of data, rather than the more traditional functions over values. Data sources are producers of data streams, and functions produce new streams based on their input streams. Producing a new value triggers a wave of functions that process the

* FCT grant SFRH/BPD/91908/2012 and H2020 project 732505 LightKone (2017-19).
** Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" is financed by the NORTE 2020 and through the European Regional Development Fund (ERDF).

new values. This paradigm became especially popular among developers of user-interfaces and reactive web pages [1,2,3,4], helping to manage the dependencies between updates (from the mouse, keyboard, network, etc.) and the display.

Recent attempts bring this paradigm to a distributed setting [5,6,7], carrying new challenges. Consider, for example, a twitter message (a *tweet*) being posted, consequently activating two independent services: one to make the tweet available, and one to notify all subscribers. Currently it is possible for a twitter client to be notified without the tweet being made available, leading to a *glitch* – a temporarily inconsistent state. In (non-distributed) reactive programming this is typically solved by scheduling the client execution after the executions of both the twitter data feeds and the notification engine. However, in a distributed setting some different, and leaner, coordination approach is required.

Distributed reactive programming [8] can attempt to fix this problem by adding extra constraints to ensure that all processing occurs on globally coordinated rounds. While this is simple and accurate, strong coordination does not scale well as more and more components need to agree on an order of execution, and faster components may have to wait for slower ones to catchup. Distributed systems are prone to regular failures on message transmission and transient partitions [9], calling for weaker coordination among components. In networks of low-resource devices such as the ones used by the LooCI middleware [10], common in the IoT, computation and communication is kept to a minimum to preserve energy, and it is often unrealistic to assume reliable communication.

This paper proposes Quarp – quality aware reactive programming – a more flexible approach to source coordination that rethinks on the amount of *out-of-synchrony* that qualifies as a genuine *glitch*, i.e. one that induces incorrect results. For instance, when combining slow varying data sources, such as environmental temperature, sensible outputs can still be derived when measurements are a few seconds apart. Reducing the synchronization requirements makes the overall system more resilient and fault tolerant. The key to this is to associate meta-data to data emitted by a source, and to assume a realistic network infrastructure where messages are eventually delivered, but can transiently be lost or received out of order. A tradeoff is to allow data loss, and still be able to progress when data goes through with sufficient synchronization quality. The alternative, of trying to act on all data, can easily stall all activity in complex deployments.

The key contributions of this paper are the formalisation of a core reactive language tailored for the IoT, that: (1) measures the quality of incoming messages; (2) can guarantee properties such as glitch-freedom; (3) supports more relaxed notions such as *"data sources are located nearby"* and *"glitch-freedom with an error margin"*; and (4) can be used in lightweight nodes since it does not rely on heavy computations or complex coordination protocols.

Organisation of the paper. Section 2 introduces the key challenges addressed by Quarp via a motivating example. Section 3 formalises the semantics of a simple pseudo-language for reactive programs: first without quality awareness, and later extended with quality attributes. Section 4 illustrates the generality of Quarp by exploring different notions of quality useful in reactive programs. Section 5

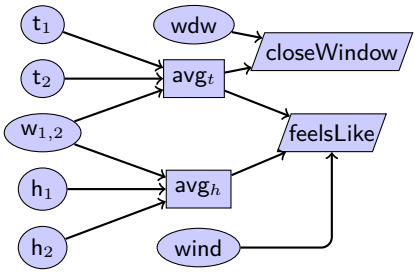

Fig. 1. Application that reacts to sensor values to either notify to close the window or to produce a feels-like value.

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 $\texttt{t}_1, \texttt{t}_2$ represent temperature sensors, $\texttt{h}_1, \texttt{h}_2$ represent humidity sensors, `wind` represents a wind sensor, `wdw` the open/closed status of a window, and $\texttt{w}_{1,2}$ produces weights that capture the relevance of each sensor for averaging. The $\texttt{avg}_t$ and $\texttt{avg}_h$ services calculate the weighted averages of temperatures and humidity values, 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 $\texttt{w}_{1,2}$ produces a value, triggering $\texttt{avg}_t$ and $\texttt{avg}_h$ to recalculate the averages, and later `feelsLike` updates its value after receiving a new value from $\texttt{avg}_t$ but *before* receiving from $\texttt{avg}_h$.

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

Geo-location The physical proximity of the sensors could also be considered when deciding on whether the input values of `feelsLike` should be taken together.

Context & Quality

The concept of *how good* are the input parameters of a given service call, with

respect to the original source of the data, is captured in Quarp by what we call the *context* of a message, and the *quality* of a context. Furthermore, we do not fix upfront what a context and a quality measurement are. Instead, we specify simple properties of contexts and operations, and properties of operations that must be defined over contexts and qualities. We also provide concrete examples of contexts and qualities that we found useful for reactive programs.

As a running example we will focus on glitch-freedom. We will use a simple context that labels every message with pairs of values, each containing a globally unique ID of a data source and the value of a local grow-only counter of the same data source. Every data source component starts by labelling its published values with a pair with its ID and counter value; every service that aggregates data, such as $\mathtt{avg}_t$, labels its published values with the joint labels of its input arguments. These labels are used to provide glitch-freedom guarantees, which is a typical concern in reactive programming. More complex labels, described later in this paper, can also include location information and wall-clock time sources.

More generally, contexts are expected to form a *commutative monoid*, i.e., to be able to be composed via an associative and commutative operator, and their associated qualities are expected to form a *bounded semi-lattice*, i.e., to have a partial order over possible qualities and to have a minimal quality.

Observe that, since every context is expected to have an associated quality, one could merge the concepts of context and quality, and require this merged qualified context to form both a commutative monoid and a bounded semi-lattice. We decided to keep this split for readability. Our notion of context and quality is inspired in constraint semirings [11], which possess two binary operators. One is similar to our composition of contexts, and the other is an idempotent operator that induces a partial order similar to our bounded semi-lattice.

3 Quarp: Quality-Aware Reactive Programming

We formalise the Quarp framework, generalising the notions of *context* and *quality*, and providing different examples of concrete instantiations for contexts and qualities. Components in a reactive system receive data by their *source ends* and publish data on their *sink ends*. A component is called a *data source* if it has no source ends, *observer* if it has no sink ends, and *mixed component* otherwise.

This section starts by formalising reactive programs, followed by an extension with quality-aware semantics and examples of useful quality metrics.

3.1 Basic reactive programs

A reactive program p is formally a set of component definitions, each written as $c \leftarrow \mathit{func}(\overline{\mathit{arg}})$, where *func* is a function with a list of arguments *arg*, and each argument $a \in \overline{\mathit{arg}}$ can be either a constant or a component. Our example in Figure 1 can be written as in Figure 2, where the interpretation of the functions is expected to be defined elsewhere. In practice, this abstraction of a reactive program could be derived from the source code of the individual components.

```
t1 ← getTemp("North")
t2 ← getTemp("South")
w12 ← getWeights()
h1 ← getHum("North")
h2 ← getHum("South")
wdw ← getWindowStatus()
```

```
wind ← getWind()
avgt ← calc-avg(t1,t2,w12)
avgh ← calc-avg(h1,h2,w12)
closeWindow ← notifyWindow(wdw,avgt)
feelsLike ← publishFL(avgt,avgh,wind)
```

Fig. 2. Encoding of the program in Figure 1.

Given a program p we say c_2 *subscribes to* c_1, written $c_1 \prec_p c_2$, if p contains $c_2 \leftarrow \mathit{func}(...,c_1,...)$. We omit p in $\prec$ when clear from context. In our example program we say that the `avgt` component subscribes to the components `t1`, `t2`, and `w12`, written `t1` $\prec$ `avgt`, `t2` $\prec$ `avgt`, and `w12` $\prec$ `avgt`. Hence $\prec$ defines a *dependency graph* between components, starting from the source components.

Informal runtime semantics Components communicate via a publish-subscribe mechanism, as in our IoT example. A program starts when a source component produces a value to be published. For example, when `t1` decides to publish the value 17. It places the value in an *output buffer* linked to `avgt`, representing the (non order-preserving) network communication. In turn, components like `avgt` have *input buffers*, one for each input, storing their last received value.

The program proceeds when the network atomically transfers one of its values to an input buffer. In our example, the networks transfers the output value 17 to the input buffer of `avgt`. Previously stored data in this buffer is overwritten, even if it was not processed yet by `avgt`, simulating failure in the communication.

Once the `avgt` service receives a new value, it checks if all its input values are ready, i.e., if all their associated input buffers are non-empty. This will only be the case after both `t1` and `w12` publish values that arrive to `avgt`. Upon receiving an update for one of its buffers, `avgt` calculates an average value based on its three parameters and places the result in its output buffer, which buffers data going to two components: `closeWindow` and `feelsLike`.

Note that, even though mixed components and observers can only process parameters when these are updated, these components can decide to ignore incoming messages, even if all the buffers are non-empty. This effectively mimics data being lost, since newer messages override previously received ones.

Formal runtime semantics Let $\mathcal{C}$ be the set of all components, $\mathcal{D}$ the domain of data produced by components, $\mathbb{P}\,X$ the set of all sets over X, and $\mathbb{M}\,X$ the set of all multisets over X. The runtime semantics of a reactive program is modelled by the evolution of so-called *input and output buffers*.

- Every source and mixed component $c \in \mathcal{C}$ has exactly one *output buffer*, written $\mathtt{out}_c : \mathcal{C} \to \mathbb{M}\,\mathcal{D}$, responsible for storing event data values published by c until they are consumed by its subscribers.
- Every observer and mixed component $c \in \mathcal{C}$ has exactly one *input buffer*, written $\mathtt{in}_c : \mathcal{C} \to (\mathcal{D} \cup \{-\})$, used to store the last value used by each input of c, whereas "$-$" represents the absence of a value used by a given input.

$$\frac{\mathtt{in}_c = \emptyset}{\langle I, O\rangle \xrightarrow{?c} \langle I, O\rangle} \quad \text{(src)} \qquad \frac{\mathtt{out}_c \in O \quad d \in \mathtt{out}_c(c')}{\langle I, O\rangle \xrightarrow{?c'} \langle I + (c, d, c'), O - (c, d, c')\rangle} \quad \text{(rcv)}$$

$$\frac{\langle I, O\rangle \xrightarrow{?c} \langle I', O'\rangle \quad \mathsf{active}_{I'}(c) \quad \mathsf{eval}_{I'}(c) = d}{\langle I, O\rangle \xrightarrow{!c,d} \langle I', O' + (c, d)\rangle} \quad \text{(pub)}$$

Fig. 3. Operational semantics of basic reactive components.

For example, $\mathtt{out}_{\mathtt{w12}} = \{\mathtt{avgt} \mapsto \{17, 19\}, \mathtt{avgh} \mapsto \{19\}\}$ means that the component `avgt` has pending values 17 and 19 from `w12`, and `avgh` has only a pending value 19 from `w12`. Regarding input buffers, $\mathtt{in}_{\mathtt{closeWindow}} = \{\mathtt{avgt} \mapsto 18, \mathtt{wdw} \mapsto \mathtt{open}\}$ means that the component `closeWindow` has previously used the values 18 and `open` as input from `avgt` and `wdw`, respectively, and $\mathtt{in}_{\mathtt{closeWindow}} = \{\mathtt{avgt} \mapsto 18, \mathtt{wdw} \mapsto -\}$ means that `closeWindow` never received a value from `wdw` before.

The use of multisets in input buffers instead of sequences captures the lack of order guarantees in the sending of messages. The state of a reactive program p is therefore captured by the set of all input and output buffers I_p and O_p in p, written $\langle I_p, O_p\rangle$. We write $\mathcal{I}$ and $\mathcal{O}$ to represent the set of all possible input and output buffers, respectively, and drop the program p in subscript when clear.

Finally the semantics of a reactive program is given by the rules in Figure 3, labelled by pairs $?c$ denoting that c is ready to be executed, and $!c, d$ denoting that c published the value d. The rule (src) represents a source component becoming ready to publish a value, the rule (rcv) represents a data value being delivered to a given component, and the rule (pub) represents a connector publishing a given data value. These rules use the auxiliary functions $\mathsf{active}_\mathsf{I}$, eval_I, $O + (c, d)$, $O - (c, d, c')$, and $I + (c, d, c')$, defined as follows.

- $\mathsf{active}_I : \mathbb{P}\,\mathcal{C}$**.** Predicate that says whether a given component c is *active* by checking if all its input buffers contain a value. Formally, $c \in \mathsf{active}_I$, also written as $\mathsf{active}_I(c)$, holds if for all $c' \prec c$, $\mathtt{in}_c(c') \neq -$, where $\mathtt{in}_c \in I$.
Example: $\mathsf{active}_I(\mathtt{avgt})$ *means that* `avgt` *is ready to be executed, i.e.,* $\mathtt{in}_{\mathtt{avgt}}(\mathtt{t1}) \neq -$, $\mathtt{in}_{\mathtt{avgt}}(\mathtt{t2}) \neq -$, *and* $\mathtt{in}_{\mathtt{avgt}}(\mathtt{w12}) \neq -$, *where* $\mathtt{in}_{\mathtt{avgt}} \in I$.

- $\mathsf{eval}_I : \mathcal{C} \to \mathcal{D}$**.** Function that, given the current buffers I and a connector c where $c \leftarrow \mathit{func}(\overline{\mathit{args}})$, (1) calculates $\overline{\mathit{args}}'$ by replacing in $\overline{\mathit{args}}$ all occurrences of input components c' by their last received value $\mathtt{in}_c(c')$ (where $\mathtt{in}_c \in I$), and returns the result of evaluating $\mathit{func}(\overline{\mathit{args}}')$.
Example: $\mathsf{eval}_I(\mathtt{avgt}) = 17.6$ *means that the result of evaluating* `calc-avg(t1,t2,w12)`*, after replacing* `t1`*,* `t2`*, and* `w12` *by the values in* I*, is 17.6.*

- $O + (c, d) : \mathcal{O}$ and $O - (c, d, c') : \mathcal{O}$**.** Functions that add and remove data to output buffers in O, respectively. Formally, when $\mathtt{out}_c \in O$ then: (1) $O + (c, d) = \{\mathtt{out}_c + d\} \cup (O \setminus \{\mathtt{out}_c\})$, where $\mathtt{out}_c + d = \{c_{out} \mapsto (M \cup d) \mid (c_{out} \mapsto M) \in \mathtt{out}_c\}$; and (2) $O - (c, d, c') = \{\mathtt{out}_c - (d, c')\} \cup (O \setminus \{\mathtt{out}_c\})$, where $(\mathtt{out}_c - (d, c'))(e) = \mathsf{if}\ (e = c')\ \mathsf{then}\ \mathtt{out}_c(e) - \{d\}\ \mathsf{else}\ \mathtt{out}_c(e)$.

• $I + (c, d, c') : \mathcal{I}$. Function that updates the buffers I by replacing the previous value of $\texttt{in}_{c'}(c)$ with d. Formally, when $\texttt{in}_{c'} \in I$ then $I + (c, d, c') = \{\texttt{in}_{c'} + (c, d)\} \cup (I \setminus \{\texttt{in}_{c'}\})$, where $(\texttt{in}_{c'} + (c, d))(e) = \mathsf{if}\ (e = c)\ \mathsf{then}\ d\ \mathsf{else}\ \texttt{in}_{c'}(e)$.

Example Consider our running example from Figure 1. Initially the input and output buffers I and O are empty, defined as follows:

$$\begin{aligned} I &= \{\texttt{in}_c \mid c \in \{\texttt{avgt,avgh,closeWindow,feelsLike}\}\} \\ \texttt{in}_c &= \{c' \mapsto - \mid c' \prec c\} \\ O &= \{\texttt{out}_c \mid c \in \{\texttt{t1,t2,w12,h1,h2,wdw,wind,avgt,avgh}\}\} \\ \texttt{out}_c &= \{c' \mapsto \emptyset \mid c \prec c'\} \end{aligned}$$

A possible trace that triggers the execution of `closeWindow` without data losses or re-orderings is, for some I^k and O^k with $k \in \{1, \ldots, 6\}$:

$$\begin{aligned} \langle I, O\rangle &\xrightarrow{!\texttt{t1},17} \langle I^1, O^1\rangle \xrightarrow{!\texttt{w12},\langle 0.6,0.4\rangle} \langle I^2, O^2\rangle \xrightarrow{!\texttt{wdw},\texttt{open}} \langle I^3, O^3\rangle \\ &\xrightarrow{!\texttt{t2},19} \langle I^4, O^4\rangle \xrightarrow{!\texttt{avgt},17.8} \langle I^5, O^5\rangle \xrightarrow{!\texttt{closeWindow},-} \langle I^6, O^6\rangle \end{aligned}$$

If a label starting with ? appears in a trace, it represents a trigger that was never used by a publish rule, i.e., data that was received but not used. After this trace, the input and output buffers in the state $\langle I^6, O^6\rangle$ should have updated into the following ones:

$$\begin{aligned} \texttt{in}_{\texttt{avgt}} &= \{\texttt{t1} \mapsto 17, \texttt{t2} \mapsto 19, \texttt{w12} \mapsto \langle 0.6, 0.4\rangle\} \\ \texttt{in}_{\texttt{closeWindow}} &= \{\texttt{wdw} \mapsto \texttt{open}, \texttt{avgt} \mapsto 17.8\} \\ \texttt{out}_{\texttt{w12}} &= \{\texttt{avgt} \mapsto \emptyset, \texttt{avgh} \mapsto \langle 0.6, 0.4\rangle\} \\ \texttt{out}_{\texttt{avgt}} &= \{\texttt{closeWindow} \mapsto \emptyset, \texttt{feelsLike} \mapsto 17.8\} \end{aligned}$$

In this final state `w12` and `avgt` published values that were not delivered yet, and `avgt` and `closeWindow` updated their input buffers with their last received values.

3.2 Adding Quality Awareness

We extend sources and mixed components to produce not only streams of data $d_1, \ldots, d_n$, but also to: (1) mark each produced data with a *context* Γ attribute, written $\Gamma \vdash d$, and (2) to compute a *quality* value $Q = (\!|\Gamma|\!)$ of contexts used to filter low-quality messages.

Context We write $\mathcal{G}$ to denote the set of all contexts, and $\mathcal{QD}$ to denote the set of data extended with a context. A data value $\Gamma \vdash d \in \mathcal{QD}$ represents a value $d \in \mathcal{D}$ that was calculated based on sources with context that were combined into $\Gamma \in \mathcal{G}$ via an associative and commutative operator $\otimes$. Hence, choosing a monoid $(\mathcal{G}, \otimes)$ defines what is a context and how are contexts composed.

Quality We write $\mathcal{Q}$ to denote the set of all quality values, and the function $(\!|\cdot|\!)$ assigns quality values to contexts. Quality values form a bounded join semi-lattice $(\mathcal{Q}, \oplus)$, where the partial order is defined in the usual way: $(Q_1 \leq Q_2) \Leftrightarrow$

$(Q_1 \oplus Q_2 = Q_2)$. We write $Q \in \mathcal{Q}$ to range over quality values, and $\emptyset_{\mathcal{Q}}$ to denote the minimal quality. Hence, choosing the semi-lattice $(\mathcal{Q}, \oplus)$ defines what is a quality value and their order, and $(\![\cdot]\!)$ defines how to qualify contexts.

Summarising, different reactive behaviours can be attained by using different definitions of the context monoid $(\mathcal{G}, \otimes)$, the semi-lattice $(\mathcal{Q}, \oplus)$, and the qualification function $(\![\cdot]\!)$.

Example: Glitch-freedom

We instantiate the structures $\mathcal{G}$ and $\mathcal{Q}$, and the operator $(\![\cdot]\!)$ as follows.

- $\mathcal{G} = \mathbb{P}(\mathcal{C} \times K)$ are sets of pairs that associate the (globally unique) ID of source components to the value of a local grow-only counter. Contexts are combined via set union, i.e., $\otimes = \cup$ with identity $\emptyset$.
- $\mathcal{Q} = \{\bot, \top\}$ are booleans indicating whether data is has glitches ($\bot$) or not ($\top$), and $\oplus = \vee$ and $\emptyset_{\mathcal{Q}} = \bot$. Observe that $\oplus$ induces the order $\bot \leq \top$.
- $(\![\Gamma]\!) = \forall (s_1, k_1), (s_2, k_2) \in \Gamma \cdot s_1 = s_2 \Rightarrow k_1 = k_2$ returns true if the context is glitch-free, i.e., if the same source is always mapped to the same identifier.

The order of the quality lattice is used by the runtime semantics (below), by allowing only values with a certain minimal quality Q^{min} to be published, and discarding the data value otherwise. In this glitch-freedom example, a sensible Q^{min} would be $\top$, meaning that only glitch-free values can be published.

Using the IoT running example with this glitch-freedom context, assume this program starts by `t1`, `t2`, and `w12` publishing the values 17, 19, and $\langle 0.6, 0.4 \rangle$, respectively. Using quality-awareness, each of these values are marked with a context value, e.g., $\{(\texttt{t1}, 0)\} \vdash 17$, $\{(\texttt{t2}, 0)\} \vdash 19$, and $\{(\texttt{w12}, 0)\} \vdash \langle 0.6, 0.4 \rangle$. The service `avgt`, upon receiving these three values, combines their contexts calculating $\{(\texttt{t1}, 0)\} \otimes \{(\texttt{t2}, 0)\} \otimes \{(\texttt{w12}, 0)\}$, obtaining $\Gamma = \{(\texttt{t1}, 0), (\texttt{t2}, 0), (\texttt{w12}, 0)\}$. It then calculates the quality of this context $(\![\Gamma]\!) = \top$, indicating that the combined context is glitch-free ($Q^{\mathsf{min}} \leq (\![\Gamma]\!)$). This gives green light to proceed, i.e., `avgt` will calculate `calc-avg(17,19,(0.6,0.4))` $= 17.8$ and publish $\Gamma \vdash 17.8$ to its buffer linked to `closeWindow` and `feelsLike`.

If, at some point in the execution, `feelsLike` receives $\Gamma \vdash 17.8$ from `avgt` and some value $\Gamma' \vdash v$ from `avgh`, it will combine $\Gamma \otimes \Gamma'$ and calculate its quality. This quality will yield $\top$ if and only if $\Gamma'(\texttt{w12}) = 0$, i.e., if the only shared data source of `avgt` and `avgh` (`w12`) has the same associated counter value (0). Otherwise $(\![\Gamma \otimes \Gamma']\!) = \bot$ and `feelsLike` does not publish a new value.

Formal runtime semantics This subsection extends the previous runtime semantics from Section 3, extending the domain from $\mathcal{D}$ to $\mathcal{QD}$. The minimum quality for publishing a value is a globally defined constant Q^{min}, such that $Q^{\mathsf{min}} \leq Q$ means that the quality Q is good enough for publishing.

In this extended semantics the output buffer of each component c is now over $\mathcal{QD}$, i.e., $c : \mathcal{C} \to \mathbb{P}\,\mathcal{QD}$. The functions active_I, eval_I, $O + x$, $O - x$, and $I + x$ are trivially adapted to data values in $\mathcal{QD}$ where necessary, and we replace the rule (pub) by two new rules that publish only when the minimal quality is met. For example, $d \in \mathtt{out}_c(c')$ is now written as $\Gamma \vdash d \in \mathtt{out}_c(c')$.

$$\frac{\langle I,O\rangle \xrightarrow{?c} \langle I',O'\rangle \quad \mathsf{active}_{I'}(c) \quad \mathsf{eval}_{I'}(c)=d \quad \mathsf{cxt}_{I'}(c)=\Gamma \quad Q^{\mathsf{min}} \leq (\![\Gamma]\!)}{\langle I,O\rangle \xrightarrow{!c,(\Gamma\vdash d)} \langle I',O' + (c,(\Gamma\vdash d))\rangle}\ (\text{pub}\checkmark) \qquad \frac{\langle I,O\rangle \xrightarrow{?c} \langle I',O'\rangle \quad \mathsf{active}_{I'}(c) \quad \mathsf{cxt}_{I'}(c)=\Gamma \quad Q^{\mathsf{min}} \not\leq (\![\Gamma]\!)}{\langle I,O\rangle \xrightarrow{!c,-} \langle I',O'\rangle}\ (\text{pub}\text{✗})$$

Fig. 4. Publishing rules for the quality-aware extension.

The new quality-aware semantics uses the same rules (src) and (rcv) as before (replacing d by $\Gamma \vdash d$), and the rule (pub) is replaced by the two new rules in Figure 4, which describe how (and when) components publish data values with context information. As before, the auxiliary functions used by these rules cxt_B and Q^{min} are presented below.

- $\mathsf{cxt}_I : \mathcal{C} \to \mathcal{G}$**.** Function that, given an active component c, collects all contexts from its inputs and returns their combination with $\otimes$. Formally, when $\mathtt{in}_c \in I$ then $\mathsf{cxt}_I(c) = \bigotimes_c \{\Gamma \mid c' \prec c, \exists d \in \mathcal{D} \cdot \mathtt{in}_c(c') = (\Gamma \vdash d)\}$, where $\bigotimes_c \emptyset = \Gamma_c$ (with Γ_c being the context of the source component c), and $\bigotimes_c \{\Gamma_1, \ldots, \Gamma_n\} = \Gamma_1 \otimes \ldots \otimes \Gamma_n$ (with $n > 0$).
Example: $\mathsf{cxt}_I(\mathtt{t1}) = \{(\mathtt{t1}, 1)\}$ *means that the current context of* `t1` *is* $\{(\mathtt{t1}, 1)\}$*, and* $\mathsf{cxt}_I(\mathtt{avgt})$ *returns the combined context* $\Gamma_1 \otimes \Gamma_2 \otimes \Gamma_3$*, where* $\mathtt{in}_{\mathtt{avgt}}(\mathtt{t1}) = \Gamma_1 \vdash d_1$*,* $\mathtt{in}_{\mathtt{avgt}}(\mathtt{t2}) = \Gamma_2 \vdash d_2$*, and* $\mathtt{in}_{\mathtt{avgt}}(\mathtt{w12}) = \Gamma_3 \vdash d_3$*, for some* $d_1, d_2, d_3 \in \mathcal{D}$*.*

- $Q^{\mathsf{min}} : \mathcal{Q}$**.** Globally defined minimum quality required to publish a value.
Example: Following our glitch-freedom example, let $Q \in \{\bot, \top\}$ *and* $\oplus = \vee$*, inducing* $\bot \leq \top$*. Hence,* $Q^{\mathsf{min}} = \top$ *means that, if* $Q^{\mathsf{min}} \leq x$*, then* x *must be* $\top$*.*

4 Beyond Glitch-Freedom: Modelling Different Contexts

Glitch-freedom is one possible distributed property that can be guaranteed dynamically using contexts in reactive programs. This mechanism to discard messages that violate a minimal quality standard can be applied to a variety of quality notions. This section presents three of these.

Geographical location The context of a value produced by a data source is now either (1) a pair of values with the geographical location where the data value was produced, or (2) the identity context if the notion of location does not apply. Combining contexts means collecting all possible locations, and they are ordered by size of the smallest bounding square, i.e., better quality means closer by locations. More precisely:

- $\mathcal{G} = \mathbb{P}(\mathcal{R} \times \mathcal{R})$ – a context is a set of coordinates that influenced the published value. Here $\otimes = \cup$ and $\emptyset$ is the identity.
- $\mathcal{Q} = \mathcal{R}_{\geq 0} \cup \{\infty\}$ – a quality value is a non-negative number measuring the size of the smallest bounding square that contains all coordinates, $\oplus = \mathsf{min}$, and $\emptyset_{\mathcal{Q}} = \infty$. Observe that smaller square means better quality, hence $\oplus$ induces a reversed order $\sqsubseteq$, i.e., $v_1 \sqsubseteq v_2$ iff $v_2 \leq v_1$.

- $(\!|\Gamma|\!) = (\mathsf{max}(\pi_1(\Gamma)) - \mathsf{min}(\pi_1(\Gamma)))^2 + (\mathsf{max}(\pi_2(\Gamma)) - \mathsf{min}(\pi_2(\Gamma)))^2$, where π_1 and π_2 return the first and second values of the pairs in a given list, respectively, returns the (square of) the diagonal of the smallest square that can contain all coordinates.
- $(\!|\emptyset|\!) = 0$, which captures the ideal quality.

Using these definitions of $\mathcal{G}$ and $\mathcal{Q}$ one needs only to specify a minimal quality Q^{min} defining the maximal accepted distance between input sources so a value can be published. Furthermore, data sources without an associated location (such as `w12`) can simply produce the empty context $\emptyset$.

In our example, assume we define $Q^{\mathsf{min}} = 10$ (for some distance unit) and `t1`, `t2`, `w12`, `h1`, `h2` publish the values, respectively, $\{(2,3)\} \vdash 17$, $\{(4,2)\} \vdash 19$, $\emptyset \vdash \langle 0.6, 0.4 \rangle$, $\{(16,18)\} \vdash 56$, and $\{(18,20)\} \vdash 58$. In this case, both services `avgt` and `avgh` are able to publish a value with an acceptable quality. For example, `avgt` will publish a value with context $\Gamma = \{(2,3),(4,2)\}$, which has the associated quality $(\!|\Gamma|\!) = 2^2 + 1^2 = 5$ (and $10 \sqsubseteq 5$, i.e., $5 \leq 10$). However, the service `feelsLike` is not able to publish a value with the data from these sensors: the combined context would be $\{(2,3),(4,2),(16,18),(18,20)\}$, which has a quality of $16^2 + 18^2 = 580$, wich is worse than the minimal quality 10.

Relaxed glitch-freedom This example relaxes the notion of glitch freedom, by introducing tolerance with respect to the counters used for glitch freedom. I.e., *small* glitches are ignored and allowed, whereas a small glitch is found whenever counters from the same source data are close enough. $\mathcal{G}$ and $\mathcal{Q}$ are defined as before, and a fix $\mathsf{tolerance}$ value is used to assign a quality to contexts.

- $\mathcal{G} = \mathbb{P}(\mathcal{C} \times K)$ are the same as before: pairs that associate the globally unique ID of source components to the value of a local grow-only counter, and $\otimes = \cup$. Unlike with strict glitch-freedom, the values in K must have a total order and there must be a distance $\mathsf{dist}(k_1, k_2)$ defined between counters.
- $\mathcal{Q} = \{\bot, \top\}$ are also the same: booleans indicating whether data is (relaxed) glitch-free ($\top$) or not ($\bot$).
- $(\!|\Gamma|\!) = \forall (s_1,k_1),(s_2,k_2) \in \Gamma \cdot s_1 = s_2 \Rightarrow \mathsf{dist}(k_1,k_2) \leq \mathsf{tolerance}$ – returns true if the distance between counters from the same data source do not differ more than the pre-defined value $\mathsf{tolerance}$.

In our example, start by defining K to be the natural numbers, $\mathsf{dist}(k_1,k_2) = \mathsf{abs}(k_1 - k_2)$, and $\mathsf{tolerance} = 1$. This choice means that counters for the same counter in different arguments can differ up to 1. For example, if `feelsLike` receives an argument from `avgt` whose context maps `w12` to a counter value ahead by 1 from the counter of the previously received argument from `avgh`, the service will still react to this input.

Wall-clock difference In some scenarios the hardware platform provides a highly accurate wall-clock among distributed data sources, guaranteeing that their internal clock is consistent up to a small error.[1] Here one may use a context

[1] This is true, for example, for modules using SmartMesh IP™ (`http://www.linear.com/products/smartmesh_ip`).

with a pair of bounds with the smallest and the largest timestamps, and require their difference to be smaller than a fixed threshold. More precisely:

- $\mathcal{G} = \mathbb{P}\, TS$ sets of relevant timestamps. Unlike in the other cases, there is no reference to the associated data source. As before, $\otimes = \cup$.
- $\mathcal{Q} = \mathcal{R}_{\geq 0} \cup \{\infty\}$ is a positive number denoting the largest time difference between timestamps. Similarly to geo-location, smaller values represent higher qualities: $\oplus = \mathsf{min}$ and $\emptyset_{\mathcal{Q}} = \infty$.
- $(\!|\Gamma|\!) = \mathsf{max}(\Gamma) - \mathsf{min}(\Gamma)$, where $\mathsf{max}(\emptyset) = \infty$ and $\mathsf{min}(\emptyset) = 0$, returns the largest difference between timestamps.

In our example, assume that our tolerance is 5 seconds, i.e., $Q^{\mathsf{min}} = 5s$, and that `t1`, `t2`, `w12`, `h1`, `h2` publish the values, respectively, $\{13{:}10{:}20\} \vdash 17$, $\{13{:}10{:}21\} \vdash 19$, $\emptyset \vdash \langle 0.6, 0.4\rangle$, $\{13{:}15{:}00\} \vdash 56$, and $\{13{:}15{:}03\} \vdash 58$. This means that temperatures and humidities are published around 5 minutes apart, the update time of the stamps is neglectable, and pairs of the same kind of sensors are less than 5s apart. Hence, both services `avgt` and `avgh` are able to publish a value with an acceptable quality, but the service `feelslike` will fail to publish a value because the combine context will be $\{13{:}10{:}20, 13{:}10{:}21, 13{:}15{:}00, 13{:}15{:}03\}$, which has an associated quality of more than 5 seconds.

Combining dimensions Given any two different choices for context $\mathcal{G}_1$, $\mathcal{G}_2$ and for quality $\mathcal{Q}_1$, $\mathcal{Q}_2$, these can be merged into a new context monoid $\mathcal{G}_{12}$ and quality metric $\mathcal{Q}_{12}$ as follows.

- $\mathcal{G}_{12} = \mathcal{G}_1 \times \mathcal{G}_2$ are pairs with an element from the first context and an element from the second one.
- $\mathcal{Q}_{12} = \mathcal{Q}_1 \times \mathcal{Q}_2$ are again pairs from both qualities, where $(q_1, q_2) \oplus_{12} (q_1', q_2') = (q_1 \oplus_1 q_1', q_2 \oplus_2 q_2')$ and $\emptyset_{\mathcal{Q}} = (\emptyset_{\mathcal{Q}_1}, \emptyset_{\mathcal{Q}_2})$. Observe that $(q_1, q_2) \leq (q_1', q_2')$ when $q_1 \leq q_1'$ and $q_2 \leq q_2'$.
- $(\!|(\Gamma_1, \Gamma_2)|\!)_{12} = ((\!|\Gamma_1|\!)_1, (\!|\Gamma_2|\!)_2)$ simply applies the encodings of each context.

One can easily prove that $\mathcal{G}_{12}$ is indeed a commutative monoid and that $\mathcal{Q}$ is a bounded semi-lattice. This allows the combination of any set of desired contexts; for example, one may want to have both glitch-freedom and geographical bounds.

5 Discussion

The Quarp approach for distributed reactive programming takes inspiration in algorithms for distributed systems that manage eventually consistent structures, such as CRDTs [12]. It does so by appending extra meta-information to messages that is used to help local nodes to react appropriately to inputs.

Unlike other approaches to distributed reactive programming (DRP) [6,5,13], we claim to be more *scalable*, more *dynamic*, and better suited for *non-reliable communication*. The cost for these desired properties is the possible loss of some values, as explained below. To support these claims we start by introducing some existing DRP approaches, and discuss each claim individually.

REScala [5,13] Drechsler et al. present an algorithm to implement distributed glitch-freedom in reactive programs, called SID-UP, and include a careful comparison with other approaches with respect to: (1) the number of steps, each consisting of a round of messages from a set of components to another set of components, and (2) the number of messages sent. Their algorithm makes the strong assumption that rounds are synchronised, i.e., the algorithm does not support *pipelining*: a round starts when a set of data sources publish some value, and it ends when no more messages are pending – a new round can only start after the previous round finished. The comparison approaches are Scala.React[14], Scala.Rx,[2] and a variation of ELM [2] that supports dynamic updates of the topology of the reactive program (but does not support pipelining). Their approach and evaluation focuses exclusively on the performance of a single round, while Quarp focuses on the performance of multiple (concurrent) rounds, where pipelining is a must. Dynamic updates to the topology are not problematic in Quarp because of the lack of a clear notion of round, and because the eventual loss of messages during reconfiguration is already tolerated by Quarp, effectively allowing for more unrestricted forms of reconfiguration than SID-UP.

DREAM [6] Is a Java distributed implementation with an acyclic overlay network of brokers that support publish-subscribe communication. The communication sub-system provides reliable message transmission by buffering and re-transmission of messages, and in this case the sub-system uses point-to-point TCP connections to provide basic FIFO properties. Several consistency guaranties are provided, ranging from causal consistency to a globally unique order of delivery by way of a central coordinator. Comparatively to Quarp, the DREAM approach is more rigid when it comes to dynamic reconfiguration. Reliable message delivery can require considerable buffering in the communication subsystem and can stale system availability when the network is dropping messages. In contrast Quarp has much weaker requirements on the communication middleware. It allows message loss and re-ordering while still enabling the system to progress when messages get received and the required quality criteria is met.

Scalability in Quarp Our proposed approach can scale up to a large number of components under the assumption that the size of the contexts does not grow too much. For example, our glitch-freedom implementation combines the local counters of all involved data sources, which behaves well with large chains of dependent components, but may require some attention when the number of dependent data sources is large. Observe that the generality of our approach allows customisation, e.g., defining the combination of contexts to create abstractions that hide information regarded as unnecessary. When compared with the above approaches, Quarp brings a large improvement with respect to the size of supported applications, since there is no need to either lock every round of data propagation (as in REScala), nor to require certain nodes to have full knowledge of the dependency graphs (as in DREAM). This advantage derives from the relaxation made that locally found inconsistencies (regarded as low quality inputs)

[2] https://github.com/lihaoyi/scala.rx

do not need to be fully solved, but can simply be blocked and ignored. I.e., when an issue such as a glitch is found, the input is ignored without guarantees that future messages will solve this glitch.

Dynamicity in Quarp Support for dynamic updates of the dependency among components was regarded as a key requirement from REScala. So much that the evaluation used a modified version of ELM's propagation algorithm that adds support for dynamic updates at the cost of losing support for pipelining, i.e., of allowing multiple rounds to be executing in parallel. In Quarp dynamic updates are trivially supported, again due to the fact that it accepts the possible loss of messages as part of the intended semantics.

Failure handling in Quarp Unlike other approaches for distributed reactive programming, Quarp uses the basic assumption that messages can be lost (and re-ordered). Lost messages are not resent – instead Quarp assumes newer messages will be more relevant, and does not try to recover from failures. This approach targets systems such as the Internet of Things, where the cost of maintaining a reliable communication is often too high or infeasible (due to mobility). Furthermore, orthogonal approaches to support reliable communication, such as TCP/IP, can be safely used with Quarp.

6 Related Work

Reactive programming is a form of event-driven programming that deals with propagating change through a program by representing events as time-varying values. Its most popular versions are not concurrent, focusing on local reactive programming on a single network node and dealing with functional transformations of time-varying values [8]. Several approaches exist on top of object-oriented languages [15,16], functional languages [2,16], and in the context of web-based applications [1,2,3]. Most approaches enforce glitch freedom, ensuring that a node in a dependency graph is updated only after all its antecedents are.

Distributed Reactive Programming (DRP) deals with time-varying inputs, distributed over multiple network nodes, and with the management of dependencies between concurrent components. In a distributed setting, the problem of glitch freedom is of crucial importance, since inconsistencies may endure due to network partitioning. Carreton et al. [17] integrate DRP with the actor model, but do not support glitch freedom. Drechsler et al. [5] propose an efficient algorithm that enables glitch free DRP for distributed programs with strong network guarantees, but not considering highly dynamic networks, network failures and partitioning. Margara and Salvaneschi [6] propose a Java-based framework that offers multiple layers of consistency each having their impact on performance. It supports glitch-freedom, but under a significant performance penalty.

Another body of related work on DRP are reactive frameworks or languages for web programming, such as Meteor,[3] Play,[4] Flapjax [1], Elm [2], and Re-

[3] www.meteor.com

[4] www.playframework.com

act.JS [3]. These are usually two-tier, client-server applications where change either originates from user interaction with the DOM (e.g., clicking buttons) or by server acknowledgements. The server and DOM elements are considered the time-varying values. Even though events may originate on a remote node (the server), the reactive program actually resides on the client and the distribution of logic is therefore much simpler than in truly distributed reactive programs.

Quarp proposes a new approach to distributed reactive programming that allows individual nodes to locally identify glitches. Glitches are not only identified but also measured, based on meta-information aggregated to events. By selecting relevant properties over measurements and over such meta-information, tradeoffs can be made between performance and quality of the produced values. This approach suits well cyber-physical systems because it avoids global synchronisers or schedulers, and supports aspects such as dynamic reconfiguration.

Observe that, in the context of the IoT, other formalisations have been proposed, many as calculus of concurrent nodes [18,19]. These focus on how to accurately describe existing IoT systems and on how to reason about notions such as behaviour equivalences. Quarp does not explore properties of the presented formal semantics; instead it experiments with a new approach to think and design distributed applications for networks of resource-constrained devices: by separating the concerns of reactive components with dependencies on other components, from when to decide when data is good enough to be used.

7 Conclusion and future work

This paper proposes Quarp – a quality aware approach for distributed reactive programming. This approach investigates how reactive languages could be used to program distributed applications for the Internet of Things (IoT), taking into account the presence of resource-constrained devices, high mobility, and unreliable communication. Furthermore, data from sensors have often some redundancy (older values are less important than new ones), making current reactive paradigm too synchronization heavy, possibly leading to never-ending waits for a message that has been lost. Our solution is to locally find unwanted inconsistencies, discarding data when they are found. Quarp is general enough to capture a range of possible inconsistencies, using attributes that must be "good enough" to be considered consistent. Hence Quarp, by not requiring messages to be always delivered, provides better performance (no need to agree with neighbours), scalability (large number of components can be executing in parallel), and availability (the system does not deadlock upon lost messages), while still guaranteeing that the messages are consistent, for some relaxed notion of consistency.

Our future work is two fold. On one hand we plan to apply Quarp to a concrete domain, exploring instances of quality attributes and performing a comprehensive evaluation. On the other hand we expect to use our formalisation to reason about reactive programs, e.g., defining notions of bisimulation to compare or minimize programs, to prove properties over reactive programs in Quarp.

References

1. L. A. Meyerovich, A. Guha, J. P. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for ajax applications," in *OOPSLA*. ACM, 2009, pp. 1–20.
2. E. Czaplicki, "Elm: Concurrent FRP for functional GUIs," Master's thesis, Harvard, 2012.
3. C. Gackenheimer, "What is react?" in *Introduction to React*. Springer, 2015, pp. 1–20.
4. B. Reynders, D. Devriese, and F. Piessens, "Multi-tier functional reactive programming for the web," in *Onward!* ACM, 2014, pp. 55–68.
5. J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini, "Distributed rescala: an update algorithm for distributed reactive programming," in *OOPSLA*. ACM, 2014, pp. 361–376.
6. A. Margara and G. Salvaneschi, "We have a DREAM: distributed reactive programming with consistency guarantees," in *DEBS*. ACM, 2014, pp. 142–153.
7. G. Salvaneschi, A. Margara, and G. Tamburrelli, "Reactive programming: A walkthrough," in *ICSE (2)*. IEEE Computer Society, 2015, pp. 953–954.
8. 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, Aug. 2013.
9. P. Bailis and K. Kingsbury, "The network is reliable," *Commun. ACM*, vol. 57, no. 9, pp. 48–55, Sep. 2014.
10. D. Hughes, K. Thoelen, J. Maerien, N. Matthys, J. Del Cid, W. Horre, C. Huygens, S. Michiels, and W. Joosen, "LooCI: The loosely-coupled component infrastructure," in *proceeding of NCA*, 2012, pp. 236–243.
11. S. Bistarelli, U. Montanari, and F. Rossi, "Semiring-based constraint satisfaction and optimization," *J. ACM*, vol. 44, no. 2, pp. 201–236, Mar. 1997.
12. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS*, ser. Lecture Notes in Computer Science, vol. 6976. Springer, 2011, pp. 386–400.
13. J. Drechsler and G. Salvaneschi, "Optimizing distributed REScala," in *Workshop on Reactive and Event-based Languages & Systems (REBLS)*, 2014.
14. I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.React," École Polytechnique Fédérale de Lausanne, Tech. Rep. EPFL-REPORT-176887, May 2012.
15. G. Salvaneschi, G. Hintz, and M. Mezini, "REScala: Bridging between object-oriented and functional style in reactive applications," in *Proceedings of the 13th international conference on Modularity*. ACM, 2014, pp. 25–36.
16. A. Courtney, "Frappé: Functional reactive programming in java," in *PADL*, ser. Lecture Notes in Computer Science, vol. 1990. Springer, 2001, pp. 29–44.
17. A. L. Carreton, S. Mostinckx, T. V. Cutsem, and W. D. Meuter, "Loosely-coupled distributed reactive programming in mobile ad hoc networks," in *TOOLS (48)*, ser. Lecture Notes in Computer Science, vol. 6141. Springer, 2010, pp. 41–60.
18. I. Lanese, L. Bedogni, and M. D. Felice, "Internet of things: a process calculus approach," in *SAC*. ACM, 2013, pp. 1339–1346.
19. R. Lanotte and M. Merro, "A semantic theory of the internet of things - (extended abstract)," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 9686. Springer, 2016, pp. 157–174.