LooCI: the Loosely-coupled Component Infrastructure

Danny Hughes, Klaas Thoelen, Jef Maerien, Nelson Matthys, Javier Del Cid, Wouter Horré, Christophe Huygens, Sam Michiels and Wouter Joosen
IBBT-DistriNet, KU Leuven, Leuven, B-3001, Belgium.
{firstname.lastname}@cs.kuleuven.be

***Abstract*— Creating and managing applications for Wireless Sensor Networks (WSNs) is complicated by large scale, resource constraints and network dynamics. Reconfigurable component models minimize these complexities throughout the application lifecycle. However, contemporary component based middleware for WSNs is limited by its poor support for distribution. This paper introduces the Loosely-coupled Component Infrastructure (LooCI), a middleware for building distributed component-based WSN applications. LooCI advances the state-of-the-art by cleanly separating distributed concerns from component implementation, supporting application-level interoperability between heterogeneous WSN platforms and providing compatibility testing of bindings at runtime. Together, these features promote the safe and efficient composition and reconfiguration of distributed WSN applications. We evaluate the performance of LooCI on three classes of sensor nodes and demonstrate that these features can be provided with minimal overhead in terms of computation, memory and message passing.**

Keywords: WSN, Components, Middleware

I. INTRODUCTION

Contemporary Wireless Sensor Network (WSN) applications are typically large in scale, requiring the coordination of tens to hundreds of embedded sensor nodes, or 'motes'. Examples of such applications include habitat monitoring [1,2], flood prediction [3], emergency response [4] and surveillance [5]. Future WSN scenarios are expected to involve thousands of nodes. The complexity of WSN environments necessitates middleware support for efficiently developing, deploying and managing large-scale WSN applications. Functionally, WSN middleware needs to play a role in managing application dynamism, which arises from evolving requirements, changing environmental conditions, mobility and unreliable networking. Non-functionally, the resource-constraints of motes [6] must be respected, while the resources that are available on more capable devices [7] and network gateways [8] need to be optimally exploited.

Component models have a strong track record of managing the complexity of developing WSN applications. NesC [9] was the first component model for WSN and is used to implement TinyOS [10], a leading WSN Operating System (OS). The static model of NesC, however, provides poor support for runtime reconfiguration. More recent run-time reconfigurable component models such as OpenCOM [11], RUNES [4], OSGI [17] and REMORA [13] allow for runtime inspection, management and reconfiguration of applications. These reconfigurable component models form part of what we refer to as *component infrastructure middleware*, which also includes a binding model and execution environment.

Contemporary component infrastructure middleware [3,13,14,17] has a number of shortcomings in WSN scenarios. Firstly, distribution concerns are poorly separated from those of component implementation, which limits the extent to which components can be reused in different distributed contexts. Secondly, contemporary component infrastructures offer poor support for analyzing and modifying relationships between cooperating components, which complicates network management. Thirdly, current component models pay little attention to the problem of promoting interoperability between heterogeneous WSN platforms. Finally the discovery and re-use of components is complicated by a lack of support for compatibility testing between component interfaces at runtime.

This paper introduces LooCI: the Loosely-coupled Component Infrastructure (the acronym is pronounced 'Lucy'). LooCI is comprised of a runtime reconfigurable application level component model, a hierarchical type system and a distributed event bus. It provides a clean separation of distribution concerns from component implementation, supports multiple languages and operating systems and provides compatibility testing between component interfaces at bind time. Together, these features promote safe and efficient application development, management and reconfiguration. Building on our previous research [12,18], this paper offers the following unique contributions: (i) a complete description of the platform-independent LooCI component model, (ii) implementations of LooCI for three archetypal WSN platforms, (iii) a detailed memory and performance evaluation of each LooCI implementation and (iv) a quantification of the network overhead of LooCI.

The remainder of this paper is structured as follows: Section II discusses related work, which leads to a set of requirements presented in Section III. In Section IV we describe the design of LooCI, which is evaluated in Section V. We conclude in section VI and discuss future work.

II. RELATED WORK

This section reviews related work in the area of component models for WSNs. Section II.A provides an overview of static component models while Section II.B discusses runtime reconfigurable component models.

A. Static Component Models

NesC [9] is used to implement TinyOS [10]. NesC extends the C language with an event-driven programming

model and mechanisms for explicitly specifying component interfaces. The NesC extensions thus allow the application developer to compose applications from generic and re-usable building blocks. At compile-time, a NesC composition is optimized and compiled to a monolithic block of executable code that can neither be inspected nor modified at runtime. Thus, NesC is a *static* model and provides poor support for scenarios with high levels of dynamism. In terms of distribution, TinyOS [10] provides Active Messages [23], which connects communication and computation by incorporating a reference to an event handler in each message. However, it is not possible to inspect or reconfigure distributed relationships, as these are hard-coded into NesC components. Ports of NesC and TinyOS are available for a range of WSN platforms [19,29], but not for mobile or back-end devices.

B. Reconfigurable Component Models

OpenCOM [11] is a run-time reconfigurable component model that has been applied to build WSN applications [3]. Unlike NesC [9], OpenCOM components remain independent throughout the application lifecycle, which allows the application composition to be inspected and modified after deployment. In terms of distribution, OpenCOM is a strictly local component model. To address this, a number of extensions to the core have been proposed such as the GridKit [3] middleware, which provides support for distribution using the Open Overlays [20] pattern. Open Overlays allows for the creation of flexible distributed interactions. However, the mechanics of distribution are not made explicit to the developer. For example, GridKit [3] uses Java Remote Method Invocation (RMI) [25] to provide distribution, but this introduces an implicit dependency on the RMI registry, which forms a single point of failure that is invisible to the application developer. Ports of OpenCOM are available for a range of mote platforms [8,19,29], mobile devices and standard PCs.

The RUNES component model [4] is a branch of OpenCOM [11] that provides support for WSNs, including additional introspection support in the kernel. RUNES supports the creation of dynamic application compositions but it is a local component model that provides no support for the creation of distributed relationships. Instead, developers must implement their own distribution mechanisms within RUNES components. This limits the extent to which distributed relationships may be reconfigured. As with OpenCOM, RUNES is available for a range of mote platforms, mobile devices and standard PCs.

OSGi [17] provides support for modeling components and application compositions using the Service Component Architecture (SCA) [26]. Components may be inspected and reconfigured at runtime and in addition OSGi provides a secure execution environment. OSGi is a local component model and is tightly coupled with the Java language making it unsuitable for embedded mote platforms that lack a JVM.

REMORA [13] provides a C-like programming language for implementing components and uses SCA [26] to specify component interfaces and application compositions. REMORA supports introspection and reconfiguration, however, it is a local component model that provides no specific support for the creation of distributed relationships. At compile time, the REMORA component implementation language is compiled to byte-code that is executed on the REMORA platform abstraction layer. In comparison to platform agnostic component models [4,11], this prevents the developer from exploiting useful platform-specific features.

III. Requirements

Based on a review of a number of prototypical WSN deployments [1,2,3,5], and the related work presented in Section II, we define the following set of requirements for WSN component infrastructure middleware.

Heterogeneity. In WSN scenarios, heterogeneous sets of hardware are used, including motes [1,2,5], smart phones [1] and embedded Linux boards [3]. Component infrastructure middleware must thus provide *common abstractions* for application development on heterogeneous platforms, while allowing the *strengths of each platform* to be fully exploited.

Interoperability. Interoperability is required between different sensor nodes [1,2] and back-end computational facilities [1,2,3,5]. Component infrastructure middleware must therefore facilitate composition of components running on heterogeneous devices into coherent applications through the use of *common networking* and *data exchange standards*.

Reconfiguration. WSN scenarios demand support for reconfiguration including fine-grained software evolution [1,2,3,5], radical application re-tasking [1,2] and self-adaptation in response to changing environmental conditions [3,5] and application requirements [1,2,3,5]. Component infrastructure middleware must therefore support *reconfiguration of software functionality* and allow efficient *reification* of the current state (i.e. discovery and analysis).

Distributed Relationships. To maximize component reuse in changing topologies [1,2,3,5], a strong separation is necessary between local component implementation and distributed concerns. A lack thereof would after all limit the usefulness of a component to a single network context. Component infrastructure middleware must thus support *flexible binding modalities*, support for *reification of distributed relationships* and a clean *separation between local and distributed functionality*.

Performance. Component infrastructure middleware for WSNs should consume *minimal memory* and offer *good performance*, even on embedded mote platforms [6,19,29]. It should allow for full exploitation of heterogeneous resources, while imposing *minimal burden on the component and application developer*. Such middleware must also promote the development of components that are themselves efficient.

In the following section we introduce LooCI; a WSN component infrastructure middleware designed to handle these requirements.

IV. A Loosely-coupled Component Infrastructure

LooCI is a platform-independent component infrastructure middleware. The architecture of LooCI is shown in Figure 1.

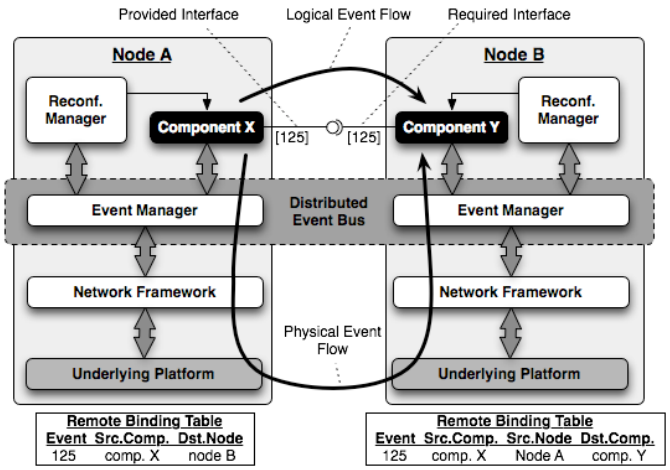

Figure 1: LooCI architecture and bindings.

LooCI allows individual *components* to be deployed on motes at runtime. Components are managed by a *Reconfiguration Manager* and communicate only over a *Distributed Event Bus*. The Reconfiguration Manager maintains references to all local components and enacts incoming deployment, control, introspection and binding commands that are received over the event bus. Using introspection one can discover which components are present on a node along with their interfaces, current state and bindings.

To realize the Distributed Event Bus, each LooCI node implements a local *Event Manager*. The Event Manager maintains local and remote binding tables, containing entries that specify to which local and/or remote components events are forwarded. All subscriptions are locally maintained, eliminating the need for distributed coordination or specialized brokers. While logically components communicate directly by exchanging events via their interfaces, these events actually pass via the Event Manager and, if sent to a remote node, the *Network Framework* and *Underlying Platform*.

The Network Framework standardizes the networking services offered by the Underlying Platform [15,16,17] and offers a uniform API to the upper layers. An extensible set of networking services is provided including network-wide broadcast, one-hop broadcast and unicast. This renders the event bus agnostic to underlying network protocols. It is important to note however that components do not use the Network Framework directly and instead solely communicate over the event bus.

The reconfigurable platform provided by LooCI allows runtime deployment and reconfiguration of components and bindings. To prevent malicious parties from exploiting these features to attack or eavesdrop on the WSN, we have developed a secure deployment protocol [31] [32] and a mechanism for implementing access constraints at the level of component interfaces [27]. Due to space constraints, we refer the interested readers to the original papers.

As LooCI is a platform-independent middleware specification, it provides interoperability across various underlying execution environments. At the time of writing LooCI supports Contiki [15], Squawk [16] and OSGi [17].

A. *The LooCI Reconfigurable Component Model*

LooCI components are individually deployable units of functionality. They are managed via a control API and connect to the event bus through a simple communication API. LooCI components build upon the unit of deployment of the underlying platform (e.g. a Contiki module, a Squawk suite or an OSGi bundle). This allows developers to exploit all features offered by various languages, operating systems and hardware platforms while providing standardized encapsulation, discovery and lifecycle management for components. LooCI should therefore be viewed as an interoperability layer that allows the application developer to compose together software resources that may be distributed across heterogeneous nodes.

Two identifiers are used to specify LooCI components; (1) a user-defined name for the component, and (2) an ID provided by the LooCI runtime that is unique in the context of the hosting node. LooCI components are thus uniquely identified by a `<node address, component ID>` tuple.

LooCI components typically implement fine-grained application level functionality and are composed together into a distributed application via *bindings* over the event bus. For this purpose, each component includes a number of interfaces, which constitute their communication API. *Provided interfaces* describe the events a component may publish to the bus, while *required interfaces* describe events that a component may read from the bus. These interfaces are typed according to the type system discussed in Section IV.B. Bindings logically connect provided interfaces to required interfaces.

Listing 1: Java LooCI Component for Squawk/OSGi

```
public class Aggregator extends LooCIComponent {
  // Declare component
  public Aggregator() {
    super("Aggregator",
      // Declare interfaces
      new byte[]{Types.AGG_SENSOR};// Provided If.
      new byte[]{Types.SENSOR});// Required If.
  }

  //Event handler method
  public void receive(Event event) {}
}
```

Listing 2: C LooCI Component for Contiki

```
//Declare interfaces
COMPONENT_PRO_IFACE(aggregator, AGG_SENSOR);
COMPONENT_REQ_IFACE(aggregator, SENSOR);

//Declare component
COMPONENT(aggregator, "Aggregator");
LOOCI_COMPONENTS(&aggregator);

//Event handler method
COMPONENT_THREAD(temp_filter, ev, data)  {
  COMPONENT_BEGIN();
  while(1) {
    LOOCI_EVENT_RECEIVE(&sensor);
    }
  COMPONENT_END(); }
}
```

Listings 1 and 2 provide example code for a LooCI component providing aggregation. Listing 1 shows Java code, which is compatible with the Squawk [16] and OSGi [17] ports of LooCI. Listing 2 shows a C component for Contiki. To save space, functional code and library imports are omitted. The aggregator component has one required interface of type SENSOR and one provided interface of type AGG_SENSOR. As we build upon existing languages and operating systems, we have been careful to minimize the effort required to declare LooCI components. As shown in Listing 1, the Java-based LooCI ports require 1 line of code to declare a component with interfaces and receptacles. In contrast, the C/Contiki version of LooCI requires 2 lines of code to declare a component, plus one additional line for required interfaces and one line for provided interfaces. On all platforms, components that specify a required interface must also implement an event handler to deal with incoming events.

B. LooCI's Hierarchical Type System

LooCI provides an extensible type system (described fully in [22]) that allows developers to create a shared conceptualization of event types. The LooCI type system organizes all events into a taxonomy that gives semantic meaning to each event. Hierarchical classification also allows for reasoning over groups of functionally equivalent events. The LooCI type system provides the following advantages:

- **Type-safe reconfiguration.** Each interface embeds its position in the type hierarchy. At bind-time, this is used to check type safety and prevent the binding of incompatible interfaces. As compatibility testing is available on all LooCI nodes, this functionality can be used to build self-adaptive systems.
- **Service discovery.** To discover services meeting the dependencies (required interfaces) of a component, a `getProvidedInterfaces` command containing the required type is sent to a single node or group. Nodes that receive this query respond by returning references to all local provided interfaces that are a subtype of that specified in the request.
- **Reduced overhead.** The hierarchical type system reduces the overhead of binding and introspection by allowing the developer to refer to services by class (e,g all `SENSOR` components on a node may be bound to a logger using a single wiring operation, rather than multiple operations to bind each sub-type of `SENSOR`).

The LooCI type system uses an efficient encoding scheme based upon prime numbers, wherein all event types e_i are classified in a hierarchical taxonomy and are associated with a prime number p_i. When an event type e_j is added as a child of e_i, we store a unique identifier uid_j, computed as the product of p_j and uid_i. As such, uid_j uniquely encodes the position of e_j in the hierarchy in relation to its ancestors. At run-time, we can efficiently compute whether an event of type e_j and corresponding uid_j is a subtype of e_i by dividing uid_j by uid_i. If the result of this operation has no remainder, then due to the unique properties of prime numbers, e_j must be a subtype of e_i (a formal proof of this property is provided in [21]). The *uid* of each type is only transmitted during deployment, binding and introspection.

C. LooCI's Distributed Event Bus

The LooCI event-bus is an asynchronous, event based communication medium that follows a decentralized topic-based publish-subscribe model. Events are typed according to the type system described in Section IV.B and can only be published and received by interfaces of compatible types. Application of the publish-subscribe model in LooCI provides loose coupling between components and eliminates the need for costly distributed quiescence protocols [24].

In order to communicate, components must be bound together. This occurs at runtime after deployment of the involved components. *Bindings* are artefacts that are stored in the binding tables of the Event Manager and which explicitly connect provided interfaces of publishing components with compatible required interfaces of subscribing components. Event Managers contain two binding tables; a *local binding table* for bindings between two local components, and a *remote binding table* for distributed bindings between a local and a remote component. Example remote binding tables can be seen in Figure 1. In general, bindings are defined by a `<source event type, source component ID, source address, destination event type, destination component ID, destination address>` tuple. Bindings are only allowed between compatible interfaces, as determined by their types as described in Section IV.B. LooCI supports the following binding modalities:

- **A one-to-one binding** is enacted by sending a `wireTo` event to the source node that identifies a unique provided-interface and a remote destination. A `wireFrom` event is sent to the destination node that identifies the unique source interface and the local destination interface.
- **A one-to-many binding** is enacted by sending a `wireTo` event to the source node that identifies a unique provided-interface and a wild-card destination (network broadcast or one-hop broadcast). A `wireFrom` event is then broadcast to all destination nodes that specifies the source address, a unique source interface and the local destination interface. (Note: a many-to-one binding is established using the inverse set of operations.)
- **Opportunistic bindings** are enacted by sending a `wireTo` event to all source nodes that specifies a provided-interface type and the 1-hop broadcast wildcard. A `wireFrom` event is then sent to the destination node(s) that specifies a wild-card source address, a wild-card source component id, the source interface type and a local destination interface.

It is important to note that components neither send nor receive events until they have been bound and activated. As data is semantically typed and all communication occurs via explicit bindings, introspection may be used to reify distributed relationships and reconfiguration may be used to modify data flows at runtime.

D. API and Management Support

To deploy or reconfigure components and bindings at runtime, the core LooCI API is provided as shown in Listing 3. The *Control*, *Introspection* and *Binding* APIs are exposed by the Reconfiguration Manager over the event bus and made available to back-end system elements via the gateway. As nodes do not have direct access to the component repository, the *Deployment* API is made available only to back-end system elements via the gateway. The core API contains the minimum functionality that is required to manage a network of LooCI nodes and while it may be used directly by the developer, we expect that it will more commonly be used to build higher-level services. Examples of such tools include the Quality Aware Reconfiguration Infrastructure (QARI) [28] and the Policy-based Management Architecture (PMA) [27].

Listing 3: The core LooCI API

Deployment

```
CompID  deploy(ComponentFile, NodeID)
Boolean removeComponent(CompID, NodeID)
```

Control

```
Boolean deactivate(CompID, NodeID)
Boolean activate(CompoID, NodeID)
```

Introspection

```
CompID[]getComponents(NodeID, ComponentType)
String  getComponentType(NodeID, CompID)
State   getComponentState(NodeID, CompID)
Event[] getProvidedInterfaces(NodeID, CompID, UID)
Event[] getRequiredInterfaces(NodeID, CompID, UID)
NodeID[]getOutWires(NodeID, CompID, EventTypeOut)
NodeID[]getInWires(NodeID, CompID, EventTypeIn)
```

Binding

```
Boolean wireFrom(EventTypeOut, SrcCompID,
              SrcNodeID, EventTypeIn, DestCompID,
              DestNodeID)
Boolean wireTo(EventTypeOut, SrcCompID,
              SrcNodeID, DestNodeID)
```

V. Implementation and Evaluation

In this section we introduce three implementations of LooCI and use these to evaluate the design of LooCI. Section V.A quantifies the overhead of LooCI on each of the three evaluation platforms, Section V.B compares LooCI to other reconfigurable component based middleware for WSN and Section V.C assesses the network overhead of LooCI.

Implementations of LooCI are currently available for Contiki, Squawk and OSGi. Table 1 summarizes each LooCI implementation and the hardware platform that was used in this evaluation. All implementations faithfully realize the design described in Section IV. The common type system and event-bus ensures that all ports of LooCI interact seamlessly, allowing for compositions that integrate Contiki [15], Squawk [16] and OSGi [17] motes.

Table 1: LooCI Implementations and Test Platforms

	Contiki	**Squawk**	**OSGi**
Language	C	Java ME	Java SE / OSGi
Environment	Contiki 2.4 [15]	Squawk [16].	Java v1.6
Test Platform	Raven [7] 20MHz ATmega1284p 16K RAM 128KB Flash	SPOT [8] 180MHz ARM920T 512K RAM 4MB Flash	GumStix [9] 400MHz XScale PXA255 16MB RAM 16MB Flash

A. Memory and Performance Overhead of LooCI

The memory overhead of LooCI itself is comprised of the additional flash memory and RAM that is consumed by the execution environment. As can be seen from Table 2, LooCI runs comfortably on all of our test platforms. In the worst case, on the Raven [6], the combined LooCI and Contiki image leaves over 54% of flash memory and 38% of RAM free for application development. On SPOT [7] and GumStix [8], the vast majority of flash and RAM remain available. Along with a compact middleware footprint, it is also important that application components are compact. Table 3 compares the size of functionally equivalent implementations of a LooCI temperature sensor component for Contiki [7], Squawk [8] and OSGi [9].

Table 2: LooCI Middleware Memory Consumption

Static Memory Consumption (Flash)			
	Contiki/Raven	**Squawk/SPOT**	**OSGi/GumStix**
LooCI	16884 bytes	45363 bytes	49987 bytes
Underlying Platform	43180 bytes	616048 bytes	87044 bytes
Flash Overhead	+ 39.10%	+ 7.36%	+57.42%
Total Flash Used	45.83%	15.77%	0.82%
Dynamic Memory Consumption (RAM)			
	Contiki/Raven	**Squawk/SPOT**	**OSGi/GumStix**
LooCI	1561 bytes	31744 bytes	84992 bytes
Underlying Platform	8885 bytes	78848 bytes	445440 bytes
RAM Overhead	+ 17.57%	+ 40.26%	+ 19.08%
Total RAM Used	61.12%	21.09%	3.16%

Table 3 shows that LooCI-Contiki and LooCI-OSGi components consume less RAM than an equivalent Contiki or OSGi module. This is because distribution support is included in the LooCI middleware and does not need to be embedded in the component. LooCI-Contiki also consumes less flash memory while LooCI-OSGi consumes marginally more. In the case of LooCI-Squawk, componentization adds minimal flash overhead, but significant RAM overhead as each component runs an Inter-Isolate RPC [16] server to communicate with the LooCI runtime. However, the 5120 byte RAM overhead is constant and thus becomes less significant as component size increases.

Table 3: LooCI Temperature Sensor Component Memory Consumption

	Contiki/Raven	Squawk/SPOT	OSGi/GumStix
	Static Memory Overhead (Flash)		
Native Application	281 bytes	1740 bytes	1511 bytes
LooCI Component	220 bytes	1843 bytes	1750 bytes
% Change	- 21.71%	+ 5.92%	+ 15.82%
	Dynamic Memory Overhead (RAM)		
Native Application	63	21504 bytes	4588 bytes
LooCI Component	59	26624 bytes	2304 bytes
% Change	- 6.35%	+ 23.81%	- 49.78%

We define performance overhead as the time required to instantiate and bind components. Table 4 summarizes the LooCI performance timings in milliseconds. Initialization and binding operations on LooCI components are fast on all evaluation platforms. Furthermore the LooCI type system allows for checking of type safety at bind time with low performance overhead (in the worst case, 4.8% of bind time).

Table 4: LooCI Middleware Performance Timings

	Contiki/Raven	Squawk/SPOT	OSGi/GumStix
Component Initialization:	0.26 ms	35 ms	1050 ms
Component Binding:	0.15 ms	12 ms	0.12 ms
Component Unbinding:	0.15 ms	12 ms	0.12 ms
Checking Type Safety:	0.03 ms	0.29 ms	0.05 ms

B. Performance Comparison against Reconfigurable Component-based Middleware for WSNs

In Table 5 we identify the distribution mechanisms, languages and hardware platforms supported by competing middleware as well as relevant evaluation metrics that have been reported in the literature.

Table 5: Features of Component-based WSN Middleware

	Distribution	Languages	Hardware	Metrics
GridKit [3]	RPC-based	Java SE	GumStix [8]	Memory & Perf.
Lorien [14]	NONE	C	Telos B [29]	Memory & Perf.
RUNES [4]	NONE	Java ME, C & Contiki	Telos B [29]	Memory & Perf.
OSGi [17]	NONE	Java SE	Various	Memory
LooCI	Event Based Bindings	Java SE, Squawk & Contiki	Raven [7], SPOT [8] & GumStix [9]	Memory & Perf.

As NesC produces a monolithic system image it cannot be meaningfully compared to LooCI and it is therefore omitted. It should be noted from Table 5, that LooCI is the *only* reconfigurable WSN middleware that provides distribution support and platform independence.

Table 6 compares the memory footprint of LooCI with other component based WSN middleware [3,4,13,14,17]. While each middleware provides a subtly different feature set, LooCI-Contiki is the smallest C based component model for WSN and LooCI-Squawk is the smallest Java-based component model. Compared to GridKit [3], the only other component-based middleware that supports distribution, all versions of LooCI have a smaller memory footprint.

Table 6: Flash Footprint of WSN Component Models

	Language	Size (Bytes)
Lorien	C	21794
RUNES	C	20000
LooCI-Contiki	C	16884
LooCI-Squawk	Java ME	45363
LooCI-OSGi	Java SE	49987
OSGi	Java SE	76800
GridKit	Java SE	106700

Table 7 compares the performance of component-based WSN middleware in terms of component initialization and binding as reported in [3,4,14]. Note: these timings were obtained on a variety of hardware platforms as outlined in Table 5 and therefore should not be directly compared without taking into account platform capability. Where performance data is omitted, it was not provided in the source paper.

Table 7: Performance of WSN Component Models

	Language	Bind	Init.
Lorien	C	N/A	799ms
RUNES	C	N/A	0.98ms
LooCI-Contiki	C	0.26ms	0.30ms
LooCI-Squawk	Java ME	35ms	24ms
LooCI-OSGi	Java SE	0.24ms	1050ms
GridKit	Java SE	80ms	118ms

Compared to the C-based component models, instantiation in LooCI-Contiki is three times faster than RUNES [4] and three orders of magnitude faster than Lorien [14]. However the compute power of the LooCI test platform (Raven [7] at 20 MIPS) is 25% greater than the Lorien and RUNES evaluation platforms (Telos B [29] at 16 MIPS).

Compared to the Java-based GridKit middleware [3], LooCI-Squawk [8] offers faster component initialization and binding, despite being evaluated on a mote providing less than half of the compute power (SPOT [8] provides 180 MIPS, while GumStix [9] provides 400 MIPS), however caution is again warranted due to differences in the supporting operating system. Comparing the performance of LooCI-OSGi to GridKit on GumStix [9], the results are mixed. LooCI offers two orders of magnitude faster component binding, but one order of magnitude slower component instantiation (due to start up of inter-isolate RPC [16]). In our view, instantiation speed is less important than

binding speed, as instantiation is typically infrequent compared to binding operations.

C. Overhead of LooCI Networking Approach

In this section we quantify the worst-case network overhead (i.e. with the maximum possible size of message payloads) introduced by LooCI in terms of number of bytes that must be transmitted to enact each LooCI command. This analysis does not take into account network dynamics and thus represents the worst-case scenario for an ideal one-hop network with no collisions or packet loss. In terms of the application messages that are transmitted between components, the LooCI messaging format adds a fixed overhead of 4 bytes per message. As all commands in the LooCI API (provided in Listing 3) are implemented using a simple request/reply scheme, the *worst-case* number of bytes transmitted to enact any LooCI command is given by: $Bytes = n * (P_{RQ} + P_{RP})$. Where n is the number of nodes targeted, P_{RQ} is the size of the request message and P_{RP} is the size of the reply message. Table 8 provides the P_{RQ} and P_{RP} costs for all commands in the LooCI API and can thus be used to calculate the worst-case number of bytes transmitted for all commands.

Table 8: Network Overhead of LooCI Commands

	P_{RQ} (Bytes)	P_{RP} (Bytes)
`deploy()`	*CS +4*	*5*
`activate()`	*5*	*5*
`deactivate()`	*5*	*5*
`removeComponent()`	*5*	*5*
`getComponents()`	*28*	*C+6*
`getComponentType()`	*5*	*29*
`getComponentState()`	*5*	*6*
`getProvidedIfaces()[TU]`	*5*	*2I + 6*
`getProvidedIfaces()[TS]`	*ST+4*	*I (ST+2)+6*
`getRequiredIfaces()[TU]`	*5*	*2I+6*
`getRequiredIfaces()[TS]`	*ST+4*	*I(ST+2)+6*
`getOutWires()`	*7*	*IA+6*
`getInWires()`	*7*	*T(A+1)+6*
`getLocalWires()`	*7*	*I+6*
`wireFrom()[TU]`	*A+8*	*5*
`wireFrom()[TS]`	*A+TS+8*	*5*
`wireTo()[TU]`	*A+7*	*5*
`wireTo()[TS]`	*A+TS+7*	*5*

CS is the size of the component to be deployed, *C* is the number of components matching an associated query, *I* is the number of interfaces matching the associated query, *A* is the size of a network address in bytes and *ST* is the worst case size of a semnatic type description. As type safety in discovery and binding may be deactivated, the overhead of both type unsafe [TU] and type safe [TS] operations is provided. For current implementations, the maximum size of A is 16 bytes for a full IPv6 address and the maximum size of TS is 22 bytes for a 65,000 element type system (in realistic scenarios, TS is likely to be far smaller [22]). All LooCI commands are compact and in normal cicumstances requests and responses will fit comfortably within a single 802.15.4 packet, with the possible exception of type safe discovery messages that return a large number of matching interface descriptions.

As taxonomic data and component names are only used to support discovery and binding, this data is not stored in the binding table of the Event Manager. Thus, each incoming wiring entry in this table consumes only 5+A bytes, while each outgoing wiring entry consumes only 4+A bytes. This allows complex patterns of component communication to be created, inspected and reconfigured while working within the resource constraints of contemporary mote platforms.

VI. Conclusions and Future Work

This paper introduced the Loosely-coupled Component Infrastructure (LooCI). LooCI is the first platform independent, distributed component infrastructure middleware for WSN. LooCI offers a language independent reconfigurable component model and standard support for: encapsulation of functionality, runtime reconfiguration and management of software components running on heterogeneous platforms. LooCI strongly separates the concerns of component implementation and distribution, allowing components to be safely reused in a range of distributed contexts. Our analysis shows that LooCI provides a richer feature set than other reconfigurable WSN middleware, with minimal performance overhead.

Our primary avenue of future work will be to validate LooCI in a large-scale WSN application using heterogeneous motes. We have previously implemented applications using prototypes of LooCI [12], but these were of small scale and implemented on a homogeneous hardware deployment [18]. We believe that large-scale, heterogeneous LooCI deployments are now required to validate the LooCI model and to evaluate our networking approach in a realistic environment.

Acknowledgements

This research is partially funded by the Inter-University Attraction Poles Programme Belgian State, Belgian Science Policy and by the Research Fund KU Leuven.

References

[1] R. Szewczyk, A. M. Mainwaring, J. Polastre, J. Anderson and D. E. Culler, "An analysis of a large scale habitat monitoring application", in proc. of 2nd ACM conference on Embedded Network Sensor Systems (SenSys'04), Baltimore, MD, USA, pp. 214-226, Nov. 3rd-5th 2004.

[2] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, C. Mascolo, B. Pasztor and S. Scellato, N. Trigoni, R. Wohlers, K. Yousef, "Evolution and sustainability of a wildlife monitoring sensor network", in proc of 8th ACM Conference on Embedded Networked Sensor Systems (Sensys'10), Zurich, Switzerland, pp. 127-140, Nov. 3rd-5th 2010.

[3] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith and K. Beven, "An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring", in Inter-Science Journal on Concurrency and Computation: Practice and Experience, Vol. 20, No. 11, pp. 1303-1316, 2008.

[4] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G.P. Picco, T. Sivaharan, N. Weerasinghe and S. Zachariadis, The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario, in proc. of 5th IEEE conference on Pervasive Computing (PerCom'07), White Plains, NY, pp. 69–78, , Mar. 19th-23rd 2007.

[5] N. Finne, J. Eriksson, A. Dunkels and T. Voigt, "Experiences from two sensor network deployments: self-monitoring and self-configuration keys to success", in proc. of 6th international conference on Wired/wireless internet communications (WWIC'08), Tampere, Finland, pp.189-200, May 28th-30th 2008.

[6] AVR RZ Raven Data Sheet, available online at: http://www.atmel.com/dyn/resources/prod_documents/doc7911.pdf, [retrieved 18th Jan. 2012].

[7] SUN SPOT, Theory of Operation, available online at: http://www.sunspotworld.com/docs/Yellow/SunSPOT-TheoryOfOperation.pdf, [retrieved 18th Jan. 2012].

[8] GumStix Connex, Legacy Product Information, available online at: http://www.gumstix.org/hardware-design/legacy-products.html, [retrieved 18th Jan. 2012].

[9] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer and D. Culler, "The NesC Language: A Holistic Approach to Networked Embedded Systems", in proc. of the ACM confernece on Programming Language Design and Implementation, SIGPLAN PLDI'03, San Diego, CA, USA, pp. 1 – 11, Jun. 9th-11th 2003.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors", in ACM SIG-PLAN, Vol. 35, No. 11, pp. 93–104, 2004.

[11] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama and T. Sivaharan T, "A generic component model for building systems software", in ACM Transactions on Computer Systems, Vol. 26, No. 1, pp. 1-42, 2008.

[12] D. Hughes, K. Thoelen, W. Horré, N. Matthys, S. Michiels, C. Huygens and W. Joosen, "LooCI: a loosely-coupled component infrastructure for networked embedded systems", in proc. of 7th international conference on advances in mobile computing & multimedia (MoMM'09). Kuala Lumpur, Malaysia, pp. 195-203, Dec. 14th-16th 2009.

[13] A. Taherkordi, F. Loiret, R. Rouvoy, A. Abdolrazaghi, Q. Le-Trung and F. Eliassen, "Programming Sensor Networks Using REMORA Component Model", in proc. of 6th IEEE/ACM Conference on Distributed Computing in Sensor Systems (DCOSS'10), Santa Barbara, CA, USA, pp. 22-28, Jun. 21st-23rd 2010,

[14] B. Porter, U. Roedig and G. Coulson, "Type-Safe Updating for Modular WSN Software", in proc. of 7th IEEE conference on Distributed Computing in Sensor Systems (DCOSS '11), Barcelona, Spain, pp. 1-8, Jun. 27th-29th 2011.

[15] A. Dunkels, B. Grönvall and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors", in proc. of 29th IEEE conference on Local Computer Networks (LCN'04), Tampa, FL, USA, pp. 455–462, Nov. 16th-18th 2004.

[16] D. Simon, C. Cifuentes, D. Cleal, J. Daniels and D. White, "Java on the Bare Metal of Wireless Sensor Devices: the Squawk Java Virtual Machine", in proc. of the 2nd International Conference on Virtual Execution Environments, Ottawa, Canada, pp. 78–88, Jun. 14th-16th 2006.

[17] J. Rellermeyer and G. Alonso, "Concierge: A Service Platform for Resource-Constrained Devices", in ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, pp. 245–258, 2007.

[18] D. Hughes, K. Thoelen, W. Horré, N. Matthys, S. Michiels, C. Huygens, W. Joosen and J. Ueyama, "Building Wireless Sensor Network Applications with LooCI", in International Journal of Mobile Computing and Multimedia Communications (IJMCMC), Vol. 2, No. 4, pp. 38-64, 2010.

[19] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization", in UC Berkeley Technical Report, 2002.

[20] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson and F. Taiani, "Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity", in proc. of the European Conference on Computer Systems (EuroSys'08), Glasgow, UK, pp. 123-136, Mar. 31st- Apr. 1tst 2008.

[21] D. Preuveneers and Y. Berbers, Encoding Semantic Awareness in Resource-Constrained Devices, in IEEE Intelligent Systems, Vol. 23, No. 2, pages 26-33, pp. 1541-1672, 2008

[22] K. Thoelen, N. Matthys, W. Horré, C. Huygens, W. Joosen, D. Hughes, L. Fang and S. Guan, "Supporting Reconfiguration and Re-use through Self-Describing Component Interfaces", in proc. of international Workshop on Middleware for Sensor Networks (MidSens'10), Bangalore, India, pp. 29-34, Nov. 29th - Dec. 3rd 2010.

[23] P. Buonadonna, J.Hill and D. Culler, "Active Message Communication for Tiny Networked Sensors", in proc. of the 20th annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'01), Anchorage, Alaska, USA, pp. 1-11, Apr. 22nd-26th 2001.

[24] P. Grace, G. Coulson, G.S. Blair, B. Porter and D. Hughes, "Dynamic Reconfiguration in Sensor Middleware", in proc. of 1st International Workshop on Middleware for Sensor Networks (MidSens '06), Melbourne, Australia, pp. 1-6, Nov. 28th 2006.

[25] Java Remote Method Invocation (RMI), available online at: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, [retrieved 18th Jan. 2012].

[26] Service Component Architecture (SCA), available online at: http://osoa.org/pages/viewpage.action?pageId=46, [retrieved 18th Jan. 2012].

[27] N. Matthys, R.S. Afzal, C. Huygens, D. Hughes S. Michiels, W. Joosen, "Towards fine-grained and application-centric access control for wireless sensor networks", in proc. of 25th Symposium on Applied Computing, Sierra, Switzerland, pp. 793-794, Mar. 22nd-26th 2010.

[28] W. Horré, D. Hughes, S. Michiels, W. Joosen, "Advanced sensor network software deployment using application-level quality goals" in Journal of Software, Vol. 6, No. 4, pp. 528-535, 2011.

[29] MoteIV, T-Mote Sky Data Sheet, available online at: http://www.snm.ethz.ch/Projects/TmoteSky, [retrieved 18th Jan. 2012].

[30] G. Coulson, D. Hughes, G. Blair, P. Grace, "The Evolution of the GridStix Wireless Sensor Network Platform" in proc. of the International Workshop on Sensor Network Engineering (IWSNE '08), co-located with DCOSS '08, Santorini, Greece, June 2008, pp. 1-6.

[31] J. Maerien, S. Michiels, C. Huygens, W. Joosen,, "MASY: Management of secret keys in federated wireless sensor networks", in proc. of 6th Int. Conf. on Wireless and Mobile Computing, Networking and Communications (WiMob '10), vol., no., pp.121-128, 11-13 Oct. 2010

[32] J. Maerien, S. Michiels, C. Huygens, W. Joosen, "Sasha: A distributed protocol for secure application deployment in shared ad-hoc wireless sensor networks", in proc. of 8th IEEE conference on Mobile Adhoc and Sensor Systems (MASS '11), vol., no., pp.43-48, 17-22 Oct. 2011.