# LooCI: A Loosely-coupled Component Infrastructure for Networked Embedded Systems

*Danny Hughes* *Klaas Thoelen* *Wouter Horré*
*Nelson Matthys* *Javier Del Cid* *Sam Michiels*
*Christophe Huygens* *Wouter Joosen*

*Report CW 564, September 2009*


Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# LooCI: A Loosely-coupled Component Infrastructure for Networked Embedded Systems

*Danny Hughes* *Klaas Thoelen* *Wouter Horré*
*Nelson Matthys* *Javier Del Cid* *Sam Michiels*
*Christophe Huygens* *Wouter Joosen*

*Report CW 564, September 2009*

Department of Computer Science, K.U.Leuven

**Abstract**

Considerable research has been performed in applying run-time reconfigurable component models to the domain of wireless sensor networks. The ability to dynamically deploy and reconfigure software components has clear advantages in sensor network deployments, which are typically large in scale and expected to operate for long periods in the face of node mobility, dynamic environmental conditions and changing application requirements. To date, research on component and binding models for sensor networks has primarily focused on the development of specialized component models that are optimized for use in resource-constrained environments. However, current approaches impose significant overhead upon developers and tend to use inflexible binding models based on remote procedure calls. To address these concerns, we introduce a novel component and binding model for networked embedded systems (LooCI). LooCI components are designed to impose minimal additional overhead on developers. Furthermore, LooCI components use a novel event-based binding model that allows developers to model rich component interactions, while providing support for easy interception, re-wiring and advanced features like distributed garbage collection and generic fault tolerance. A prototype implementation of our component and binding model has been realised for the SunSPOT platform. Our preliminary evaluation shows that LooCI has an acceptable memory footprint and imposes minimal overhead on developers.

**Keywords :** Wireless Sensor Networks, Component Models, Binding Models, Publish-Subscribe.
**CR Subject Classification :** D.2.12

# LooCI: a Loosely-coupled Component Infrastructure for Networked Embedded Systems

Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid,
Sam Michiels, Christophe Huygens, Wouter Joosen

IBBT-Distrinet, Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Leuven, Belgium.

{ firstname.lastname } @ cs.kuleuven.be

## ABSTRACT

Considerable research has been performed in applying run-time reconfigurable component models to the domain of wireless sensor networks. The ability to dynamically deploy and reconfigure software components has clear advantages in sensor network deployments, which are typically large in scale and expected to operate for long periods in the face of node mobility, dynamic environmental conditions and changing application requirements. To date, research on component and binding models for sensor networks has primarily focused on the development of specialized component models that are optimized for use in resource-constrained environments. However, current approaches impose significant overhead upon developers and tend to use inflexible binding models based on remote procedure calls. To address these concerns, we introduce a novel component and binding model for networked embedded systems (LooCI). LooCI components are designed to impose minimal additional overhead on developers. Furthermore, LooCI components use a novel event-based binding model that allows developers to model rich component interactions, while providing support for easy interception, re-wiring and advanced features like distributed garbage collection and generic fault tolerance. A prototype implementation of our component and binding model has been realised for the SunSPOT platform. Our preliminary evaluation shows that LooCI has an acceptable memory footprint and imposes minimal overhead on developers.

## Categories and Subject Descriptors

D.2.12 [**Interoperability**]: Distributed objects – *component and binding models*

## General Terms

Languages

## Keywords

Wireless Sensor Networks, Component Models, Binding Models, Publish-Subscribe

## 1. INTRODUCTION

Wireless Sensor Networks (WSNs), composed of embedded computers equipped with low power radios and low-cost sensors are being employed to support a growing range of fixed and mobile applications such as habitat monitoring [1], flood warning [2], industrial process control [3] and disaster management [4]. WSNs are typically large in scale, subject to unreliable networking, node mobility, high risk of node failure and are expected to operate unattended for long periods. Recently, lightweight component models [4] [5] [6] [7] have emerged as a promising approach to managing complexity in WSN environments. However, these models have a steep learning curve and impose a significant burden on the developer. Furthermore, the binding models used in these systems are primarily based on traditional Remote Procedure Call (RPC) approaches, which do not adequately reflect the dynamism of sensor network environments. RPC approaches also require that developers explicitly specify relationships between single nodes, rather than modelling interactions between implicit groups (e.g. neighbours or networks) and explicit groups (e.g. nodes belonging to a specific organisation). Thus, RPC-style communication does not scale effectively in unreliable network environments. In addition, in mobile scenarios with high rates of churn, RPC interaction models require that developers deploy complex fault-tolerance functionality to deal with intermittent connectivity.

This paper introduces the Loosely-coupled Component Infrastructure (LooCI), which features a loosely-coupled, *event-based* binding model inspired by event-driven programming models [6], Service Oriented Architectures (SOA) [8], publish-subscribe interaction models [9] and pluggable networking support [10]. The resulting architecture is light-weight and promotes a loose coupling between software components while facilitating advanced features such as *generic fault tolerance* and *distributed garbage collection*.

The remainder of this paper is structured as follows. Section 2 discusses component models for WSNs. Section 3 discusses binding models for WSNs. Section 4 presents the LooCI middleware. Section 5 evaluates an initial implementation of LooCI. Section 6 discusses directions for future work. Finally, section 7 concludes.

## 2. COMPONENT MODELS FOR WSN

A number of lightweight component models have been proposed for networked embedded scenarios including: NesC [6], OpenCOM [7], RUNES [4] and OSGi [5]. We discuss each of these below.

NesC [6] is perhaps the best known and most widely deployed component model for WSN and is used to implement the TinyOS operating system [12]. NesC provides an event-driven programming approach together with a *static* component model. The NesC binding model is based upon statically declared bi-directional component interfaces. Unlike OpenCOM [7], RUNES [4] or OSGi [5], NesC components cannot be dynamically re-wired to support reconfiguration and adaptation. However, the

static programming approach used in NesC allows for whole-program analysis and optimization [6], which is advantageous in resource constrained WSN environments. In terms of remote communication, TinyOS provides an implementation of the Active Messages paradigm [13]. Active Messages integrate communication and computation by incorporating a reference to a user-handler in each message. This allows for an event-based handler invocation model and prevents processes blocking while waiting for incoming messages. While TinyOS itself provides no support for remote bindings, extensions have been proposed to support traditional RPC-type bindings [14].

OpenCOM [7] is a general purpose, run-time reconfigurable component model. While OpenCOM does not target WSN applications specifically, it is used to implement the GridStix [15] sensor network platform. OpenCOM features a compact run-time kernel that supports both static and dynamic compositions. In the case of static compositions, the kernel performs component linking at run-time and then exits, reducing overhead. In dynamic systems, the kernel persists and may be used to support rich run-time reconfiguration. OpenCOM also offers a higher level of abstraction, known as Component Frameworks (CFs) [7], which are used to model interactions between cooperating components. CFs may be local or distributed and can be used as a tool to support dynamic reconfiguration. In the case of distributed component frameworks, a Meta Object Protocol (MOP) allows reconfiguration actions to be applied to groups of components. While OpenCOM notionally supports diverse binding types, all current instantiations use RPC bindings. Furthermore, while the OpenCOM component model is rich, the OpenCOMJ component model alone consumes 52KB of RAM, while the complete WSN profile consumes 104KB [10], significantly more memory than is available on popular embedded sensor motes such as the T-Mote [16] or Mica-Z [17] platforms.

The RUNES [4] middleware brings OpenCOM functionality to more embedded devices. The RUNES model has been realized in C and Java and as with OpenCOM, RUNES allows for the use of different binding types, however, all current implementations use RPC. RUNES also adds a number of introspection API calls to the OpenCOM kernel and has achieved a significantly smaller footprint than OpenCOMJ, as reported in [10]. The C and Java versions of RUNES consume less than 20KB [4] of memory.

The OSGi component model [5] targets powerful embedded devices such as smart phones and network gateways along with desktop and enterprise computers. OSGi provides a secure execution environment, support for run-time reconfiguration, life-cycle management and various system services. OSGi interfaces are modeled using SCA [23] and, as with OpenCOM [7] and RUNES [4], OSGi offers RPC-based bindings. Unfortunately, while OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [5] consumes more than 80KB, making it unsuitable for very resource constrained devices.

# 3. INTERACTION MODELS FOR WSN

In [19] Parlavantzas et al. present a component-based model that can be used to define diverse forms of component binding including: RPC, publish/subscribe, shared data spaces and pipes. In this model, binding types are expressed as UML *collaborations* which are modeled using 11 standard components. While this model is very flexible, this comes at a significant cost in terms of complexity and, as will be argued in section 4.4, only a subset of binding types fit well with WSN environments.

In [14] May et al. present an RPC extension to NesC [6] which allows for remote procedure calls (RPC) with similar semantics to local NesC calls. These RPC calls also support a simple one-hop service discovery scheme. This scheme allows developers to call methods on a neighbor, rather than targeting calls to a specific mote address. This reduces the burden on developers, who need not deal solely in terms of individual motes, and allows for simple fault tolerance in the presence of node failure or high levels of mobility. When an RPC call is made to a node's neighbors, the call will be served by exactly one neighbor, in an anycast fashion. This approach however has a number of significant limitations. Firstly, the lack of a common event or service description model complicates discovery of third party services and monitoring of interactions. Secondly, this approach still requires that developers model component interactions primarily on a mote-by-mote basis, which we believe is inefficient, if not infeasible, in large-scale or mobile WSNs.

Jini [26] provides an event based service oriented architecture for Java which leverages on RMI. Jini improves on the RMI registry by making the lookup service distributed and allowing clients to search for services based on their type, name or description. While Jini promotes service discovery and re-use, the underlying network implementation is based on RMI and TCP/IP, which are a poor fit with unreliable WSN network environments. Furthermore, the smallest Jini implementation [26] has a footprint of over 1MB, making it unsuitable for highly resource constrained devices.

TeenyLIME [27] provides a tuple space abstraction in which processes communicate by writing and reading tuples into a shared virtual memory space. To support mobile applications, the tuple space in TeenyLIME is only shared by one-hop neighbours. Although multi-hop communication is possible by traversing multiple tuple-spaces, this limits the scope of possible interactions between software components. Furthermore, in contrast with event based approaches, tuple spaces require active polling to receive data updates. This leads to higher communication overhead than event based approaches.

# 4. THE LOOCI MIDDLEWARE

The LooCI middleware is designed for Java devices such as the Sun SPOT [11] and Sentilla Perk [21]. As these platforms support standard Java ME [22], they reduce the burden on developers compared to bespoke WSN technologies such as TinyOS [12] or Contiki [18]. In a broader sense, the new generation of Java sensor motes opens the field of WSN development to millions of existing Java ME developers [22]. LooCI aims to bring the many advantages of reconfigurable, component-based software to Java-based WSN platforms, while preserving the benefits of familiarity and ease-of-use that Java ME offers. LooCI accomplishes this through the introduction of an easy-to-use *component model*, a simple yet extensible *networking framework* and a common *event bus* abstraction. The event bus provides a common mechanism to connect software components for any kind of data exchange. These features are discussed in sections 4.1 to 4.3 respectively. Section 4.4 then discusses how LooCI may be used to compose distributed applications. A simplified overview of the LooCI middleware is shown in Figure 1.

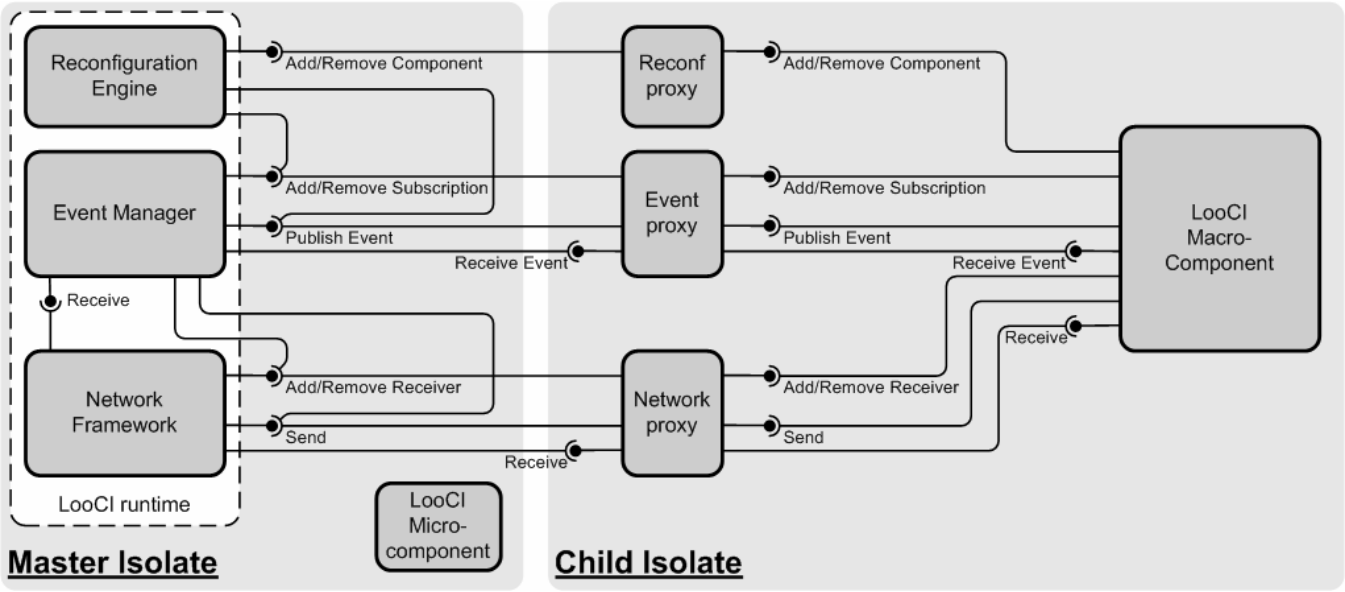


**Figure 1: The Loosely Coupled Component Infrastructure (LooCI)**

## 4.1 The LooCI Component Model

LooCI offers support for two component types, *macrocomponents* and *microcomponents*:

*Macrocomponents* are coarse-grained and service-like, building upon the notion of Isolates [25] inherent in the Sentilla [21] and SQUAWK [11] virtual machines. Isolates are process-like units of encapsulation and provide varying levels of control over their execution (exactly what is provided is dependant on the specific JVM). LooCI standardizes and extends the functionality offered by Isolates. Each macrocomponent runs in a separate isolate and communicates with the run-time middleware via Inter Isolate RPC (IIRPC), which is offered by the underlying VM. Unlike microcomponents, macrocomponents may use multiple threads and utility libraries.

*Microcomponents* are fine-grained and self-contained. All microcomponents run in the master Isolate alongside the LooCI runtime. Unlike macrocomponents, microcomponents must be single threaded and self contained, using no utility libraries. Aside from these restrictions, microcomponents offer identical functionality to macrocomponents with lower memory consumption

Both macrocomponents and microcomponents offer *run-time reconfiguration*, *interface definitions*, *introspection* and support for the *re-wiring* of bindings. Each LooCI component has a unique identifier which is generated based upon the interfaces and dependencies of the component (see sections 4.1.2 and 4.3.1).

### *4.1.1 Run-time Reconfiguration and Introspection*

In order to implement a LooCI component, developers extend a generic component base-class, which provides standard methods to START and STOP a component as well as to place the component into quiescent state (MAKE_QUIESCENT) and RESUME operation from quiescent mode. For simple component implementations the control methods provided by the base-class are sufficient, however, for components with more complex requirements, developers may override these methods with their own implementations of START, STOP, MAKE_QUIESCENT and RESUME. Extending the component base-class also provides implicit access to the LooCI Network Framework and Event Bus.

LooCI components may be deployed on demand via a LooCI application known as the Network Manager (see Figure 2), a maintenance application that has responsibility for every node in its associated WSN. Upon deployment, all LooCI components register with the per-node *reconfiguration engine*. The reconfiguration engine maintains a reference to all LooCI components running on the node and exposes a remote reconfiguration interface on the event bus. Thus, run-time reconfiguration may be easily enacted by LooCI components running on sensor nodes, gateways or back-end devices.

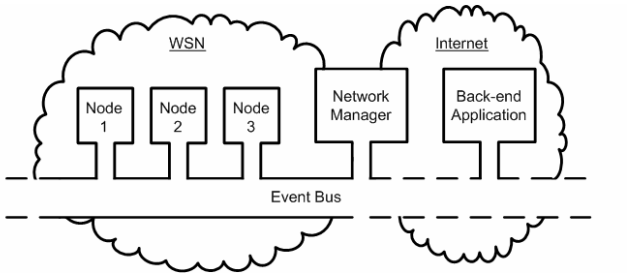


**Figure 2: WSN and Back-End Integration**

As LooCI components extend and build upon common Java concepts, they have a significantly reduced learning curve for standard Java or Java ME developers. Indeed, in the simplest case, converting a standard Java ME CLDC 1.1 application into a LooCI component requires only that a single ‘extends’ statement

be modified and that one method call - `initialize()` be added.

The reconfiguration engine provides introspection at the *node*, *component* and *binding* level. At the *node* level, it is possible to inspect the LooCI components that are deployed on a mote. At the *component* level, the state of components can be inspected (STARTED, STOPPED, QUIESCENT etc.). At the *binding* level, the addresses to which a component is bound can be inspected. As with run-time reconfiguration, introspective information is accessed via the *reconfiguration engine* over the *event bus*.

### *4.1.2 Interface Definitions and Wiring*

LooCI components define their *provided interfaces* as the set of LooCI events that they publish. The *receptacles* of a LooCI component are similarly defined as the events to which they subscribe. Each LooCI event has a globally unique identifier which classifies the event in terms of a global descriptive hierarchy (described in section 4.3).

Wiring of components occurs after deployment via the LooCI *reconfiguration engine*, which exposes WIRE and UNWIRE operations. These mechanisms take as their argument a LooCI Component ID, an Event ID and a network address to which the specified event should be bound or unbound. LooCI addresses may map to a single node, a group of nodes, or an entire WSN. Wiring can be inspected at any time through the introspection API and re-wiring may be enacted by any element on the event bus.

As LooCI components communicate indirectly over the event bus, it is easy to build complex and flexible relationships between individual nodes, groups of nodes, or entire networks. Furthermore, re-wiring operations are low cost and easily enacted. As stated previously, this loose coupling is well suited to mobile environments. Furthermore, in conjunction with a global event classification, the event bus allows for advanced features like *generic fault tolerance* and *distributed garbage collection* as will be explored in section 4.4.3.

## 4.2 Supporting Network Framework

The LooCI network framework abstracts over and extends the networking services provided by the various underlying sensor platforms [11] [21] and provides a simple, uniform API to the upper middleware layers and applications. This abstraction is supported by an extensible set of networking components, as shown in Figure 3.

The initial set of networking components consists of a *UnicastComponent* (UC) which provides both reliable and unreliable point-to-point multi-hop communication, a *BroadcastComponent* (BC) which implements network-wide broadcast and a *NeighbourcastComponent* (NC) which implements one-hop broadcast. The interface to the upper layers is agnostic to the various underlying communication paradigms and provides a simple *send(message, destination) interface*. Based upon the IPv6 address provided and flags in the message header, the network framework automatically selects the most appropriate networking component to dispatch the message, as shown in Figure 3.

Incoming messages are passed from the underlying JVM to the Receiver component which in the case of reliable unicast responds with an ACK. As with Active Messages [13], LooCI messages are dispatched to the upper layers using an event-based handler invocation model. This approach avoids the need for connection setup and encourages programmers to think in an event-based fashion that we believe is appropriate for WSN (we explore this issue further in section 4.4). In the future, we intend to provide a more detailed description of the LooCI *network framework*, however, we consider such a description outside of the scope of this paper.

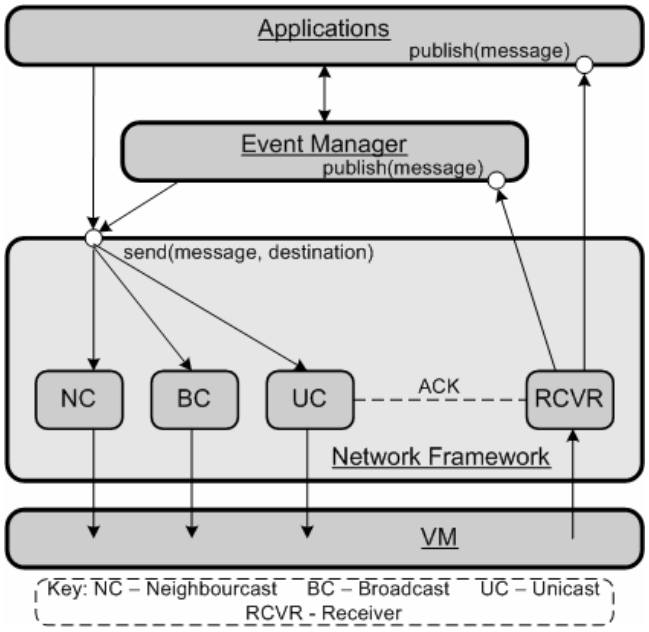


**Figure 3: Supporting Network Framework**

## 4.3 The LooCI Event Model

The LooCI Event Model embodies a generic communication substrate for disseminating events of any kind. This includes, but is not limited to: sensor readings, reconfiguration messages and state information. In concert with the networking framework described in the previous section, the event manager forms a distributed *'Event Bus'* to which all LooCI components are connected. A per-node instance of the LooCI Event Manager implements a simple topic-based publish-subscribe event model, wherein events are disseminated to subscribers based upon their type. For example, a software element may subscribe to events of type 'TEMPERATURE' and may then be wired to a component at a given network location (local, a remote node or a remote group of nodes) that produces these events. The interface to the event bus is simple and lightweight, however, in concert with LooCIs global, hierarchical type system (discussed in section 4.3.1) the event manager allows for rich modeling of interactions between nodes.

Available components and event publishers may be discovered directly, by interrogating the introspection interface of individual nodes or, more commonly, through the NetworkManager of a WSN (a LooCI application which provides aggregated

information about, and control over, all nodes in the associated WSN).

LooCI events are defined based upon a global, hierarchical name space, or taxonomy, represented by a spanning tree. The root of the tree is the base 'EVENT' type and each successive layer of child nodes describes events with increased specificity. Each node in this tree is assigned a unique two-byte identifier based upon the order in which this event was added to the hierarchy. This disadvantage of our current scheme is that it limits the size of the LooCI name-space to 65,536 unique events. However, the overhead of transmitting event IDs is thus predictable and acceptable at 2 bytes per event. In a general sense, hierarchical classification has a number of advantages in terms of component re-use and binding flexibility. In terms of *re-use*, a global type system makes it possible for developers to easily discover deployed 3rd party components suitable for use in their compositions. This prevents the unnecessary deployment of redundant functionality and is thus critical to conserving resources in an embedded environment. In terms of *binding flexibility*, the hierarchical classification allows developers to easily subscribe to groups of events. For example, a temperature monitoring component may include temperature conversion functionality, and thus would subscribe to all events of type TEMP, rather than specifying TEMP_°C and TEMP_°F. In the future, we hope to transition to a more efficient taxonomy encoding scheme such as that presented in [29].

A hierarchical classification also allows for automatic optimization of compositions at deploy-time. For example, as part of a composition, a developer may request deployment of a TEMP_°C component, however, a TEMP_°F component already exists at the desired location. In this case the deploying entity may choose not to deploy the requested component (which would consume additional resources), but instead to wire a lightweight TEMP_CONVERSION component into the provided composition and connect this to the existing TEMP_°F component. While this example is trivial, we believe that in an infrastructure sensor network which supports multiple applications, automatic optimization of compositions may hold significant benefits.

## 4.4 The LooCI Binding Model

Section 4.1 – 4.3 introduced the supporting elements of the LooCI binding model in which interfaces specify relationships between components on the event bus. This section now specifically discusses the suitability of this binding model for supporting programming in WSN and other networked embedded systems. Section 4.4.1 makes the case for loosely coupled, standardized bindings. Section 4.4.2 gives examples of how this binding model is used. Finally, section 4.4.3 gives a preliminary sketch of how we intend to exploit this binding model to support advanced features like *fine grained security*, *generic fault tolerance* and *distributed garbage collection*.

### *4.4.1 The Argument for Event-Based Bindings*

As previously described, LooCI bindings are formed by the subscription of one component to the events generated by another component. Combined with our flexible addressing scheme and connectionless network model, the result is a very loose coupling between components. LooCI bindings are implicit, distributed and multi-party.

- *Implicit:* while concrete, bindings need not be represented by a specific component. Interfaces are defined by the event types that a component publishes and subscribes to.
- *Distributed:* local or remote bindings are semantically identical, allowing components to be easily bound to local or remote resources, whether this is a single node, a group of nodes or a whole network.
- *Multi-party:* unlike traditional RPC-based approaches, which require that relationships be modeled between single nodes, LooCI bindings allow for rich interactions between nodes, groups and networks.
- *Asynchronous:* event publishers do not block while producing events and subscribers are notified asynchronously when an event is received. This is an excellent fit with unreliable, resource constrained WSNs.

The *implicit* nature of LooCI interfaces reduces the burden on developers. This somewhat reduces the learning curve inherent in writing LooCI components and brings our model closer to standard Java ME CLDC 1.1. The *distributed* nature of our bindings allows for rich interactions between network entities to be modeled. This is in contrast to the static, local component model of NesC [6] and a significant improvement over the RPC-like bindings offered by OpenCOM and RUNES which can be used to build relationships only between single motes. LooCI components support group-bindings that are richer than those provided by May et al. [14], and more lightweight than the web services approach employed by Pohl et al. [3].

In a general sense, we believe that the characteristics of WSN data flows are a good fit with the features of publish-subscribe interaction models [9]. However, unlike traditional publish-subscribe systems such as Jini [25] that require specialized service brokers, LooCI is entirely decentralized, allowing any node to act as an event broker, which is critical in mobile environments where network segmentation is possible. Through the provision of a per-WSN *Network Manager*, that supports aggregate control and introspection, we believe that LooCI balances the benefits of publish-subscribe systems with the characteristics of real world WSNs and is a clear improvement over interaction models.

### *4.4.2 Example Bindings*

This section provides some simple examples which show how our binding model can be used to support the creation of distributed software compositions for a mobile warehouse monitoring scenario. Each example shows how the necessary bindings may be realized using LooCI API calls. Each binding is illustrated in Figure 5.

In this scenario, a company, 'STORAGE_CO' uses a WSN to monitor the location of packages stored in their warehouse using RF-based localization. A sensor mote is installed in each package and the LooCI middleware runs on both WSN motes and the back-end systems of STORAGE_CO.

***Example 1 - Tracking a Suspicious Package:*** In this example, a specific package has been tagged as 'suspicious' by a STORAGE_CO employee and its location will thus be monitored

as it moves through the warehouse, until a customs officer arrives to inspect it. To support this, the SUSPECT_TRACKING component will dispatch a WIRE event to the address of the mote in the identified package. This event specifies that the LOCATION_EVENT produced by a LOCATION_COMPONENT running on this mote should be wired to the address of the mote hosting the VISUALISATION_COMPONENT, which will then begin to receive location telemetry from the suspect package. This is an example of how a simple *one-to-one* binding may be created:

```
wire(TARGET_ADDRESS, LOCATION_COMPONENT,
        LOCATION_EVENT, VISUALISATION_ADDRESS,
        VISUALISATION COMPONENT);
```

***Example 2 – Emergency Data Logging:*** In this example, STORAGE_CO has been tasked to safely store sensitive materials, for which they must provide an unbroken location audit trail. To handle the possibility of disconnection from back-end logging systems during periods of mobility, STORAGE_CO also deploys an in-network LOGGING_COMPONENT which logs SENSOR events, storing them to flash. When the node moves out of range, the LOCATION_COMPONENT will use LooCI's *introspection* facilities to discover all LOGGING_COMPONENTs available on neighboring nodes, storing them to an array of addresses (logAddresses). The LOCATION_COMPONENT will then wire its LOCATION events to the addresses of each discovered LOGGING_COMPONENT:

```
for (each address in logAddress)  {
        wire(local_address, LOCATION_COMPONENT,
                LOCATION_EVENT, remote_address
                LOGGING_COMPONENT);
        }
```

As a 'LOCATION' event is a child of the 'SENSOR' event type in the global event classification (described in section 4.3), the LOGGING_COMPONENT is implicitly subscribed to LOCATION events. This is an example of how *introspection*, coupled with a global type system can be used to support decentralized *service discovery* and *re-use*. It is also an example of a one-to-many binding. Furthermore, the LOGGING_COMPONENT provides a simple example of how hierarchical event types allow for the creation of flexible components, such as a generic logging service.

***Example 3 – Filtering location data:*** Following deployment of a WSN and location-tracking software, STORAGE_CO discovers that their location data is subject to intermittent interference and thus inaccuracy. STORAGE_CO software engineers develop a filtering algorithm that can weed-out bogus location data. In order to install the filtering component, the NetworkManager is used to discover the address of LOCATION_COMPONENTs, which is stored to an array (publishers). The NetworkManager then discovers all addresses to which these LOCATION_COMPONENTs are bound:

```
for (each pub_address in publishers)  {
  subscribers = discoverBindings(pub_address,
        LOCATION_COMPONENT, LOCATION_EVENT);
  }
```

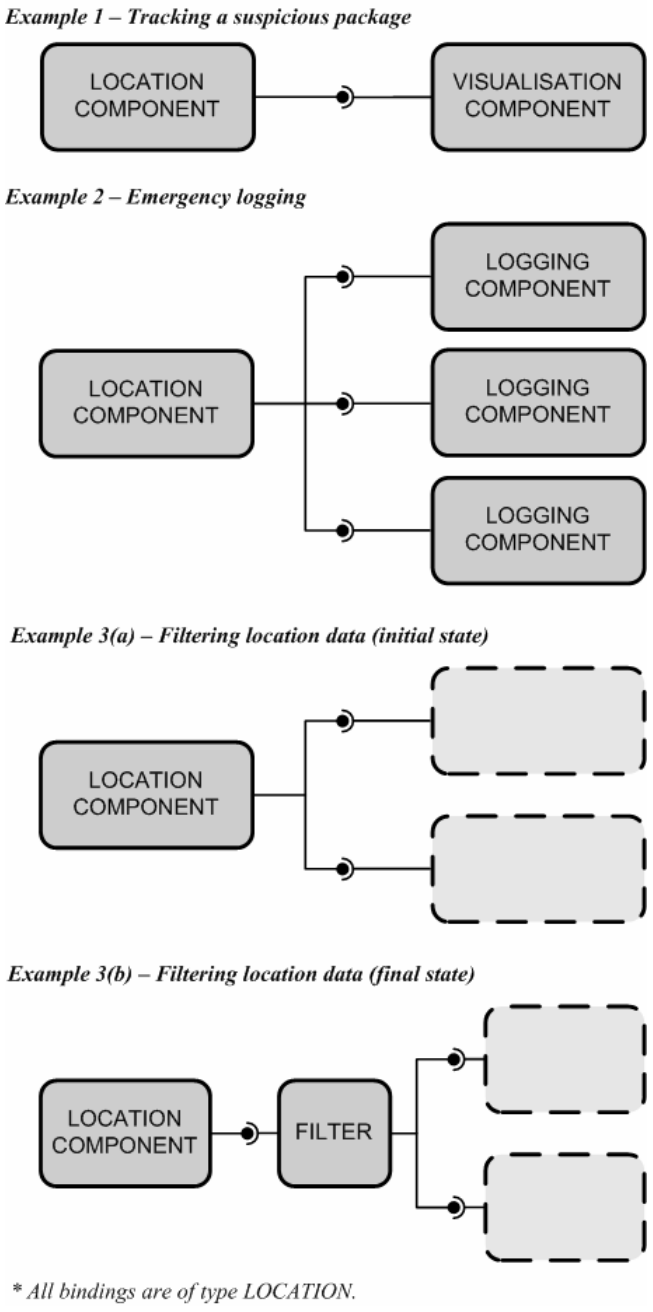


**Figure 5: Example LooCI bindings**

A FILTER_COMPONENT is then deployed on each mote hosting a LOCATION_COMPONENT (The FILTER_COMPONENT depends upon, and produces

LOCATION events) and the LOCATION_COMPONENT is then rewired to the FILTER_COMPONENT:

```
deploy(pub_address, FILTER_COMPONENT);
wire(pub_address, FILTER_COMPONENT,
        LOCATION_EVENT, pub_address,
        LOCATION_COMPONENT);
```

Each subscriber is then unwired from all current bindings and re-connected to the deployed FILTER_COMPONENT:

```
for (each sub_address in subscribers)  {
        unwire(sub_address, LOCATION_COMPONENT,
                LOCATION_EVENT, pub_address,
                sub_component);
        wire(sub_address, FILTER_COMPONENT,
                LOCATION_EVENT, pub_address,
                LOCATION_COMPONENT);
        }
```

Thus all components that were previously bound to a LOCATION_COMPONENT and receiving unfiltered LOCATION events are now bound to a FILTER_COMPONENT producing filtered LOCATION events. This is an example of how dynamic component re-wiring can be used to modify the functionality of an existing composition. This example also shows how the loosely coupled event-bus abstraction supports easy *interception* of events.

Each of these simple case-study examples has been implemented. The compactness of their implementation is analyzed in terms of source lines of code in section 5.3. Each example binding is illustrated in Figure 5.

### *4.4.3 Supporting Advanced Features*

This section briefly discusses how the LooCI component and event model can be used to support three forms of advanced functionality: *fine-grained security, distributed garbage collection* and *generic fault tolerance*.

LooCI leverages existing in-network security mechanisms to provide secure communications. However, the component and event model of LooCI also facilitates *fine-grained security* for more complex usage scenarios departing from the monolithic application model. The emerging view of the WSN as long-lived multi-purpose infrastructure coupled with the LooCI component and event model promotes scenarios where multiple applications run on a shared WSN infrastructure, and federation, where applications span WSNs owned by different parties. These scenarios require specific security support as sharing infrastructure implies competition for available resources. The per-WSN Network Manager is closely tied to the WSN infrastructure and its owner, and is thus ideally positioned to evaluate component deployment requests from third party applications. To support fine grained security in a shared infrastructure scenario, the per-node Event Manager, under control of the WSN administrator, may be extended with a policy driven enforcement engine, which evaluates wiring and reconfiguration requests locally before distribution via the event bus. Since wiring to a large extent defines resource usage, information in the wiring request (source/destination component, source/destination address, event type) can be used to inform access policy decisions. Wiring attempts that violate access policies will be refused and the application that requested the wiring will be signaled with an exception. The policy decision can be made node-local if the wiring remains within an administrative domain, or redirected to a remote decision component when the application spans federated domains. Ultimately, we envisage policy-driven interception engines will be injected in the underlying event based communication substrate, possibly as components, to provide additional assurance by enforcing security at critical locations – for example at the ingress of administrative domains where events cross federation boundaries.

*Distributed garbage collection:* this paper has argued that WSNs are increasingly being required to support multiple applications and are being viewed as long-lived infrastructure. Run-time reconfigurable component-based middleware is a good fit for this usage scenario as it allows for the modification of existing functionality along with the dynamic deployment of new functionality. However, as applications are retired and their constituent components are no longer needed, they must be removed from the network to prevent a build-up of 'orphan' components that waste resources. However, in multi-user scenarios, it may not be in the interests of a developer to invest significant time in component clean up. Alternatively, applications may crash, and go off-line in an unplanned fashion. Even where the removal of components is attempted, network failure, such as mobile nodes moving out of range, may result in incomplete clean-up of components.

Fortunately, the use of a global event type system together with indirect bindings over the event bus allows for the possibility of *distributed garbage collection*. This functionality is provided by the NetworkManager application, which, as previously stated, has responsibility for a single WSN. The NetworkManager periodically polls the introspection interface of each mote and component to gather deployment and wiring information. The network manager will also inspect the introspection interface of each remote location to which components are bound. Where no live dependent components are found, the potentially orphaned component will be flagged and, after a given time-out, will be garbage collected, releasing the resources it was using.

An equally significant problem is the possibility that a mote fulfilling a binding will fail, move out of range, or become inaccessible, leaving a dependency unmet. Fortunately, the use of a global type system also allows for the implementation of *generic fault-tolerance*. As with *distributed garbage collection*, this functionality is provided by the Network Manager, which maintains a list of the bindings associated with components in the WSN. The network manager will periodically poll the remote components participating in these bindings and, where a component appears to have failed, the manager will attempt to repair the binding. In the simplest case, this may involve rewiring to an alternative component. Alternatively, the network manager may request the deployment of a new component that produces the required event. In this way it is possible to provide generic fault tolerance for LooCI applications.

Distributed garbage collection and generic fault tolerance provide specific examples of the broader advantages inherent in maintaining simple and consistent communication abstractions. These services lower the burden on the developer, while allowing her intentions to be more closely honored by the middleware.

# 5. EVALUATION

A prototype implementation of LooCI has been realized in Java ME for the Sun SPOT platform, running the 'BLUE' version of the SQUAWK JVM, implementing Java ME CLDC 1.1. This section now provides a preliminary evaluation of LooCI. Section 5.1 examines the footprint of LooCI. Section 5.2 examines the performance of LooCI. Finally, section 5.3 investigates the overhead that working with LooCI components imposes on developers.

## 5.1 Middleware Footprint

It is particularly critical that middleware for embedded systems maintains a minimal memory footprint. This section compares LooCI to the GridKit middleware [10] (which uses OpenCOMJ [7]) and RUNES [4].

The complete LooCI implementation may be most directly compared to the GridKit WSN middleware [10] which, like LooCI, offers network functionality along with support for component based software development and dynamic reconfiguration through OpenCOMJ [10]. However, it is also possible to compare a subset of the LooCI implementation (component model and reconfiguration support) to the Java implementation of the RUNES [4] component model. Table 1 below compares the footprint of LooCI with GridKit and RUNES as reported in [10] and [4] respectively.

**Table 1 – LooCI Footprint Comparison**

| | LooCI | GridKit | RUNES |
|---|---|---|---|
| **Core:** | 20.8K | 52.4K | 15.5K |
| **Networking:** | 23.4K | 51.8K | N/A |
| **Total:** | 44.3K | 104.2K | N/A |

As Table 1 shows, the LooCI component model has a footprint of just 20.8K, slightly more than the RUNES component model at 15.5K but significantly smaller than OpenCOMJ at 52.4K. Furthermore, the *event bus* network abstraction offered by LooCI consumes less than half of the memory of the GridKit networking framework while offering a more flexible communication abstraction. We believe that through further optimization, it will be possible to significantly reduce the footprint of LooCI while offering the same functionality.

LooCI components also have a minimal footprint. A null LooCI macrocomponent requires *686 bytes* of disk space, while a null microcomponent consumes just *587 bytes*, which is similar to a RUNES component at *544 bytes* (figures are not available for an equivalent OpenCOMJ component). We believe that further reductions in LooCI component size may be possible through optimization of the base component class.

**Table 2 – LooCI Memory Usage**

| | | Total RAM Used | |
|---|---|---|---|
| **SunSPOT SDK:** | (a) No Applications | 77K | |
| | (b) 1 Application | 98K | |
| | | **Macro.** | **Micro.** |
| **LooCI:** | No components | *84K* | *84K* |
| | 1 component | *123K* | *86K* |
| | 2 components | *150K* | *88K* |
| | 3 components | *176K* | *90K* |

Table 2 compares the memory consumption of the LooCI middleware to the standard SunSPOT SDK (blue version). The memory consumed by a dummy component running in the master isolate (a) may be directly compared to the LooCI middleware (core and networking) running with no components registered. Thus it can be seen from the table that the LooCI run-time consumes only 8K of RAM. The table also shows that running a standard SunSPOT component in a child isolate (b) increases memory consumption by 20K. As macrocomponents are instantiations of an isolate, we can expect a similar increase for each macrocomponent added. Our evaluation however shows an increase of 39K for the first macrocomponent and 26K for each additional macrocomponent. The disparity in overhead between SunSPOT applications and macrocomponents can be explained by the instantiation process of the inter-isolate RPC server and its proxies. Table 2 also shows that microcomponents have a much smaller memory footprint at 2K per component. Finally, it should be noted that these experiments were performed using the Solarium management tool provided as part of the SunSPOT SDK. As Solarium provides over the air monitoring of motes, it would be expected to consume a non-negligible amount of memory on each mote. However, Solarium memory consumption is constant for all experiments and thus the relative differences in memory usage shown in Table 2 remain accurate.

## 5.2 Performance Overhead

We evaluated LooCI on a standard SunSPOT mote (180MHz ARM9 CPU, 512KB RAM, SQUAWK VM 'BLUE' version) using the Solarium management application. We logged the time required to initialize the LooCI run-time, the time required to initialize a null LooCI component and the time required to send and receive an event. In each case, we performed 10 experiments. Table 3 shows the average performance characteristics observed in these experiments.

From Table 3 it is clear that even on an embedded platform such as the Sun SPOT, the time required to initialize the LooCI run-time and LooCI components is not prohibitive, especially as such operations are likely to be infrequent. Microcomponents take significantly longer to initialize as they are delivered in an isolate and must first be transferred to the master isolate for execution. The performance of event dissemination for microcomponents is good; however, the performance of event publication for macrocomponents is slower due to the overhead of IIRPC calls in the SQUAWK 'BLUE' JVM.

**Table 3 – LooCI Performance**

| | Time (ms) | |
|---|---|---|
| **Run-time Init:** | 498ms | |
| | **Macro.** | **Micro.** |
| **Null-component Init:** | 35ms | 738ms |
| **Event Publication:** | 14ms | 4ms |
| **Event Reception:** | 14ms | 4ms |

## 5.3 Overhead for Developers

In this section, we revisit the example components and bindings introduced in section 4.4.2. Each component is analyzed in terms of Source Lines of Code (SLoC). While SLoC is an imperfect metric for assessing development overhead, we believe that the results we have obtained are fair and representative. In brief, the functionality of each component is summarized below:

- *NULL_COMPONENT:* the null component contains no functional code and has no interfaces or receptacles.
- *LOCATION_COMPONENT:* the location component publishes a LOCATION_EVENT on the event bus.
- *LOCATION_VIEW_COMPONENT:* the location view component subscribes to the LOCATION_EVENT, displaying location data.
- LOG_COMPONENT: the log component subscribes to the SENSOR_EVENT, storing it to flash.
- LOCATION_FILTER_COMPONENT: the location filter component subscribes to LOCATION_EVENT and publishes a filtered LOCATION_EVENT.

**Table 4 – SLOC of Example Components**

| **Component** | **Source Lines of Code** | |
|---|---|---|
| | **Functional** | **Component** |
| NULL | 0 | 8 |
| LOCATION | 12 | 8 |
| LOCATION_VIEW | 10 | 8 |
| LOG_COMPONENT | 11 | 8 |
| LOCATION_FILTER | 21 | 11 |

As can be seen from Table 4, implementing LooCI components does not impose unreasonable overhead on developers in terms of source lines of code.

# 6. FUTURE WORK

Our future work will focus upon four major areas: (i) extension of the LooCI network framework, (ii) porting LooCI to the Sentilla Perk platform, (iii) implementation of distributed garbage collection and generic fault tolerance strategies and (iv) development of a component parameterization model.

The current implementation of the LooCI networking framework provides support for *unicast*, *broadcast* and *neighbourcast* as discussed in section 4.2. However, we believe that support for more flexible group messaging and multicast are essential to support efficient communication in challenging network environments. We also intend to develop an implementation of LooCI for the Sentilla Perk platform which has more challenging resource constraints [21]. We will then deploy a large test-bed of heterogeneous LooCI nodes (Sun SPOTS [11] and Perk motes [21]) in order to test the performance of loose event couplings in such a large scale setting with simulated node mobility. Another key objective in the short-term is to realize and evaluate the *distributed garbage collection* and *generic fault tolerance* functionality introduced in section 4.4.3. These approaches will be evaluated in terms of their network overhead and potential benefits in realistic case-studies. Perhaps the most critical area of future work is that of developing a model for *component parameterization,* which is necessary to deal with the differing requirements of dependent components. A simple example of the need for parameterization becomes apparent if one considers the case of two components which depend on temperature readings. COMPONENT_A requires that readings be produced every minute while COMPONENT_B requires that readings be produced every 10 seconds. Clearly, modeling this functionality as two separate components is wasteful. To avoid this, a common method is required to parameterize component behavior and to negotiate an optimal configuration where remote components request conflicting parameterizations.

# 7. CONCLUSIONS

This paper introduced LooCI, a novel component and binding model for WSNs and other network embedded systems. Key features of LooCI are support for *run-time reconfiguration*, *introspection* and low-overhead for Java developers.

LooCI bindings are loosely coupled and indirect, operating over the LooCI event bus. We have shown how the proposed component and binding model may be used to model rich interactions between single motes, groups of motes or entire WSNs. We argue that a loosely coupled, globally typed event bus is a good fit for implementing bindings in WSN environments that are inherently asynchronous and unreliable. We have also shown how a simple, consistent abstraction can be used to support advanced features like *distributed garbage collection* and *generic fault tolerance.*

Critically, the LooCI accomplishes these goals while offering good performance and maintaining a minimal memory footprint and offering good performance. Crucially, we show that realizing LooCI components requires very little additional when overhead compared to writing standard Java ME code, which we hope will promote the adoption of LooCI with Java ME developers.

# 8. ACKNOWLEDGEMENTS

Research for this paper was partially funded by IMEC and the Instituut voor de Aanmoeding van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT). This research is conducted in the context of the IWT-STADIUM project No. 80037 [28].

# 9. REFERENCES

[1] Mainwaring A., Polastre J., Szewczyk R., Anderson J., Wireless Sensor Networks for Habitat Monitoring, in proc. of 1st ACM International Workshop on Wireless Sensor Networks and Applications, Atlanta, Georgia, USA, 2002, pp 88 – 97.

[2] Hughes D., Greenwood P., Coulson G., Blair G., Pappenberger F., Smith P., Beven K., An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring, in Wiley Inter-Science Journal on Concurrency and Computation: Practice and Experience, vol. 20, no 11, November 2007, pp 1303-1316.

[3] Pohl A., Krumm H., Holland F., Stewing F. J., Lueck I., Service-Orientation and Flexible Service Binding in Distributed Automation and Control Systems, in proc. of the 22nd International Conference on Advanced Information Networking and Applications – Workshops (IANA), Okinawa, Japan, March 2008, pp. 1393 - 1398

[4] Costa P., Coulson G., Gold R., Lad M., Mascolo C., Mottola L., Picco G.P., Sivaharan T., Weerasinghe N., Zachariadis S., The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario, in proc. of the 5th Annual IEEE International Conference on Pervasive Computing and Comunications (PERCOM'07), White Plains, New York, March 2007, pp. 69 – 78.

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

[6] Gay D., Levis P., Von Behren R., Welsh M., Brewer E., Culler D., The nesC Language: A Holistic Approach to Networked Embedded Systems, in proc. of the conference on Programming Language Design and Implementation, ACM SIGPLAN 2003, San Diego, California, USA, pp. 1 – 11.

[7] Coulson G., Blair G., Grace P., Taiani F., Joolia A., Lee K., Ueyama J. and Sivaharan T, A Generic Component Model for Building Systems Software, in ACM Transactions on Computer Systems, Vol. 26, No. 1, Feb 2008.

[8] Bell M., Introduction to Service-Oriented Modeling, Service-Oriented Modeling: Service Analysis, Design, and Architecture, Wiley & Sons. , 2008, pp. 3.

[9] Eugster P. T., Felber P. A., Guerraoui R., Kermarrec A. M., The Many Faces of Publish Subscribe, in ACM Computing Surveys (CSUR), Vol. 35 , No. 2, June 2003, pp. 114 – 131.

[10] Grace P., Hughes D., Porter B., Blair G., Coulson G., Taiani F., Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity, in proc. of the European Conference on Computer Systems (EuroSys'08), Glasgow, Scotland, UK, March 2008, pp. 123-136.

[11] Sun Microsystems, Small Programmable Object Technology, "Inspiring Java developers to create a whole new breed of devices and technologies - and accelerating the growth of the 'Internet of Things'": http://www.sunspotworld.com/vision.html

[12] Hill J., Szewczyk R., Woo A., Hollar S., Culler D., Pister K., System Architecture Directions for Networked Sensors, in ACM SIGPLAN, Vol. 35, No. 11, November 2000, pp. 93-104.

[13] Buonadonna P., Hill J., Culler D., 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, April 2001.

[14] May T. D., Dunning S. H., Hallstrom J. O., An RPC Design for Wireless Sensor Networks, in proc. of the IEEE International Mobile Adhoc and Sensor Systems Conference, (MASS'05), Washington, DC, USA, November 2005, pp. 138.

[15] Smith P., Hughes D., Beven K., Cross P., Tych W., Coulson G., Blair G., Towards the Provision of Site Specific Flood Warnings using Wireless Sensor Networks, in Wiley Inter-Science journal on Meteorological Applications, vol. 16, no.1, January 2009, pp. 57 – 64

[16] MoteIV, T-MOTE Sky Ultra-low Power Wireless Module Data Sheet: http://www.cs.uvm.edu/~crobinso/mote/tmote-sky-datasheet-102.pdf

[17] Crossbow, MICA-Z Wireless Measurement System, Data Sheet: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf

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

[19] Parlavantzas N., Coulson G., Blair G., An Extensible Binding Framework for Component-Based Middleware, in proc. of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), Brisbane, Australia, September 2003, pp 252.

[20] Huygens C., Joosen W., Federated and Shared Use of Sensor Networks through Security Middleware, in proc. of the 6th International Conference on Information Technology: New Generations (ITNG'09), Las Vegas, Nevada, USA, April 2009, pp. 1005-1011.

[21] Sentilla, Perk Platform Frequently Asked Questions: http://www.sentilla.com/perk_faq.html

[22] Sun Microsystems, Java ME - the Most Ubiquitous Application Platform for Mobile Devices: http://java.sun.com/javame/index.jsp

[23] IONA et al., Service Component Architecture: Building Systems using a Service Oriented Architecture: www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf

[24] Grace P., Coulson G., Blair G., Porter B., Hughes D., Dynamic Reconfiguration in Sensor Middleware, in the proceedings of the 1st International Workshop on Middleware for Sensor Networks (MidSens'06), Melbourne, Australia, November 2006, pp. 1 – 6.

[25] Simon D., Cifuentes C., Cleal D., Daniels J., White D., Java on the Bare Metal of Wireless Sensor Devices: the Squawk Java Virtual Machine, in the proceedings of the 2nd International Conference on Virtual Execution Environments, Ottawa, Canada, June 2006, pp 78 – 88.

[26] Jini-Based Ubiquitous Computing Middleware Supporting Event and Context Management Services in Lecture Notes in Computer Science, Volume 4159/2006, pp. 786-795

[27] Costa P., Mottola L., Murphy A.L., Picco G.P , Programming Wireless Sensor Networks With the TeenyLime Middleware, in the proceedings of the ACM/IFIP/USENIX International Conference on Middleware, (Middleware'07), Newport Beach, California, December 2007, pp. 429-449.

[28] IWT Stadium project 80037, software technology for adaptable distributed middleware: http://distrinet.cs.kuleuven.be/projects/stadium/

[29] Preuveneers D., Berbers Y., Encoding Semantic Awareness in Resource-Constrained Devices, in IEEE Intelligent Systems, Vol. 23, No. 2, pp. 26-33, March 2008