# Refraction: Low-Cost Management of Reflective Meta-Data in Pervasive Component-Based Applications

Wilfried Daniels[1], José Proença[1,2], Dave Clarke[3], Wouter Joosen[1], Danny Hughes[1]

1. iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium.
{firstname.lastname}@cs.kuleuven.be

2. HASLab/INESC TEC, Universidade do Minho, Portugal

3. Computing Science Division, Uppsala University, Box 337, SE-751 05, Uppsala, Sweden.
dave.clarke@it.uu.se

## ABSTRACT

This paper proposes the concept of *refraction*, a principled means to lower the cost of managing reflective meta-data for pervasive systems. While prior work has demonstrated the benefits of reflective component-based middleware for building open and reconfigurable applications, the cost of using remote reflective operations remains high. Refractive components address this problem by selectively augmenting application data flows with their reflective meta-data, which travels at low cost to *refractive pools*, which serve as loci of inspection and control for the distributed application. Additionally *reactive policies* are introduced, providing a mechanism to trigger reconfigurations based on incoming reflective meta-data. We evaluate the performance of refraction in a case-study of automatic configuration repair for a real-world pervasive application. We show that refraction reduces network overhead in comparison to the direct use of reflective operations while not increasing development overhead. To enable further experimentation with the concept of refraction, we provide *RxCom*, an open-source refractive component model and supporting runtime environment.

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*; K.6.4 [**Management of Computing and Information Systems**]: System Management

## Keywords

Component-based systems, low-cost reflection, meta-data collection, pervasive systems, reactive reconfiguration

## 1. INTRODUCTION

Building applications for pervasive systems is notoriously difficult. In addition to the usual complexities of creating distributed applications, pervasive systems are *resource-constrained*, often deployed at large scale and in inaccessible locations, such as flood plains [1] or volcanoes [2]. This requires that all configuration and management must be performed remotely.

*Reflective component models* [3, 4, 5] have a strong track record of realising adaptable and evolvable applications for pervasive systems. The complexity of embedded software development is mitigated by the reuse of generic software components. Furthermore, customisable middleware [1] conserves system resources through the removal of redundant software functionality. Finally, runtime reconfiguration enables software evolution [6] and adaptation [7] after system deployment.

A reflective component model provides a per-component *meta-space* in which selected elements of the component implementation are reified through a *meta-model* [8]. This meta-model externalises elements of the component implementation, such as its incoming and outgoing connections and its properties. The meta-model is causally connected to the component implementation, which allows it to support both *introspection* (reading the meta-model) and *reconfiguration* (modifying the meta-model). The use of a per-component meta-model ensures that the scope of reconfiguration actions is bounded. However, when reconfiguring distributed applications it is frequently necessary to work in a coordinated fashion with the meta-model of multiple distributed components. This leads to increased development complexity and message passing overhead.

This paper proposes the concept of *refraction*, a principled means to lower the cost of reflection for pervasive applications. Refraction minimises the need to inspect and reconfigure individual components by incorporating an efficient meta-data distribution mechanism in the component model kernel. Refractive components use this mechanism to selectively augment the application data that they process with elements of their own meta-model. When refractive components are bound together they naturally form *refractive streams* along which component meta-data may flow. These streams terminate at *refractive pools*, a network location, where all components that contributed to the stream can be inspected and reconfigured. Reconfiguration is facilitated by the introduction of *reactive policies* which can be deployed on any node in the reactive stream or pool. These policies offer a mechanism to trigger reconfiguration based on incoming meta-data. We evaluate the benefits of refrac-

*CBSE'15,* May 4–8, 2015, Montréal, QC, Canada.
Copyright © 2015 ACM 978-1-4503-3471-6/15/05 ...$15.00.
http://dx.doi.org/10.1145/2737166.2737168.

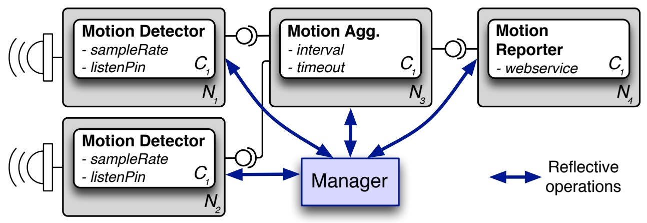

**Figure 1: Performing reflective operations on a distributed component composition.**

```
// introspection operations
N1.getProperties(C1)
N1.getProperty(C1,sampleRate)
N3.getWiresFrom(C1)
// reconfiguration operations
N3.setProperty(C1,timeout=30s)
N4.activateComponent(C1)
N3.wireTo(C1,N4,motionEvent)
N4.wireFrom(N3,C1,C1,motionEvent)
```

**Listing 1: Example of reflective operations.**

tion in a real-world case-study scenario from the domain of Wireless Sensor Networks (WSN): *configuration monitoring and repair.* Our evaluation shows that refraction significantly reduces message transmissions while not increasing development overhead.

The core contributions of this paper are two-fold. First, we the introduce the concept of refraction. Second, we apply refractive techniques to reduce the cost of automatic configuration repair. To support experimentation with the concept of refraction, we also provide an open-source refractive component model: *RxCom*, which is available online at `http://people.cs.kuleuven.be/~wilfried.daniels/refraction` together with all programming artefacts used in our evaluation.

The remainder of this paper is structured as follows. Section 2 provides background on reflection and its shortcomings. Section 3 outlines the principles of refraction. Section 4 applies these principles to realize a refractive component framework. Section 5 evaluates this framework in a real-world case-study scenario. Section 6 gives an overview of related work. Finally Section 7 concludes and discusses directions for future work.

## 2. REFLECTION AND ITS SHORTCOMINGS IN PERVASIVE SYSTEMS

Reflection is the capacity of a software system to inspect itself *(i.e. introspection)* and modify its structure, properties and behaviour at runtime *(i.e. reconfiguration)* [9]. The benefits of reflection for building open and reconfigurable distributed systems have been demonstrated in previous work [10, 7, 8, 9]. We build our ideas and a proof-of-concept implementation on top of LooCI, an existing component model for pervasive systems [3], although we believe that they are applicable to any reflective component model.

Figure 1 shows an example LooCI software composition that is used in a real-world WSN deployment for detecting motion in a room. In this example, two Motion Detector components —residing on $N_1$ and $N_2$—transmit their sensed data to a Motion Aggregator component on $N_3$, that aggregates motion readings and forwards processed data to a Motion Reporter component. Motion Reporter resides on a resource rich node $N_4$ and pushes the data to a web platform for viewing. Gray boxes represent computational platforms, where components are deployed and executed. Software components are shown as white boxes with solid black lines, which publish values via their *provided interfaces* (—○), and receive values from via their *required interfaces* (—( ). Components also have key-value pairs of properties that can be used to parameterise their behaviour.

The application is controlled by a Manager entity, which issues *reflective operations* to the component model kernel running on each node in order to *introspect* the software system, perform *structural reconfiguration* by connecting or disconnecting required and provided interfaces and *behavioural reconfiguration* by modifying component properties. The interaction between the Manager and reflective software system is depicted with thicker arrows in Figure 1. Examples of the reflective operations that may be used by the manager are shown in Listing 1—full details of the reflection API in LooCI can be found at `https://distrinet.cs.kuleuven.be/software/looci`.

Considering the example shown in Figure 1, two shortcomings of reflection quickly become evident. Firstly, reflection requires the transmission of many messages to query and reconfigure remote components. This is very problematic, as research [6] has shown that radio transmissions are the primary source of energy consumption for pervasive devices. Secondly, writing reflective code for distributed management, as shown in Listing 1 is complex. These problems could be mitigated by transmitting all meta-data to the managers, however, the memory and network resources consumed by storing reflective meta-data must also be minimised. This gives rise to three requirements that should be tackled by *refraction*:

- **Requirement 1:** The number of messages that are required to perform reflection should be minimised.
- **Requirement 2:** The subset of meta-information that is distributed should be customisable.
- **Requirement 3:** Mechanisms are required to specify in-network reconfiguration behaviour, that operates close to the point of action.

## 3. REFRACTION IN PRINCIPLE

As described in Section 2, reflective software processes are empowered to reflect upon and modify their implementation. Inspired by the power of this metaphor, we introduce the complementary metaphor of refraction. In the same way that characteristics of a material can be inferred from the light that traverses it, the characteristics of a refractive software component can be inferred from application data that it processes. Refracted meta-data travels together with application data across the distributed component graph, allowing the components that receive it to perform reflection at reduced cost.

Figure 2 shows how refraction distributes component meta-data across the network. Each node is extended with a *re-*

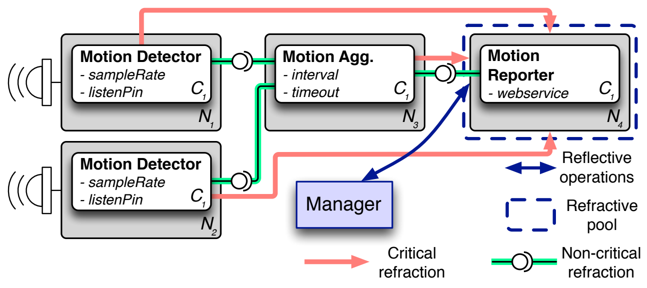

**Figure 2: Replacing reflective by refractive operations on a distributed system.**

*fraction engine*, which keeps relevant elements of the meta-model synchronised between nodes. A selected node, in our case $N_4$, collects all relevant meta-data and acts as a single-point of introspection for the 3 upstream nodes. This node is referred to as a *refractive pool*, depicted with a light blue contour line. Two types of *refractive streams* are offered: *non-critical* streams, shown in green and *critical* streams, shown in red. Non-critical refractive streams transport meta-data in an opportunistic way by augmenting application data, while critical refractive streams transport important meta-data directly to the refractive pool. Critical streams are offered as an alternative when the rate of application data is too low or unpredictable and a minimum latency should be guaranteed. We call this extended version of LooCI with refraction support *RxCom*.

RxCom provides mechanisms to selectively aggregate reflective meta-data updates that describe component properties and bindings and transmit them using either critical or non-critical refractive streams. *Refractive policies* provide a mechanism to specify what meta-data should be refracted using which type of refractive stream and where it should be stored. *Reactive policies* provide a mechanism to trigger reconfigurations based on incoming refracted meta-data. This new architecture minimises message passing through the use of aggregation, while not increasing development overhead. The following subsections address three key questions for RxCom:

*What does the basic meta-model describe?* The basic meta-model of a software component allows for the introspection and reconfiguration of software functionality, structure and behaviour. RxCom extends the basic LooCI component model, which is described in Section 3.1.

*How should the meta-model be refracted?* To minimise memory footprint, only a subset of the meta-model should be refracted. Yet it is not possible to know a-priori which meta-data should be distributed to support introspection and reconfiguration. We therefore advocate for customisable *refractive policies*, which determine which subset of the meta-data is distributed, and how it is distributed. This is described in Section 3.2.

*How to react to incoming refracted data?* Reconfiguration behaviour is frequently triggered by changing application context. To support this behaviour, nodes are equipped with *reactive policies* specifying how to store, forward and react to refracted meta-data. This is described in Section 3.3.

$$
\begin{array}{rcl}
\mathsf{Node} & ::= & \mathit{address} \times \mathsf{Comp}^* \times \utilde{\mathsf{Binding}}^* \\[1ex]
\mathsf{Comp} & ::= & \mathit{c\text{-}id} \times \mathit{c\text{-}type} \times \mathsf{Property}^* \times \\
 & & \mathsf{Interface}^* \times \utilde{\mathsf{Status}} \\[1ex]
\mathsf{Property} & ::= & \mathit{p\text{-}name} \times \utilde{\mathit{p\text{-}val}} \times \mathit{p\text{-}type} \\[1ex]
\mathsf{Interface} & ::= & \underline{\mathtt{required}} \times \mathit{e\text{-}type} \mid \underline{\mathtt{provided}} \times \mathit{e\text{-}type} \\
\mathsf{Status} & ::= & \underline{\mathtt{active}} \mid \underline{\mathtt{inactive}} \\[1ex]
\mathsf{Binding} & ::= & \mathsf{Local} \mid \mathsf{RemoteIn} \mid \mathsf{RemoteOut} \\
\mathsf{Local} & ::= & \mathit{e\text{-}type} \times \mathit{c\text{-}id} \times \mathit{c\text{-}id} \\
\mathsf{RemoteOut} & ::= & \mathit{e\text{-}type} \times \mathit{c\text{-}id} \times \mathit{address} \\
\mathsf{RemoteIn} & ::= & \mathit{e\text{-}type} \times \mathit{address} \times \mathit{c\text{-}id} \times \mathit{c\text{-}id}
\end{array}
$$

**Figure 3: RxCom's basic meta-model.**

## 3.1 The Refractive Meta-Model of RxCom

The basic LooCI API, which is shown in Figure 3 defines the meta-model of a LooCI node, component and binding. This meta-model determines what can be inspected and modified using reflection. RxCom then extends this meta-model with refractive concepts. While we use the LooCI meta-model as a running example, the refractive extensions are themselves generic and could be added to any component model.

In this paper, reflective operations are written using a simplified Java-like notation. For example, given a node $N$ we write $N$.**getProperties**($C$) to retrieve the set of properties of the component $C$ deployed on $N$, and $N$.**setProperty**($C$, `property=value`)) to modify the value of a property of a component $C$ deployed on node $N$. The API used in this paper is a simplified version of the full RxCom API, which is available online along with code samples and supporting middleware at: `http://people.cs.kuleuven.be/~wilfried.daniels/refraction`.

The following notation is used in the meta-model specifications: keywords written in a Sans font are terms defined by the grammar; *italic* keywords denote terms defined outside of the grammar, such as strings and numbers; underlined keywords denote constant symbols; the star operator $\cdot^*\cdot$ denotes a set of a given attribute and the times operator $\cdot \times \cdot$ creates tuples of elements. All elements of the meta-model can be introspected. Wave-underlined keywords denote parts of the meta-model that can also be reconfigured using reflective operations. The **bold blue** keywords in the code listings denote operations over the meta-model.

As shown in Figure 3, a node consists of an address, a set of *components*, and a set of *bindings*. Each component has a local instance identifier *c-id*, an identifying type *c-type*, a set of properties, a set of required (—( ) and provided (—○) interfaces, and a status indicating whether it is active or not. Each property associates a name *p-name* to a value *p-val* of a given type *p-type*. A binding connects one or more provided interfaces to required interfaces, and has a type *e-type* corresponding to the type of the events sent and received by both interfaces. This binding is said to be *local* when it connects two components running on the same node, and *remote* if it connects a components hosted on different nodes. Outgoing bindings specify the local component and the address of the remote node, while incoming bindings specify the originating node address, source component and the destination component.

$$\begin{aligned}
\mathsf{Node} &::= \mathit{address} \times \mathsf{Comp}^* \times \underset{\sim\sim\sim}{\mathsf{Binding}^*} \times \underset{\sim\sim\sim}{\mathsf{RefPolicy}^*} \\
\mathsf{RefPolicy} &::= \mathsf{SComp} \times \mathsf{SelectElem}^* \times \mathsf{Frequency} \times \mathsf{Target} \\
\mathsf{SComp} &::= \underline{\texttt{allComps}} \mid \textit{c-id} \mid \textit{c-type} \\
\mathsf{SelectElem} &::= \mathsf{SProps} \mid \mathsf{SBindings} \mid \mathsf{SStatus} \\
\mathsf{SProps} &::= \underline{\texttt{allProps}} \mid \textit{p-name} \mid \textit{p-type} \\
\mathsf{SStatus} &::= \underline{\texttt{status}} \\
\mathsf{SBindings} &::= \underline{\texttt{allBindings}} \mid \underline{\texttt{localBindings}} \\
&\quad \mid \underline{\texttt{inBindings}} \mid \underline{\texttt{outBindings}} \\
\mathsf{Frequency} &::= \underline{\texttt{on-change}} \mid \underline{\texttt{always}} \mid \underline{\texttt{never}} \\
\mathsf{Target} &::= \underline{\texttt{non-critical}} \mid \mathit{address}
\end{aligned}$$

**Figure 4: Extension of RxCom's meta-model with *refractive policies.***

```
// refract the properties of every component on N1
N1.refract(allComps,allProps,on-change,non-critical)
// continuously refract status of C1 on N3
N3.refract(C1,status,always,non-critical)
// critical refraction of C1 on N2 to node N4
N2.refract(C1,status,on-change,N4)
```

**Listing 2: Updating a refractive policy by modifying the meta-model.**

## 3.2 Specifying Refractive Policies

A refractive policy determines what subset of the reflective meta-model is refracted (i.e. transmitted with application data) and when. Refractive policies are specified by RefPolicy in Figure 4, which extends the basic meta-model of RxCom (Figure 3). A refractive policy consists of:

- SComp – specifies which component instance's meta-model to refract
- SelectElem – a query used to identify a collection of elements from the component's meta-model, such as the properties or bindings.
- Frequency – specifies when the selected meta-data elements should be refracted. The options are: `always`, which appends the selected refracted data to every outgoing message, `on-change`, which appends the selected refracted data only when it differs to the previously sent refracted data, and `never`, which causes the meta-data of the selected element not to be refracted.
- Target – specifies which mechanism should be used for transmitting the meta-data. The options are: `non-critical`, which augments outgoing events with meta-data, or *address*, which dispatches meta-data directly to the refractive pool located at the node with the given address.

Refractive policies are set on a per node basis and do not change the underlying component meta-model, rather they specify a systematic and customisable way to distribute that model. The API for refractive policies is exemplified in Listing 2, which shows how it is applied to the running example of Figure 2.

It should be noted that multiple policies may refer to the same element of the component meta-model with different frequency values. For example, there may be a general policy for `allComps` and a more specific one for a given component ID. In this case the most specific policy takes precedence.

$$\begin{aligned}
\mathsf{Node} &::= \mathit{address} \times \mathsf{Comp}^* \times \underset{\sim\sim\sim}{\mathsf{Binding}^*} \times \\
&\quad \underset{\sim\sim\sim}{\mathsf{RefPolicy}^*} \times \underset{\sim\sim\sim}{\mathsf{RctPolicy}^*} \\
\mathsf{RctPolicy} &::= \mathsf{SComp} \times \mathsf{Mark} \\
&\quad \mid \mathsf{Reconf} \\
\mathsf{Mark} &::= \underline{\texttt{forward}} \mid \underline{\texttt{store}} \mid \underline{\texttt{discard}} \\
\mathsf{Reconf} &::= \mathit{guard} \times \mathit{reconfiguration}
\end{aligned}$$

**Figure 5: Extension of RxCom's meta-model with *reactive policies.***

```
N4.store(C1)
N3.store(C1)
N3.forward(C1)

N4.addReconfiguration("sync.pol")

// contents of the "sync.pol" file
if ( N1.C1(sampleRate) != N1.C1(sampleRate)' ) {
  N2.C1(sampleRate) = N1.C1(sampleRate);
}
```

**Listing 3: Example specification of a reactive policy.**

## 3.3 Specifying reconfiguration policies

A node in RxCom can react in four different ways upon receiving refracted meta-data, it can: (i.) *discard* it; (ii.) *forward* it through a specific interface; (iii.) *store* it to the local meta-data registry; or (iv.) *trigger reconfigurations* based on the received meta-data update. The extension to the RxCom meta-model to accommodate these changes is presented in Figure 5, and exemplified in Listing 3.

Reactive policies can either be *component marks* or *conditional reconfigurations*. The former mark local components as being `forward` or `store`, and the latter associate a triggering condition to reconfiguration instructions. Upon receiving a remote message over the interface of a component $C$, RxCom searches for refracted data aggregated to it. If refracted data is found, its mark is checked. If $C$ is marked as `discard`, it the data is thrown away, otherwise, it is processed as follows:

1. If $C$ is marked as `forward` then the refracted data is queued to be aggregated to the next outgoing message from each provided interface of $C$ and of all local components that are connected to $C$.
2. If $C$ is marked as `store` then the refracted data is added to a *refraction table*, which is indexed by the component IDs and network addresses that provided the refracted data. If the data overrides a previous entry, then the old value is temporarily saved until the

next step completes. All nodes which store data become *refractive pools*.

3. Each reconfiguration $R$ with a guard $G$ that evaluates to true is executed. $G$ is a boolean expression over the local meta-model and the refraction table, both before and after the store operations. $R$ describes modifications to the meta-model of local or remote components using the standard reflection API.

The example in Listing 3 showcases how reactive policies apply to the composition of the running example depicted in Figure 2. The Motion Reporter component on $N_4$ and the Motion Aggregator on $N_3$ are set to store all refractive data, while Motion Aggregator is also set to forward. A reconfiguration is then added to $N_4$, which sets the rate parameter of the Motion Detector component on $N_1$ to the Motion Detector component on $N_2$ whenever it is updated. The reconfiguration and its associated guard are written in a domain specific language, which includes traditional arithmetic and logical operators. This language supports reading and the modification of meta-data, and uses the prime ($'$) as a suffix to represent the value prior to the store operations.

A refractive pool can be queried by a remote manager for the meta-model of both the node itself as well as for the refracted meta-data from all upstream nodes. Hence the API of refracted pools is extended to satisfy queries over a group of nodes, rather than over a single node. Examples of such queries are listed in Listing 4.

```
// get matching components from $N_1$
n1Compnts   = $N_1$.getComponents("Motion Detector")
// get matching components from all refracted nodes
moreCompnts = $N_4$.getRefComponents("Motion Detector")

// update component properties
$N_1$.setProperty(n1Compnts.head,sampleRate,120s)
forall (c ∈ moreCompnts) {
    c.getNode().setProperty(c,sampleRate,120s)
}

// add policy to all nodes known by $N_3$
forall (n ∈ $N_3$.getRefNodes()) {
  n.refract(allComps,allProps,on-change)
}
```

**Listing 4: Example queries on the refractive pool.**

## 4. REFRACTION IN PRACTICE

We realise the principles of refraction in RxCom by extending the Loosely-coupled Component Infrastructure (LooCI) [3] with support for refractive policies and reactive policies. Section 4.1 describes the implementation of LooCI, while Section 4.2 describes the implementation of refractive extensions.

### 4.1 The Standard LooCI Component Model

The Loosely-coupled Component Infrastructure (LooCI) [3] is a platform-independent component model and supporting middleware for networked embedded systems. The LooCI middleware is open-source and ports are available for embedded operating systems such as Contiki [11], Squawk [12] and Android. The remainder of this subsection provides a basic overview of the relevant features of LooCI. For brevity, we focus upon the Java version of LooCI on which we have built our prototype.

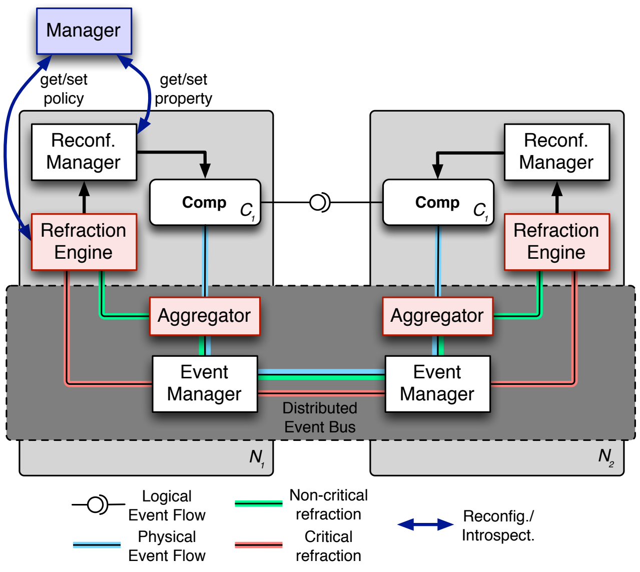

**Figure 6: The LooCI architecture (boxes with black lines) extended with RxCom modules (boxes with red lines).**

**Components** LooCI components are individually deployable units of functionality. They are managed by creating an instance of the basic LooCI meta-model described in Figure 3, using a simple component declaration and communication API consisting of required and provided interfaces. LooCI is language-agnostic and components may be implemented in C or Java, allowing developers to exploit language-specific features while providing standardised encapsulation, discovery and lifecycle management.

All LooCI components are connected to the LooCI runtime. Each component declares its human-readable name, its required interfaces (i.e. services) and provided interfaces (i.e. dependencies). All messages that travel across component interfaces are hierarchically typed as described in [13]. Components may also declare properties that allow for customisation of component behaviour.

**Architecture** The standard architecture of LooCI nodes is illustrated in Figure 6, excluding the components highlighted in red, which are the refraction extensions later explained in Section 4.2. In the figure we show several interactions: (i.) the logical event flow between components in green, (ii.) physical event flow between components in blue and (iii.) reconfiguration and introspection operations between an external Manager and a node. In the next sections the architectural components will be explained more in depth.

**Manager** Pervasive LooCI applications are deployed, inspected, and configured by Manager nodes. In principle any LooCI node may serve as a manager and a network may have multiple managers. The manager interacts with the nodes by using the reflection API to inspect and reconfigure LooCI's meta-model, as depicted in Figure 6.

**Reconfiguration Manager** The Reconfiguration Manager module exposes and maintains the component meta-model. As the meta-model is causally connected to the underlying software implementation, it may be manipulated in order to enact reconfiguration. Introspection allows for the discovery of component characteristics such as: type, status, properties, provided interfaces, required interfaces, current bindings and the associated code-base file. Reconfiguration manipulates the meta-model to control: the component life-cycle, configuration of properties and the binding of component interfaces.

**Distributed Event Bus** The distributed event bus is an asynchronous event-based communication medium that follows a decentralised topic-based publish-subscribe model. These topics are the event types mentioned in the previous section. Local and remote bindings are established by creating new subscription relationships, supporting one-to-one, many-to-one, and one-to-many bindings (as specified in Figure 3). The binding of components occurs at runtime and after component deployment. All bindings and event routing between nodes are handled by the Event Manager module, rather than by components, providing a strong separation between local component functionality and the management of distributed relationships. The difference between this physical event flow and the logical event flow is visualized in Figure 6.

### 4.2 The extended RxCom component model

RxCom extends LooCI's component model and supporting middleware with refractive mechanisms. Currently we have developed a version of RxCom for Java LooCI running on OSGI [14]. A version of RxCom is also under development for the C/Contiki [11] port of LooCI. This subsection emphasises the modifications that RxCom makes to the core LooCI model.

**Components** A component definition in RxCom preserves the same information as a component in the original LooCI meta-model. However, the refraction engine stores a mapping between refractive or reactive policies and their associated components. The former maps components to a list of pairs $(\{e\}, f, t)$ with the elements $\{e\}$ being refracted, $f$ the frequency of refraction and $t$ the target. The latter maps components to the flag `forward` or `store`. References to refracted meta-data and to the reconfigurations that are triggered by refracted data are maintained in separate tables. Reactive policies are evaluated and executed whenever matching meta-data is received via refraction.

**Architecture** The architecture illustrated in Figure 6 includes two refractions extensions: the Refraction Engine and Aggregator modules. These additional modules are isolated from existing functionality and use hooks available throughout LooCI to modify behaviour.

**Refraction Engine** The Refraction Engine module maintains the refractive policy table and the reactive reconfiguration policy table as described above, which specify how data should be refracted and used in reconfiguration respectively. The Refraction Engine module receives new policies and policy modifications from manager nodes; receives updates to the refractive meta-model from the Reconfiguration module; executes local reconfiguration instructions when triggered by refracted data and exchanges meta-data either through non-critical or critical refraction. Non-critical refractive meta-data is relayed to the Aggregator module, while critical refractive meta-data is sent directly as a single event over the distributed event bus.

**Aggregator** The Aggregator operates on the level of the distributed event bus and intercepts incoming and outgoing application events, and is responsible for aggregating and deaggregating the refracted data to and from the main application traffic. It uses a queue of outgoing refracted data for each output interfaces, to cache the data to be aggregated in the next outgoing message.

**Manager** The standard LooCI network manager is extended with refraction-related calls. This extended API allows, for example, to query the refraction tables (stored refracted data in the node), to update or add refraction and reactive policies, as illustrated in listings 3 and 4.

In addition to adding refraction-related calls, the manager was extended to parse and recognize the reactive policies shown in Listing 3. These policies are first parsed locally by the manager, and then serialized for transmission over the network. The target node will deserialize the policy and enforce it when relevant meta-data is received via a refractive stream.

## 5. EVALUATION

We begin our evaluation with an analysis of the performance overhead associated with evaluating policies and aggregating meta-data. We then present a configuration repair scenario that showcases the benefits of using refraction to inspect and reconfigure pervasive applications. This scenario is based on a real-world pervasive application. In the original approach, application configuration is periodically introspected and, when faults are discovered, repaired using remote reflective operations. In the case study we compare this reflection based approach to a refraction based approach. These results are presented in Section 5.2.

All the results presented in this section are benchmarked on an extended version of the Java/OSGi port of LooCI. All tests were conducted on a standard desktop machine, with an Intel Core i5-2400 CPU and 8GB of RAM.

### 5.1 Performance and overhead analysis

Refraction introduces some performance overhead when sending and receiving application data due to (i.) the aggregation of refractive meta-data with *outgoing* application traffic and (ii.) the deaggregation of refractive meta-data from *incoming* application traffic and the evaluation of reactive reconfiguration policies pertaining to the new meta-data. Table 1 shows minimum, maximum and average performance timings for all case study policies described in the following two sections. The mechanisms that underlie refraction perform well in terms of both the time required

**Table 1: Performance overhead of aggregation and policy evaluation.**

| | Min. | Max. | Avg. |
|---|---|---|---|
| Data Aggregation | 0.29 ms | 0.51 ms | 0.36 ms |
| Deaggregation + Policy Evaluation | 0.35 ms | 0.48 ms | 0.42 ms |

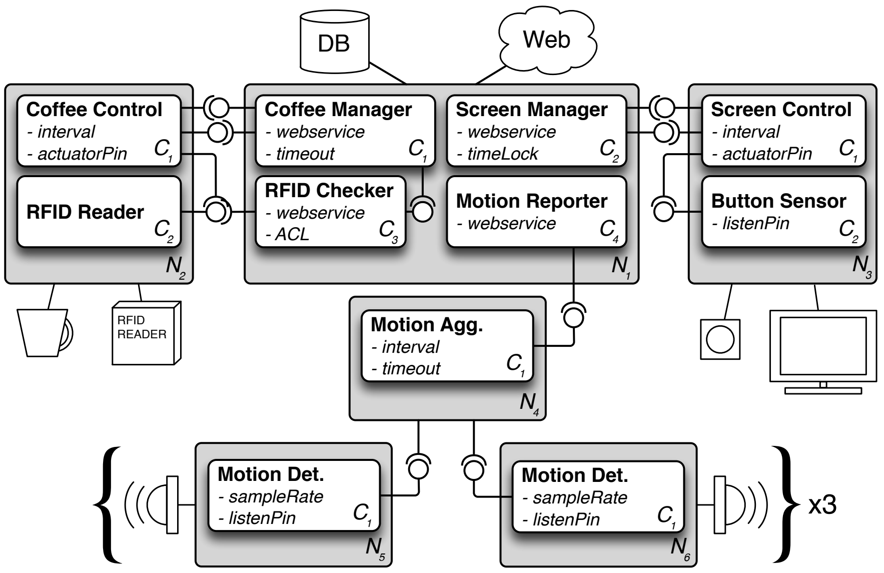

**Figure 7: Configuration repair scenario.**

to aggregate reflective meta-data and to deaggregate it and evaluate policies.

## 5.2 Configuration repair case-study

This case-study looks at a common problem with pervasive systems [15]: repairing faulty system configurations that arise due to faults, damage or power-loss. For example, configuration elements may be lost due to memory corruption after a node reboots. A classical solution to this is a monitoring service running on a reliable back-end. The monitor periodically queries the nodes for their configuration and checks if it matches with a *desired state*. If not, reconfiguration is carried out.

Refraction offers a more efficient alternative. First refractive policies are used to specify which configuration parameters from the meta-model should be refracted. Next, a refractive pool is created on the reliable back-end. Finally, reactive policies are installed on the back-end to check the current configuration against the desired configuration, and where necessary carry out repair operations.

**Scenario** Both approaches are benchmarked by implementing a common scenario, depicted in Figure 7, which is a subset of a real world ‘smart lab’ deployment in our research facility. More specifically, 3 services are offered by this composition: (i.) a *Motion detection* service using data from 6 embedded nodes equipped with motion sensors spread around the lab, (ii.) a *Screen control* service, which allows to either remotely or locally with a switch turn on or off screens used for presentations, and (iii.) a *Coffee control* service, which authorizes access to a coffee machine through RFID authorization.

$N_1$ is a reliable always-on back-end, and runs components with more complex functionality. Furthermore, this node is connected to a database for logging purposes, and all back-end components have configurable webservice interfaces to allow integration with a web platform. All other nodes in the network are embedded nodes with volatile configuration parameters. The result of this deployment is accessible online at `http://smartlab.looci.org/`.

While it is possible with both approaches to monitor a subset of the configuration parameters, in this scenario we consider a complete monitoring approach. We monitor the configuration of 15 components spread over 10 nodes. This amounts to 71 configuration parameters that have to be consistent for this deployment to successfully work.

**Development effort** A first point of comparison is the amount of development effort spent implementing a purely reflective solution versus using one that leverages refraction.

In case of a reflective monitoring solution, introspection commands are scripted in the backend to query all the parameters of the component composition spanning the network. When a specific parameter is deviating from the desired value, a reconfiguration command is sent to rectify the problem. An example of reflective code used for monitoring is shown in Listing 5. In this example, the parameters of one of the Motion Detector components are monitored.

Listing 6 on the other hand shows the reactive policies that can be used when using refraction for monitoring the same configuration parameters. It can be seen that the reactive policies closely match reflective operations from the point of view of the developer. The primary difference is that the execution of reflective code is statically scheduled, while reflective policies pertaining to a parameter are automatically evaluated when an updated parameter value is available.

We quantified general development effort in the form of LoC (Lines of Code) for both approaches. When implementing configuration repair for Figure 7, **142 lines** of reactive policies are required versus **146 lines** of reflective code. In conclusion, reflection imposes no development overhead when implementing a configuration repair system. On the contrary we expect that the absence of scheduling code will lead to development savings in more complex reconfiguration scenarios.

```
while(True) {
    // Component has to be active
    if(N5.getStatus(C1) == deactivated)
        N5.activateComponent(C1)

    // Guarantee a sampleRate >= 60
    if(N5.getProperty(C1, sampleRate) < 60)
        N5.setProperty(C1, sampleRate, 60)

    // Motion detector is connected to pin 2 on C1
    if(N5.getProperty(C1, listenPin) != 2)
        N5.setProperty(C1, listenPin, 2)

    // Wire motion events to aggregator on N4
    if(N5.hasWireTo(C1, N4, motionEvent))
        N5.wireTo(C1, N4, motionEvent)

    ...

    // Monitoring rate: hourly
    sleep(1h);
}
```

**Listing 5: Example of reflective code for monitoring the Motion Detector's parameters on $N_5$.**

```
if(N5.C1.Status == deactivated)
    N5.Status = activated

if(N5.C1(sampleRate) < 60)
    N5.C1(sampleRate) = 60

if(N5.C1(listenPin) != 2)
    N5.C1(listenPin) = 2

if(! hasWireTo(N5.C1,N4,motionEvent)
    wireTo(N5.C1,N4,motionEvent)

...
```

**Listing 6: Example of refractive code for monitoring.**

**Average Latency** A second point of comparison is the average latency of configuration repair. This metric indicates how long it takes before a fault is repaired. Figure 8 breaks this down for each component in the compositions and every monitoring method.

As can be seen from Figure 8, repair latency based on non-critical refraction varies for each component. This is caused by delays imposed by application data. RFID Reader and Button Sensor are very ill-suited for non-critical refraction, because their application data is sent unpredictably. Only when somebody pushes the button or swipes an RFID card is application data generated. In the data shown in Figure 8, it is assumed that on average every 30 minutes an event is sent. However, in reality application events are unpredictable and there are no hard guarantees in terms of average repair latency.

Both critical refraction and classic monitoring at two different rates give the same average repair latency for every single parameter in the network. When comparing all

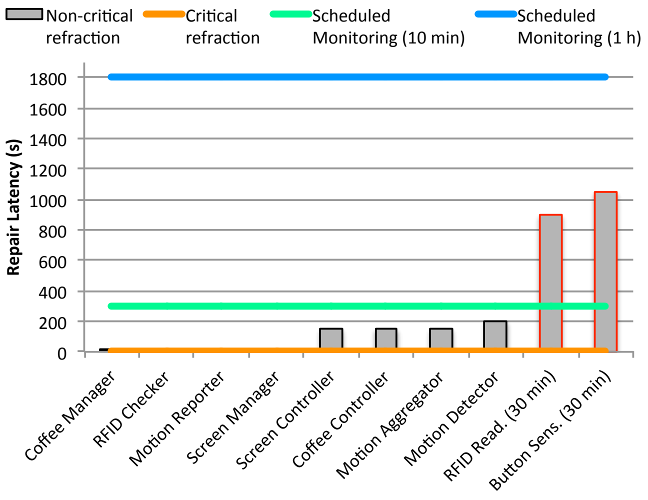

**Figure 8: Average latency of configuration repair.**

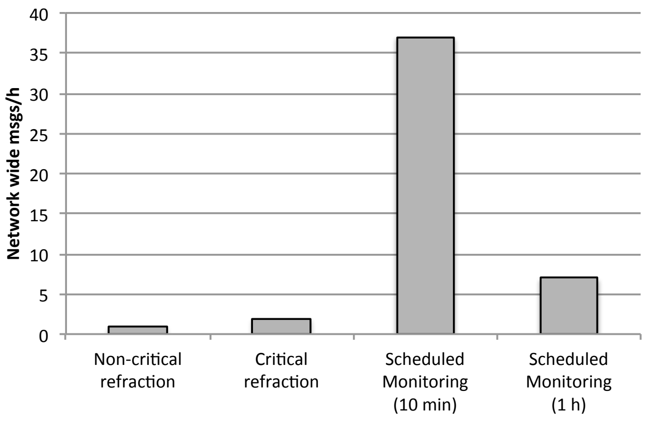

**Figure 9: Explicit messages sent per hour network wide for 1 configuration repair.**

approaches, we can conclude that both critical and non-critical refraction give significant latency advantages when compared to classic monitoring with reflective operations. When dealing with stochastic application data, critical refraction performs best.

**Network overhead** Lastly, we compare the amount of explicit network messages per hour required for repairing one *remote* configuration fault in the same timespan. Figure 8 shows the results.

We can determine that a classic monitoring approach almost always causes more explicit network traffic than either critical or non-critical refraction when using reasonable checking intervals. The message overhead for monitoring shown in Figure 9 is still fairly conservative, because the assumption is made that all configuration parameters of a single node can be packed in one explicit messages. Furthermore, explicit messaging overhead of monitoring will increase linearly with the scale of the network, while critical and non-critcal refraction will scale gracefully.

## 6. RELATED WORK

There are two key streams of related work: (i.) reflective component models and (ii.) reconfiguration frameworks. These are reviewed in Sections 6.1 and 6.2 respectively.

### 6.1 Reflective Component Models

The application of reflective programming techniques to distributed middleware was pioneered by Blair et al. [4], who advocated for a systematic approach to reflection based upon a per-component *meta-space* wherein select elements of software implementation are systematically externalised and reified through a *meta-model*. This stream of research has culminated in a number of reflective component models for pervasive systems.

OpenCOM [4] is a generic reflective component model that has been applied to build pervasive sensing applications [7]. OpenCOM is platform and language independent and supports runtime reconfiguration and introspection of the meta-model. The local OpenCOM component model may be extended with binding model *plug-ins* to support distributed application composition. RUNES [16] is a specialized branch OpenCOM that provide dedicated support for reflection in resource constrained systems.

OSGi [14] is a reflective Java-based component model that targets gateway-class pervasive devices. In addition to support for introspection and reconfiguration, OSGi also provides a secure execution environment. R-OSGi [17] extends OSGi with support for creating distributed bindings.

REMORA [5] provides a C-like programming language to specify component interfaces and application compositions. At compile time, the REMORA component implementation language is compiled to byte-code that is executed on the REMORA platform abstraction layer. REMORA supports introspection and reconfiguration.

LooCI [3], as described in detail in Sections 3.1 and 4.1, is a language and platform independent component model designed to support pervasive applications. LooCI supports both introspection and reconfiguration at runtime. In contrast to the models discussed above, which extension to support distribution, the LooCI runtime inherently supports the creation of distributed component bindings.

All of the prior models discussed above use a local meta-space and offer no inherent support for distributing component meta-data in order to support distributed reconfiguration. This results in high development complexity and message passing overhead when introspecting and reconfiguring components. While there are various small differences in the component meta-models described above, refraction offers a systematic approach to distributing meta-data that is independent of a specific component meta-model or binding approach.

### 6.2 Reconfiguration Frameworks

In a drive to reduce the complexity of distributed reflective programming and increase the reusability of software components, Parlavantzas et al. [18] introduce the concept of a *component framework* (CF). A CF first enforces a common composition structure and second allows its constituent components to be managed as a single entity. For example, Open Overlays [7] provides a generic *overlay network* CF that enforces a control-state-forward seperation of concerns and allows reflective operations to be applied to all constituent components via the CF.

Distributed component frameworks [18] address the problem of reducing development effort for reflective distributed systems, by aggregating distributed components into a single software entity that can be more easily introspected and reconfigured. However, there are a number of key differences with *refraction*. First, Distributed CFs make direct use of reflective primitives and therefore do not reduce the number of messages transmitted, whereas refraction reduces network load by aggregating meta-data with application messages. Second, CFs must be declared before system deployment and configuration, whereas refractive policies may be modified at runtime. In our view, refraction is a natural complement to CFs and could be used to create an efficient flow of meta-data from constituent components to their host component framework.

## 7. CONCLUSION AND FUTURE WORK

This paper proposed *refraction*, a principled means to lower the cost of reflection in pervasive systems, and introduced RxCom, a component model that realises this concept. RxCom provides a concise and low-overhead mechanism to distributed reflective meta-data accross distributed pervasive applications. RxCom provides the developer with: (i.) *refractive policies* to manage meta-data distribution, (ii.) *reactive policies* to automatically react to updates of meta-data from remote neighbours and (iii.) efficient runtime support for meta-data distribution and reconfiguration enactment.

Evaluation of RxCom shows that policy evaluation has low overhead. Furthermore, we reimplemented a configuration repair case-study from prior work [15]. In comparison to monitoring purely through reflection, the refractive solution developed using RxCom requires only a small fraction of the development effort and significantly reduce network overhead.

Our future work will proceed along three fronts. First we will port RxCom to the embedded C/Contiki version of LooCI in order to demonstrate its applicability resource constrained pervasive scenarios. Second, we plan to develop and deploy a large-scale case study application using RxCom. Finally, we will investigate the support for *managing* refractive primitives. While the evaluation results presented in this paper are very positive, we believe that more ature support for developing, deploying and (re-)configuring refractive functionality will result in further effort savings.

## Acknowledgements

This research is partially funded by the Research Fund KU Leuven, FCT grant BPD/91908/2012 in Portugal, and by the Agency for Innovation by Science and Technology in Flanders (IWT).

## 8. REFERENCES

[1] D. Hughes, P. Greenwood, G. S. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. J. Beven, "An experiment with reflective middleware to support grid-based flood monitoring," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 11, pp. 1303–1316, 2008.

[2] K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, "Deploying a wireless sensor

network on an active volcano," in *IEEE Internet Computing*, 2006, pp. 18–25.

[3] 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 *In proceeding of 11th IEEE International Symposium on Network Computing and Applications*, 2012, pp. 236–243.

[4] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1:1–1:42, Mar. 2008.

[5] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "Programming sensor networks using remora component model," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, Eds. Springer Berlin Heidelberg, 2010, vol. 6131, pp. 45–62.

[6] D. Hughes, E. Canete, W. Daniels, R. G. Sankar, J. Meneghello, N. Matthys, J. Maerien, S. Michiels, C. Huygens, W. Joosen, M. Wijnants, W. Lamotte, E. Hulsmans, B. Lannoo, and I. Moerman, "Energy aware software evolution for wireless sensor networks," in *WOWMOM*. IEEE, 2013, pp. 1–9.

[7] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taïani, "Experiences with open overlays: a middleware approach to network heterogeneity," in *EuroSys*, J. S. Sventek and S. Hand, Eds. ACM, 2008, pp. 123–136.

[8] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware '98. London, UK, UK: Springer-Verlag, 1998, pp. 191–206.

[9] B. C. Smith, "Procedural reflection in programming languages," Ph.D. dissertation, MIT, 1982.

[10] D. Hughes, P. Greenwood, G. S. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. J. Beven, "An experiment with reflective middleware to support grid-based flood monitoring," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 11, pp. 1303–1316, 2008.

[11] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *In proceedings of 29th Annual IEEE International Conference on Local Computer Networks*, Nov 2004, pp. 455–462.

[12] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices," in *Proceedings of the 2nd international conference on Virtual execution environments (VEE '06)*. New York, New York, USA: ACM Press, 2006, pp. 78–88.

[13] K. Thoelen, D. Preuveneers, S. Michiels, W. Joosen, and D. Hughes, "Types in their prime: sub-typing of data in resource constrained environments," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2013)*, 2013.

[14] A. L. Tavares and M. T. Valente, "A gentle introduction to OSGi," *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 5, pp. 8:1–8:5, Aug. 2008.

[15] J. Maerien, C. Huygens, D. Hughes, and W. Joosen, "Enabling resource sharing in heterogeneous wireless sensor network," in *to appear in proc. of the Middleware for IoT workshop (MW4IoT'15)*, 2014.

[16] P. Costa, G. Coulson, C. Mascolo, G. Picco, and S. Zachariadis, "The runes middleware: a reconfigurable component-based approach to networked embedded systems," in *In proceedings of IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*, 2005, pp. 806–810 Vol. 2.

[17] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed applications through software modularization," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 1–20.

[18] N. Parlavantzas and G. Coulson, "Designing and constructing modifiable middleware using component frameworks," *IET Software*, vol. 1, no. 4, pp. 113–126, 2007.