Tomography: lowering management overhead for distributed component-based applications

Wilfried Daniels, José Proença, Nelson Matthys, Wouter Joosen, Danny Hughes

iMinds-DistriNet, KU Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT

This paper introduces the concept of *tomography*, a mechanism to lower management overhead for component-based IoT applications. Previous research has shown the advantages of component-based software engineering, wherein applications are built and reconfigured at runtime through the composition of components. While this approach promotes code-reuse and dynamic reconfiguration, the introspection and reconfiguration of distributed applications is cumbersome and inefficient. *Tomography* addresses this problem by reimagining the visitor design pattern for distributed component based compositions. We evaluate the performance of this approach in a case-study of discovering/introspecting and reconfiguring a real-world IoT application. We show that in comparison to classic management operations, *tomography* reduces both the number of explicit queries and the volume of network messages. This significantly reduces management effort and energy consumption.

1. INTRODUCTION

IoT applications are known to be hard to build and maintain. Typical IoT applications are run on a large scale infrastructure of extremely *resource constrained* embedded systems, which are often deployed in hard to reach locations like flood plains [1] or volcanoes [2]. Because of this, IoT systems call for remote management and reconfiguration. A promising solution for this is reflective component-based middleware [3, 4, 5]. Components are small units of functionality with clearly defined interfaces and parameters, which can be deployed and managed remotely. Applications are built by binding components together in a composition. These mechanisms allow for software evolution [6] and adaptation [7] after deployment, while at the same time promoting software reuse through the combination of generic components.

Reflective component models introduce a per component meta-model, which is causally connected to the implementation of the component. The meta-model exposes elements of the component—such as interfaces and parameters that influence component behaviour—that can either be *introspected* (querying of meta-model) or *reconfigured* (modifying the meta-model). The introspection and reconfiguration of the meta-model is an important tool to monitor and enact change in a component composition. However, the per-component nature of the meta-model implies a large message passing overhead and high management complexity when introspecting or reconfiguring a distributed component composition, which spans many nodes as in the IoT.

This paper proposes the concept of *tomography*, a new approach to efficiently introspect and reconfigure component-compositions and thus distributed applications. Tomography is inspired by the visitor design pattern from object-oriented programming. The visitor pattern is a structured way of performing an operation on an object hierarchy by having a visiting object traverse the hierarchy [8]. Tomography applies this concept to distributed component compositions, exploiting existing communication flows for efficiency.

Tomography is based on the notion of *probes* that can be injected into a component composition, flowing along the same path as the application data. While traversing the component composition, the probe can either query or modify the meta-model of individual components as it traverses them. We evaluate the benefits of tomography in a real-world IoT case-study: a *Smart Lab* composed of *12 sensor nodes*. Our evaluation shows that tomography greatly reduces the number of commands that developers must issue and also the number of messages that are transmitted.

The main contributions are twofold. Firstly, we introduce a notion of region of a distributed component-based application based on selected entry points for visiting the region. Secondly, we introduce an approach to inspect and reconfigure distributed regions while traversing them, dubbed tomography. This new approach can reduce the number of remote messages for inspection and reconfiguration, and can alleviate the effort of managing regions with dynamic topologies and potentially across third party networks.

The remainder of this paper is structured as follows. Section 2 provides background of traditional reflective operations in component-based middleware and its shortcomings. Section 3 outlines the principles of tomography. Section 4 evaluates this framework in a real-world case-study scenario. Section 5 discusses related work. Section 6 concludes and discusses directions for future work.

M4IoT 2015 December 07–11, 2015, Vancouver, BC, Canada.
©2015 ACM. ISBN 978-1-4503-3731-1/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2836127.2836128

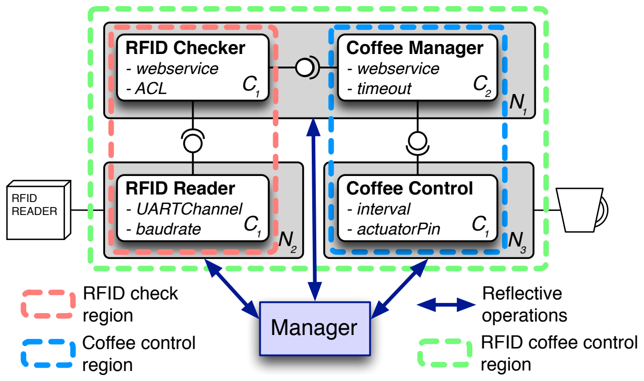

Figure 1: Example of a composition of components.

2. REFLECTIVE MIDDLEWARE

Reflection is a way for software systems to *introspect* and *reconfigure* themselves at runtime [9]. In the context of IoT, previous work [10, 7, 11] has shown that reflective component-based middleware is a viable way to ensure flexibility and adaptability of running deployments, while at the same time providing a way to monitor application configurations. In this paper we build our ideas on top of LooCI [3], a reflective middleware for the IoT, although this approach can be applied to other reflective component-based systems.

Figure 1 shows a typical component composition, used in a real-world IoT smart lab deployment, accessible online at `http://smartlab.looci.org`. An extended version of this application is used later in our evaluation. This example application authorises users to operate a coffee machine through an RFID tag. White boxes denote software components, while grey boxes denote physical nodes. Components publish values via their *provided interfaces* (—○), and receive values via their *required interfaces* (—( ). Components also have key-value pairs of properties that can be used to parameterise their behaviour.

In the distributed component composition shown in Figure 1, node N_2 is equipped with RFID hardware, and is running an RFID Reader software component. This component transmits swiped IDs to a remote RFID Checker component residing on resource rich back-end node N_1, which authorises or denies access based on an access control list (ACL). Additionally, access attempts are logged and viewable on a web platform. The coffee machine is physically connected to node N_3, where the Coffee Control component controls its state. The Coffee Control component is connected to the back-end Coffee Manager component, which exposes control over it on a webpage. Lastly, RFID Checker and Coffee Manager are bound in the back-end, so an authorised RFID swipe enables the coffee machine. Deployed components have a node-local identifier, denoted in Figure 1 with C_i.

Each component has a local meta-model that is causally connected to the underlying implementation, consisting of immutable (i.e. component type, provided and required interfaces) and mutable (i.e. bindings to and from other components, parameters, status) meta-data. Reflection allows the per-component meta-model of components to be remotely *instrospected* and *reconfigured* by a Manager entity. These interactions are visualised in Figure 1 by the thicker bidirectional arrows. Reconfiguration can either be *structural*, by connecting and disconnecting bindings, or *behavioural* by changing component parameters. Listing 1 exemplifies the usage of reflective operations.

```
// introspection operations
N1.getProperty(C1,ACL)
N3.getWiresFrom(C1)
// reconfiguration operations
N3.setProperty(C1,interval=30s)
N2.wireTo(C1,N1,rfidEvent)
N1.wireFrom(N2,C1,C1,rfidEvent)

// introspection: properties of RFID coffee control app
N2.getProperties(C1)
N1.getProperties(C1)
N1.getProperties(C2)
N3.getProperties(C1)
// reconfiguration: deactivate RFID check region
N2.deactivateComponent(C1)
N1.deactivateComponent(C1)
```

Listing 1: Example of reflective operations.

Based on our experience, the same reflective operation often has to be performed over a group of connected components in an application. In our IoT application in Figure 1, for example, we distinguish 3 *regions* that group distributed components that are often queried together. The last 2 blocks of operations in Listing 1 exemplify operations made to groups of components in our smart lab deployment: to collect properties of all components of a region, and to deactivate components of another region.

Our example exposes the two main shortcomings of reflection in component-based systems. Firstly, remote reflection requires that many messages are sent over the network in order to introspect or reconfigure sets of components. In the context of IoT applications, these networks are typically extremely low power and every packet sent imposes a large energy overhead. Previous research has shown that radio transmissions are the primary source of energy consumption on sensor nodes [6]. Secondly, the manager has a too many responsibilities: keeping track of every component in each region, and querying individually every component of a region for every region-wide query. This becomes impractical with large and dynamic regions.

3. TOMOGRAPHY

Tomography is used to inspect or modify a *region* of a component-based application, i.e., a set of connected components. This is achieved by a generalisation of the visitor pattern for distributed systems. A *probe* is broadcasted to a set of *starting components*, which search for the desired region by traversing the components using the dataflow order. It is also possible to traverse the components in the opposite direction, called *upstream tomography*, but in the rest of this paper we will use the dataflow order.

This section begins by defining regions of IoT applications and explaining how they are specified by a manager. It then describes what queries can be performed to inspect and modify connectors, and how these queries are propagated.

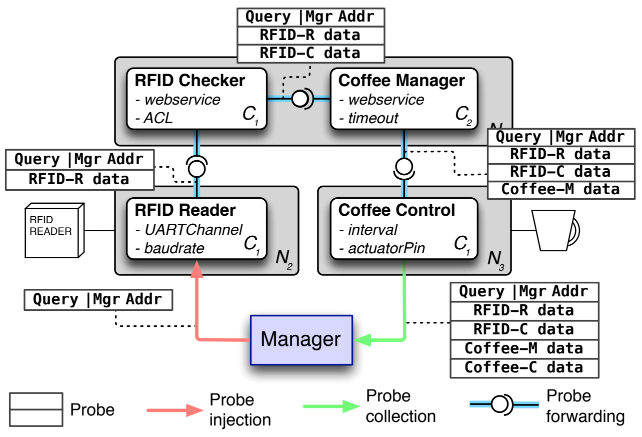

Figure 2: Sending and collecting a probe.

3.1 Regions of components

A *region* is formally a set of connected components. It is specified by: (i) a set of *starting* components, and (ii) a (possibly empty) set of *ending* components. Starting components mark the beginning of the region, which includes all components traversed by following the dataflow direction until either an ending component or to a component without outgoing interfaces. The example in Figure 1 contains 3 different regions, represented by 3 dashed rectangles. The *RFID Check* region, for example, consists of the two components on the left side, and it is specified by indicating that RFID Reader is a starting point and RFID Checker is an ending point of this region. For both of the other two regions in the figure it is enough to specify only the starting point. A finer control of regions is possible by marking interfaces, not components, as starting and ending points, which we do not explore for simplicity.

3.2 Specifying regions

The Manager component is responsible for specifying regions. It starts by identifying the boundaries of a region (starting and ending components) and assigning a unique ID to that region. It then sends a reconfiguration request to the nodes with the selected boundary components to mark them as being starting and/or ending components of the region with the assigned ID. Finally, the Manager stores the region ID and the nodes where the starting components are deployed. This ID and reference nodes are enough for the manager to inspect and reconfigure the full region, explained in more detailed in the following subsection.

Our approach to mark boundaries of regions instead of marking components belonging to regions has two main advantages: *compactness*, as in general less data is needed to specify a region; and *flexibility*, as it becomes easier to adapt regions to newly added components or removed components without large changes to the marking data. Furthermore, the number of messages required to inspect all nodes of a region is typically smaller than when using more naive approaches that query all involved nodes.

3.3 Using tomography

The extra information about regions allows the precise definition of scope for inspection or modification of components. For example, the last two blocks of operations in Listing 1 can be written as in Listing 2. The first sends a query for a list of all properties of the components in the *RFID coffee control* region, and the second sends a request to deactivate all components in the *RFID check* region. In both cases, the manager only sends the query or request to the nodes with starting components, which is here node N_2.

```
// introspection: get properties of coffee control app
coffeeControl.getComponentProperties()
// reconfiguration: deactivate RFID check app
rfidCheck.deactivateComponents()
```

Listing 2: Tomography for reflective operations.

More generally, a tomography sends a query to all components in a region requiring only the starting components to be known in advance, and not the all components of that region. The result of this query is returned back to the manager by every component in an ending point of the region – either a component marked as ending component, or a component with no provided interfaces. The sending of queries is managed by our tomography supported middleware, as illustrated in Figure 2 for our coffee application. The manager wraps the desired query in a *probe* that is sent (injected) to the starting component RFID Reader. This probe contains initially the query and the return address of the manager, and is forwarded by each component in the region until it reaches a dead-end, at which point the probe is collected by the manager. Upon receipt in the meta-space of a component the query is executed, meaning that the inspection operation is applied, its result is appended to the probe, and the reconfiguration operations are executed. The general case with multiple starting points exist and where probes are split during their traversal is explained in detail in the following subsection.

3.4 Probe propagation

This subsection provides more technical details on how the probes are propagated from the starting points until the manager. These probes have to be *split* when traversing a component with multiple provided interfaces and when multiple starting points exist, and the probes must stop the traversal when reaching a component already traversed by a probe with the same query (*merge* of probes).

Splitting probes. When a probe is split into n probes, its header is extended with a new pair containing: (1) the identifier of the component that created the split (or Root if it was the manager), and (2) the number n of splits. This is illustrated in Figure 3. This information is needed by the manager to know when all the probes are collected. Upon collecting a probe, the extra header will describe when was the probe split and in how many probes, allowing it to clearly conclude if there is any split probe missing.

Merging probes. Each probe also includes an unique transaction ID. During the traversal of the probe, each component is marked as being visited by that ID. Hence, if a visited component receives a (split) probe with the same ID, it will simply send it back to the manager without executing its query. Figure 4 illustrates this process. In Step 1 the manager sends split probes to the 2 entry points Comp 1 and Comp 2, who execute its query, are marked as visited,

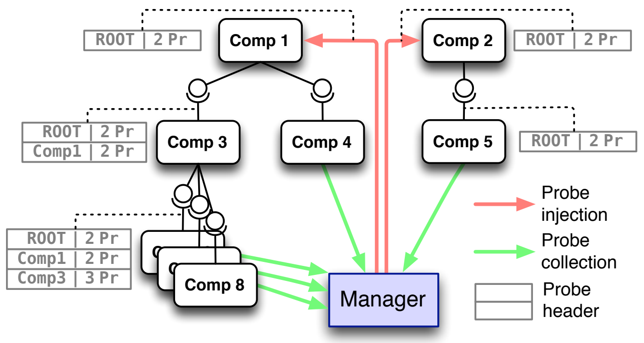

Figure 3: Splitting probes, extending headers of probes to tell the manager when to stop waiting.

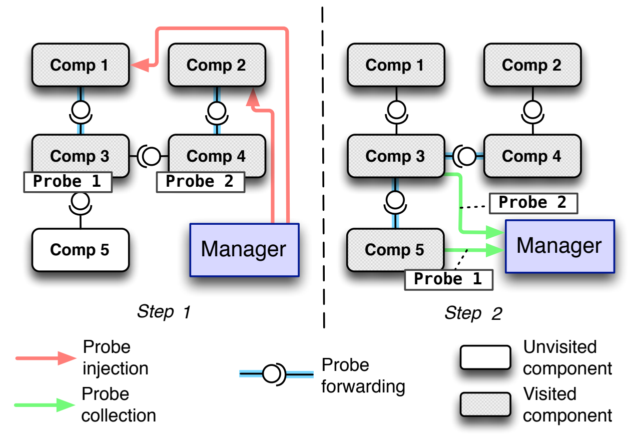

Figure 4: Merging of probes, by marking components as visited.

and forward it to Comp 3 and Comp 4. In Step 2 the probes are propagated further, and component Comp 3 receives the same probe again from Comp 4, consequently returning the probe to the manager without any action.

4. EVALUATION

A preliminary evaluation is done by comparing classic reflective operations with our tomography approach for inspecting and reconfiguring a region of components. The cost of querying a region is measured by counting the number of messages sent over the low-power network (Section 4.1), the cost of setting up a region is measured by the number of messages sent to create a region (Section 4.2), and the cost of managing a region is based on how easy it is for the manager to perform queries and to maintain the required knowledge about regions (Section 4.3). We apply these measurements to our real-world Smart Lab deployment in Section 4.4.

4.1 Number of messages

We claim that, in general, the number of messages required to query a region defined by its starting and ending points is smaller than when the manager contains a list of all components. For example, the query illustrated in Figure 2 uses 4 messages between nodes: a message from the manager to node N_2, 2 messages between the 3 nodes, and a message from N_3 to the manager. A more naive approach where the manager queries all 3 nodes independently would use 8 messages: 4 to perform the query and 4 to collect the result.

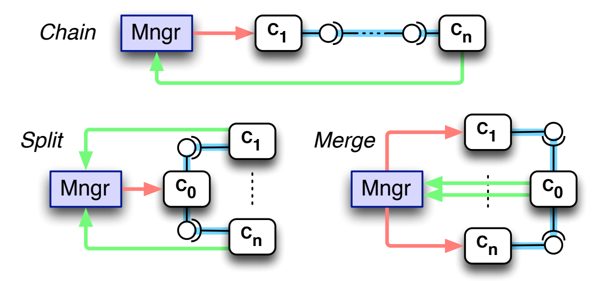

Figure 5: Evaluation scenarios to count messages.

We make our claim more precise by analysing 3 regions with different topologies in Figure 5: a chain of n components, a split of n components, and a merge of n components. Without tomography, querying all components in all these scenarios requires around $n{*}2$ messages ($(n{+}1){*}2$ for the split and merge cases). Using tomography, the chain scenario reduces this number to $n+1$ messages, the split scenario uses a similar number of messages ($(n{*}2)+1$), and the merge scenario increases this number to $n*3$ messages. Summarising, the number of messages is reduced in half with chaining, is not affected with splitting, and is increased by n with merging.

Our experience indicates that chaining is more commonly found in IoT applications than splitting or merging. Indeed, all the examples presented so far exchange a smaller number of messages with tomography than without it. Our larger deployment, described later in Section 4.4, has a region that benefits from tomography only when using upstream tomography, i.e., when traversing components in the oposite direction of dataflow. Furthermore, simple optimisations can avoid the increased number of messages in the merge case. For example, one could allow nodes to wait for probes that could be merged, sending the combined probe to the rest of the chain instead of replying to the manager. This would require extra annotations to nodes and probes, increasing the complexity of modifying bindings within regions without breaking the traversal process of regions.

4.2 Setting up

Setting up a region means updating the nodes with information that describes the region. For tomography this means marking starting and ending components of a region by sending request over the low-power network to their deployment nodes to mark them as such. This is a fixed cost that has to be paid before a region can be queried. Without tomography, no components have to be marked and the setup consists of storing a list of all the components of the region and the nodes where they reside locally on the manager. In this case, there is no overhead on the network.

For tomography, the number of messages required to setup a region depends on its number of boundary nodes. In the *best case* it is enough to mark a single component as a starting component to define a whole region; this is the case for 2 out of the 3 regions in our example in Figure 1, and for the chain and split scenarios in Figure 5. In the *worse case* all components have to be tagged as being starting and/or ending components, producing as many messages as there are components in the region.

4.3 Region management

The biggest advantage of our approach is that, after setting up the regions, the Manager can more easily introspect these with tomography than without it. This claim is supported by 2 performanance indicators: (i) the complexity to query regions, and (ii) the amount of data stored by the manager.

Querying regions. As illustrated in Listing 1 (example without tomography) and Listing 2 (example with tomography), querying explicitly a region with tomography requires less instructions than querying each component individually. The exception for this scenario is when the manager only needs to query part of a region—e.g., all RFID readers in the coffee application—, in which cases it may be more performant to query the desired components individually.

Recalling regions. In order to query a region the manager must know how to reach it. Using tomography, the manager needs to store information about every node with a start component for each region. This means N regions times n_s node addresses with starting components (in average) per region. Without tomography, the manager needs to store information about every node and component in each region. This means N regions times n_c pairs of components and node addresses (in average) per region. The size of this management information is always strictly smaller with tomography, since it does not store component information (only nodes), and only nodes with starting components.

4.4 Smart Lab case-study

This case-study looks at how tomography performs in comparison to classic reflection in a real world case-study, using the 3 previously discussed metrics as performance indicators. Both approaches are benchmarked by introspecting and reconfiguring the component composition shown in Figure 6, which is a subset of a *Smart Lab* deployment in our research facility. This component composition offers 3 services: (i) a *Motion detection* service using data from 6 embedded nodes equipped with motion sensors around the lab, (ii) a *Screen control* service, which allows either remote or local control of screens used for presentations, and (iii) a *Coffee control* service, which was used as a running example throughout this paper and authorises access to a coffee machine through RFID. The component composition has 3 regions grouping the components each of the 3 applications are composed of, denoted by dashed lines in Figure 6.

Message overhead. The number of messages used to inspect and reconfigure regions in our example are presented in Table 1. In most cases tomography requires less messages to be sent over the low-power network when compared to classic reflection. The exception is the *Motion detect* region that is less efficient because of the 6-way merge, following our discussion in Section 4.1. This can be improved by traversing the region upstream, which we call *Motion Detect* ***up*** in the table, turning the merge into a split.

Setup. For tomography, only the starting points need to be tagged at setup time. Assuming we do an upstream tomography for *Motion detect*, setting up all regions in our scenario would cost **3 messages**. In case we traverse *Motion detect* downstream, the set-up cost is **8 messages** due to the multiple start points of that region.

Management overhead. Tomography greatly simplifies the specification of queries over groups of components (Section 4.3). In our case-study a total of **15 queries** are required when using standard per-component reflection to query all components, tomography only requires **1 query**.

Region	**Tomography**		**Reflection**		**Gain**
	Intr.	Recf.	Intr.	Recf.	
RFID Coffee	4	3	8	4	42%
Motion Detect	19	13	16	8	-33%
Motion Detect **up**	14	8	16	8	9%
Screen Control	3	2	6	3	44%
All components	20	12	30	15	29%

Table 1: Message passing overhead

Summarising, tomography imposes a minimal set-up cost and outperforms classic reflective operations both in terms of message passing overhead and management overhead.

5. RELATED WORK

Existing approaches to manage groups of components or nodes in low-power networked systems have been proposed in the literature. This section compares our approach with the notion of *abstract regions* to group nodes, proposed by Welsh et al. [12], and the notion of *component frameworks* to structure groups of components, proposed by Parlavantzas and Coulson [13].

Abstract regions [12] are a family of spatial operators that capture local communication within regions of a network, defined based on properties of the nodes. These operators alleviate development overheads for distributed sensing applications, used to address nodes, to share data in local regions, and to reduce the amount of shared data. Hence abstract regions allow programmers to specify complex distributed sensing applications over regions of nodes using higher-level abstractions than the ones provided at the network level. Both abstract regions and our tomography-based regions try to solve the complexities of distributed embedded software development. The major difference is that abstract regions are defined at the level of computational nodes, and regions are based on properties of these nodes, while tomography are defined at the level of components. Therefore tomography is used to reason about the configuration of component-based applications, and how these components are coordinated, while abstract regions are design to help developing distributed applications with lower-level abstraction than components. Tomography has a more flexible way to specify regions that does not rely on properties of nodes, and focus instead in reducing the overhead of managing components.

Component frameworks (CFs) [13] provide a minimal and abstract component model with mechanisms to combine groups of components (component frameworks) into larger ones. These CFs expose easy-to-use facilities to perform reconfigurations at runtime. Components are primitive building blocks of CFs that support general reconfiguration patterns, and the hierarchal construction of CFs preserves these reconfigurations interfaces, effectively supporting reconfiguration that scale up to large systems. Tomography, unlike CFs, decouples the building of component-based applications from the designing of regions for reconfigurations. It gives the flexibility of building regions at runtime of an existing application based on a simple set of rules, instead of using these regions as building blocks for larger (and recon-

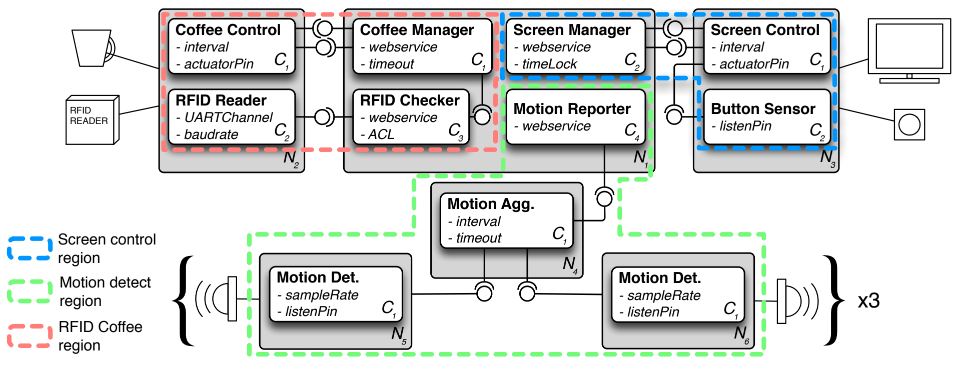

Figure 6: Smart Lab case-study component composition.

figurable) systems. Consequently, the application developer does not decide on the scope of regions for reconfiguration, and only the application manager creates (flexible) regions that can be reconfigured as a group. The idea of building regions based on other existing regions could also be explored in our boundary-based regions, which we consider to be out of the scope of this paper.

6. CONCLUSIONS AND FUTURE WORK

This paper introduced *tomography*, an approach to lower overheads for component-based IoT applications. Tomography provides a way specify regions of connected components and to introspect and reconfigure these based on a distributed variation of the visitor design pattern.

An initial evaluation based on a real-world scenario has shown promising results. Tomography outperforms classic reflection both in terms of message and management overheads, while only imposing a minimal set-up cost.

As future work, we plan to provide a prototype for embedded devices build on top of the LooCI middleware to investigate the performance of tomography on extremely resource contrained devices. Furthermore, we believe that while this initial evaluation is very positive, more improvements can be made to both the granularity of regions and the efficiency of region traversal by storing more information locally.

7. ACKNOWLEDGEMENTS

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

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 *proceeding of NCA*, 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. 1–42, 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 *proceedings of DCOSS*, ser. LNCS. 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*. ACM, 2008, pp. 123–136.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[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] 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.

[12] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *proceedings of NSDI*. USENIX Association, 2004.

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