Agimone: Middleware Support for Seamless Integration of Sensor and IP Networks

Gregory Hackmann, Chien-Liang Fok, Gruia-Catalin Roman and Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{ghackmann, liang, roman, lu}@wustl.edu

Abstract

The scope of wireless sensor network (WSN) applications has traditionally been restricted by physical sensor coverage and limited computational power. Meanwhile, IP networks like the Internet offer tremendous connectivity and computing resources. This paper presents Agimone, a middleware layer that integrates sensor and IP networks as a uniform platform for flexible application deployment. This layer allows applications to be deployed on the WSN in the form of mobile agents which can autonomously discover and migrate to other WSNs, using a common IP backbone as a bridge. It facilitates data sharing between WSNs and the IP network through remote tuple space operations, allowing sensors to easily defer expensive computations to more-powerful devices. We demonstrate the expressiveness of Agimone's programming model by examining a prototype cargo-tracking application that has been deployed using this system. We also provide an empirical evaluation of Agimone using a series of benchmarks deployed on two WSNs consisting of MICA2 sensor nodes connected by an IP network. These benchmarks show that inter-network tuple space operations take 10ms, and that one-way agent migrations between two different WSNs take approximately 83ms.

Keywords: Agimone, Agilla, Limone, integration, WSN, middleware, mobile agents

Technical Areas: Wireless Sensor Networks, Mobile Agent Middleware

I. Introduction

Wireless sensor networks (WSNs) consist of tiny sensors embedded within the environment. Example applications include habitat monitoring, microclimate research, surveillance, medical care, structural monitoring, and cargo tracking [10], [25], [16], [19], [15]. Many of these applications require that sensor nodes be deeply embedded in areas where they are difficult to physically access, such as scattered in forests or embedded in the sides of large cargo containers. In such a scenario, it is impractical to physically gather the nodes in order to collect data or deploy new applications.

This necessitates WSN systems where the nodes operate for very long periods of time without physical access. Data collection and application deployment is done over wireless networks. WSN systems must also be flexible enough to adapt to changing user requirements without completely reprogramming the sensors. However, typical WSN platforms often lack sufficient support for flexible application deployment. For example, the TinyOS [17] operating system hard-wires software components. Once deployed, application behavior can only be marginally tweaked by changing specific parameters defined prior to deployment.

To complicate matters, the power consumption of these sensors must be very low so that they can be deployed for months or even years without battery replacement. This requires that memory and other computational resources be scarce, and radio communication range and reliability be sacrificed [31]. These limitations impose severe restrictions on the complexity and scope of applications that can be realistically deployed on WSN systems.

Many of these restrictions can be eased by logically combining multiple, physically disconnected WSNs using a common IP network, such as the Internet. For example, WSNs can be used for cargo tracking and monitoring by attaching sensors

to individual cargo containers. However, these containers are frequently too far apart to be covered by a single WSN, since they are housed in separate warehouses and eventually relocated by boat or rail. Thus, the sensors form multiple independent WSNs which are unable to directly communicate with each other. The utility of the cargo tracking application would greatly increase if the user could issue a query — such as searching the containers for a specific item — simultaneously to all of these containers, even though their WSNs are not physically connected.

PCs with attached WSN gateways, or embedded devices like Stargate [1], can act as gateways between the IP network and their respective WSNs. By coordinating these disjoint networks to act as a single logical network, sophisticated WSN applications can be developed. Such applications are no longer restricted in scope to sensors within their own WSN: hundreds or thousands of nodes located in clusters around the world can collaborate autonomously on a single task.

However, communication and coordination between these networks is a complex undertaking, since WSNs are constantly being formed and reshaped as the application evolves. Nodes on the WSN must be able to determine the availability of other WSNs at run-time. Further, agents should be able to engage in transactions over the IP network without being affected by temporal disconnections and other short-term communication failures. For these WSN applications to be useful to clients on the IP network, application developers must be able to channel data between devices on the IP network and nodes in the WSNs in a simple and straightforward manner.

Middleware aims to meet these needs, providing high-level programming constructs that greatly simplify WSN application development and increase utility. To address the limitations of existing WSN middleware systems, we have developed a middleware called Agilla [12] for deployment on wireless sensors like the MICA2 platform. Limone [11], a lightweight middleware for communication and coordination over IP networks, provides a similar programming model and benefits to devices ranging from PDAs to desktop computers. Both middleware use a mobile agent-based paradigm, where programs are composed of agents that can migrate across nodes.

Though these middleware offer similar programming models, they partition the application into two sets of distinct, incompatible APIs and data structures. This discrepancy is not limited to these two particular middleware platforms. WSN operating systems like TinyOS offer such different APIs and capabilities from general-purpose operating systems like Windows and Linux, that the need for two incompatible development platforms is inevitable. Traditionally, developers have been forced to manually develop a translation layer for each application that crossed middleware boundaries, a tedious and error-prone procedure.

The main contribution of this paper is providing a general-purpose model which WSN devices can use to exploit the vast computational resources — including other WSNs — found in IP networks such as the Internet. We have developed Agimone, a thin and reusable integration layer between the Agilla and Limone middleware, which facilitates agent interactions that cross middleware boundaries. In Section II, we discuss the shortcomings of the current state-of-the-art, and explain the motivation behind creating a general-purpose integration layer. Section III provides a brief overview of the programming models used by Agilla and Limone. Section IV describes the architecture of our integration layer. Section V presents a cargo tracking application that highlights the capabilities and expressiveness of Agimone. A performance evaluation is provided in Section VI. We discuss related middleware systems in Section VII. Finally, we conclude in Section VIII.

II. Problem Statement

As the number and size of WSN deployments increase, so does the capacity for sophisticated WSN applications. This potential remains largely untapped due to the difficulty in distributing and coordinating applications across WSN boundaries. In this section, we discuss how this potential can be more-easily realized using a middleware system that integrates IP networks and WSNs.

A. Cargo Tracking: A Motivating Application

Consider the problem of cargo tracking. Cargo tracking is vital for national security and useful for shippers and their customers. 7 million cargo containers arrive annually into the United States, making it impossible to manually inspect every container. Instead, each shipping container can be equipped with a sensor, which will form a WSN with the other sensors and monitor the containers' contents. These sensors will need to be accessed by many different types of users — such as customs agent, shipping companies, and customers — who have different and evolving requirements. It is impossible to predict the needs of all these users ahead-of-time, so simple, inexpensive application re-deployment is paramount. Mobile agents are invaluable for this scenario. Each authorized user can deploy custom mobile agents to query the sensors on the containers. These agents can be deployed over-the-air inexpensively, and multiple overlapping applications can be deployed simultaneously by different parties.

However, complex applications cannot be built using only sensors and their wireless radios. The limited communication range of individual sensors forces WSN networks to form in physically-localized clusters. In many applications, it is unreasonable to expect the user to interact individually with each of these clusters. The current state-of-the-art is to logically connect these WSNs over an IP network.

For example, in the cargo tracking application described above, the nodes will form WSNs in localized clusters. Base stations in each cluster are connected together using a common IP network. Queries take the form of mobile agents that traverse each of these clusters. These queries should be able to migrate autonomously between the clusters using the IP network as a bridge. WSN nodes should also be able to easily share data, such as alert messages, with devices on the IP network.

B. Challenges

These application needs are essential, but at the same time difficult to satisfy. Since the devices that populate these two kinds of networks have such vastly different capabilities, it is impractical to deploy a uniform software layer across all devices. Today, this problem is largely tackled by partitioning the network into devices hosted on WSNs and the IP network, with separate software support platforms for each. Application-specific software is used to pass messages and translate queries between these two classes of devices.

However, using an application-specific translation layer is unwieldy at best. Writing such a support layer requires programming experience with both types of devices. Using an application-specific solution also often involves modifying and re-deploying this support layer whenever the application's features or protocols change. This is unacceptable for applications which have a constantly-evolving set of capabilities, like cargo tracking.

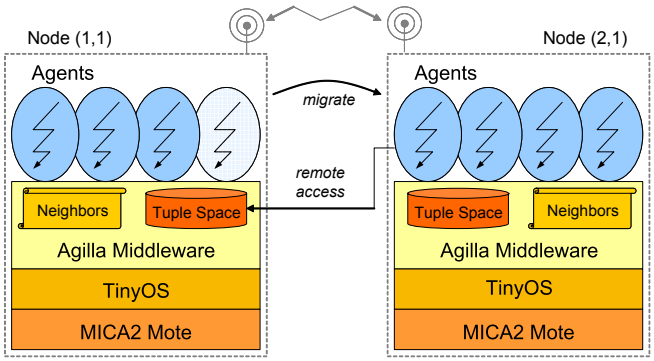

Fig. 1. The Agilla Middleware Architecture

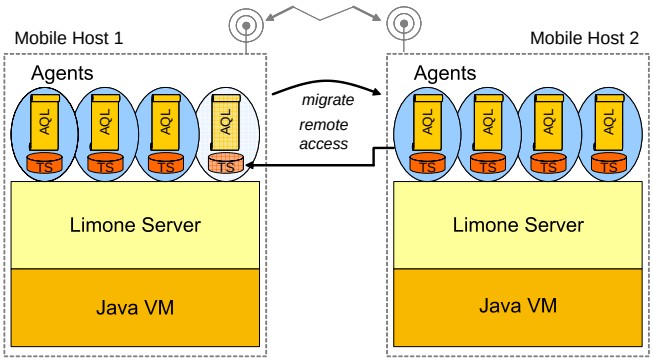

Fig. 2. The Limone Middleware Architecture

In this work, we aim to develop a middleware platform that supports seamless integration of WSNs and IP networks into a uniform software platform. Our middleware provides several services that facilitate the development of WSN applications which exploit the IP network as a resource for computation and communication. Mobile agents within a WSN are provided with a list of all other WSNs attached to the same IP network. Agents can autonomously migrate over the IP network to any of the WSNs advertised in this list. Finally, we provide a common data space where devices on the IP network and WSNs can share messages and data. These services offer application developers a straightforward yet powerful programming model for implementing complex WSN applications, like the cargo tracking given above.

III. Background

This section provides a brief overview of the programming models offered by Agilla and Limone. More details on the implementation are available in [12] and [11].

A. Agilla

Agilla programs consist of mobile agents that coordinate through tuple spaces. Agilla's architecture is shown in Figure 1. Each agent is hosted on a virtual machine with dedicated instruction and data memory. As an agent executes, it may execute special instructions that allow it to interact with the environment and move across nodes. Multiple agents can coexist on a single node. Agilla provides agents with local data storage in the form of a heap and operand stack. Agilla agents use a stack-based architecture and are programmed in a bytecode language based on that of Maté [21], but tailored to the mobile agent paradigm. Like Maté, most Agilla instructions fit in a single byte. Agilla is available for the Mica2, MicaZ, Tyndall25 nodes and is distributed through TinyOS' CVS respository [2] within the module `tinyos-1.x/contrib/wustl`. See Agilla's website [3] for more details.

Agilla's tuple spaces offer a lightweight shared data space where the datum is a tuple that is accessed via pattern matching. This allows one agent to insert a tuple containing data (such as a sensor reading) and another to later retrieve it without the two knowing each other, thus achieving a high level of decoupling. Unlike messages passed over sockets, tuples placed in a tuple space survive temporal disconnections, which frequently occur due to node mobility or unreliable links.

```
1:  pushn mrk       // string "mrk"
2:  pushcl 15       // integer 15
3:  pushc 2         // length of tuple (2 fields)
4:  out             // out(<15, "mrk">)
```

Fig. 3. Agilla **out** Code Snippet

```
1:  pushn mrk       // string "mrk"
2:  pusht VALUE     // type VALUE (integer)
3:  pushc 2         // length of template (2 fields)
4:  rd              // rd(<VALUE, "mrk">)
```

Fig. 4. Agilla **rd** Code Snippet

Operation	Parameters	Return Value	Description
`out`	tuple	none	place the tuple in the local tuple space
`in`	template	tuple	remove a matching tuple from the local tuple space, or wait until one exists (blocking)
`rd`	template	tuple	copy a matching tuple from the local tuple space, or wait until one exists (blocking)
`inp`	template	tuple or **null**	search for a matching tuple in the local tuple space, remove it if found (non-blocking)
`rdp`	template	tuple or **null**	search for a matching tuple in the local tuple space, copy it if found (non-blocking)
`rout`	destination, tuple	none	perform an `out` operation on the destination node's tuple space
`rinp`	destination, template	tuple or **null**	perform an `inp` operation on the destination node's tuple space
`rrdp`	destination, template	tuple or **null**	perform a `rdp` operation on the destination node's tuple space

Fig. 5. Agilla Tuple Space Operations

Each sensor in the WSN has a single local tuple space. Tuple spaces offer many of the same programming benefits as shared data systems, but with far less message-passing required at run-time. Data is stored in the form of fields; tuples containing one or more fields can be added to the tuple space using the **out** primitive. In Agilla, tuple fields contain 16 bits of data, using a handful of well-known data types (integer, string, sensor reading, etc.). <15, "mrk"> is an example of a such a tuple; it contains the integer 15 in the first field, and the string "mrk" in the second. Figure 3 gives a snippet of Agilla code that places this tuple in the sensor's tuple space. (Note that, due to Agilla's stack-based architecture, the fields are pushed onto the stack in reverse order.)

rd and **in** operations respectively remove and copy tuples from a tuple space. These operations are parameterized by patterns that specify forms of tuples that match, or *templates*. In Agilla, templates take the same form as tuples. However, agents may fill a template's field with a type (e.g., `VALUE` or `STRING`) rather than a specific value. This indicates that any value is acceptable, as long as the field's data type is correct. <`VALUE`, "mrk"> is an example of a template that matches the tuple <15, "mrk">. Agilla compares fields pair-wise according to the order that they are added to the tuple. Thus, the template <"mrk", `VALUE`> does not match the tuple <15, "mrk">. The code snippet in Figure 4 gives an example of a **rd** operation that copies a tuple matching the template <`VALUE`, "mrk"> from the sensor's tuple space.

If a matching tuple is not available when **rd** or **in** are executed, then the operations will block until one is placed in the tuple space. Since this behavior is not always desirable, Agilla offers agents additional tuple-space primitives which perform "probing" (i.e., non-blocking) tuple removals and copies. To allow agent interactions that span sensors, Agilla also provides tuple space operations which manipulate tuple spaces residing on remote sensors. These operations are summarized in Figure

Operation	Parameter	Description
`smove`	destination	strong move
`wmove`	destination	weak move
`sclone`	destination	strong clone
`wclone`	destination	weak clone

Fig. 6. Agilla Agent Migration Operations

```
ETuple tuple = new ETuple();
tuple.addField(new EField(“ID”, 15));
// Field <ID: 15>
tuple.addField(new EField(‘Flag”, “mark”));
// Field <Flag: “mark”>
getTS().out(tuple);
// out(<ID: 15, Flag: “mark”>
```

Fig. 7. Limone **out** Code Snippet

```
ETemplate template = new ETemplate();
template.addConstraint(new EConstraint(“ID”, Integer.class,
        new DefaultConstraintFunction()));
// Match field ID containing any Integer
template.addConstraint(new EConstraint(“Flag”, String.class,
        new EquivalencyConstraintFunction(“mark”)));
// Match field flag containig exactly “mark”
ETuple tuple = getTS().rd(template);
// rd a tuple matching the above template
```

Fig. 8. Limone **rd** Code Snippet

5. Finally, Agilla offers a *reaction* mechanism, where a piece of code is executed when a specified type of tuple is placed in the local tuple space.

All tuple space operations occur atomically. It is not possible for an agent to see a partially inserted tuple, or for a single tuple to be partially removed or removed more than once. Since tuple spaces are local, atomicity is enforced by serializing the operations. If the operation blocks, its execution is delayed until after the matching tuple arrives in the tuple space. If there are multiple operations and reactions waiting for the same tuple, all reactions are notified, but the blocked operations are executed sequentially until all pending operations are executed, or the tuple is removed by an **in** or **inp**.

Agilla agents may move or clone onto other hosts in the WSN using either *weak* or *strong* migration operations as shown in Figure 6. Weak migrations include only the agent’s code, so any computations must restart from the beginning on the new host. Strong migrations include computational state as well as code, so computations can resume after the agent is migrated. Because Agilla agents run on top of a virtual machine, agents can migrate between devices of different hardware architectures, provided that the radios are compatible.

B. Limone

Limone provides a similar agent-based programming model using tuple spaces for inter-agent communication. Its architecture is shown in Figure 2. Limone provides the same primitive local and remote tuple space operations as Agilla, as well as a reaction mechanism analogous to the one in Agilla. However, each Limone agent has its own dedicated tuple space, whereas (due to memory limitations) all Agilla agents on a single host share one tuple space. Limone also provides a pluggable device discovery mechanism, where each agent-specified *profile* is automatically propagated to other interested agents as new agents enter or leave the network.

Limone’s tuple contents do not suffer from many of the restrictions imposed their Agilla counterparts. Fields in Limone tuples are indexed by a user-specified name rather than by numerical order. These fields can contain any Java data type of any size. An example of a Limone tuple is <ID: 15, Flag: “mark”>; this tuple contains a field named “ID” with the Integer value 15, and another field named “Flag” with the String value “mark”. The code fragment in Figure 7 demonstrates how to place this tuple in the agent’s tuple space.

Similarly, Limone templates are more flexible than Agilla templates. Limone templates consist of a series of *constraints*. These constraints describe the names of fields; the types of values they should contain; and a *constraint function*. These functions provide more fine-grained control over pattern matching than is possible in Agilla. For example, the constraint <“ID”, Integer, GreaterThanConstraint(10)> matches fields named “ID” that contain an Integer greater than 10. Most constraints use

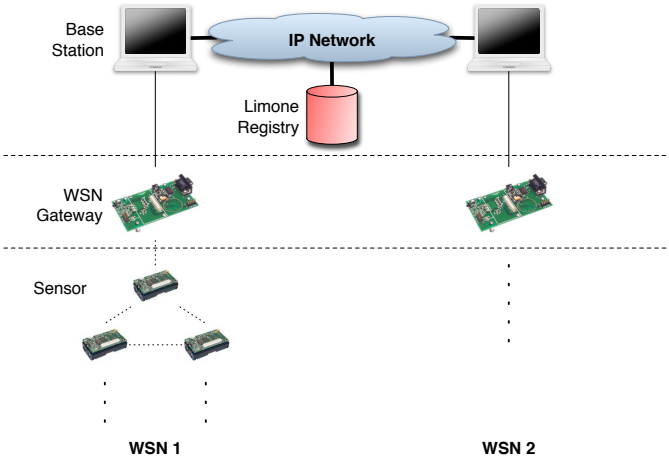

Fig. 9. Agimone Network Architecture

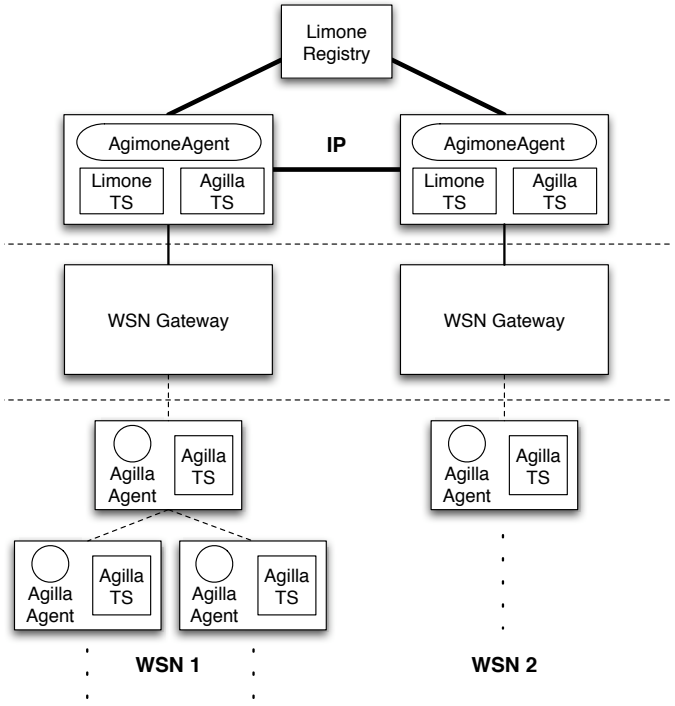

Fig. 10. Agimone System Components

either DefaultConstraintFunction (match any value, as long as the type is correct), or EquivalencyConstraintFunction (match only if the field contains the specified value). The template << ID, Integer, DefaultConstraintFunction()>, <Flag, String, EquivalencyConstraintFunction("mark")>> is an example of a template that matches the tuple <ID: 15, Flag: "mark">. In Figure 8, we see Limone code that attempts to copy a tuple matching this template out of the agent's tuple space.

IV. Architecture of Integration Layer

We have constructed the Agimone architecture (shown in Figure 9) which integrates the Agilla and Limone middleware platforms. Each WSN is associated with a base station such as a laptop or a Stargate. The WSNs are populated with Agilla agents which perform computations and collect sensor data. Inter-agent communication is facilitated by Agilla tuple spaces. Each node in the WSN hosts one Agilla tuple space, and up to three Agilla agents.

The IP network and WSNs are spanned by WSN gateways attached to these base stations: sensors can communicate with a nearby gateway wirelessly, while the base stations communicate with their attached gateways using a wired interface (e.g., UART or USB). The base stations communicate with each other over the IP network using Limone. Communication in Limone is performed using tuple spaces; each Limone agent is provided with its own Limone tuple space.

WSNs may discover each other using a beaconing scheme where multicast routing is supported, or a centralized service directory elsewhere. We have implemented a simple prototype Limone service registry that is suitable for a small number of agents. However, it is not designed to scale for deployment on larger networks like the Internet. Since Limone's discovery mechanism is pluggable, applications that require a greater degree of scalability can replace this registry with a more sophisticated protocol, like WSDL [30] or Bonjour [8].

```
1:    pusht STRING                    // type STRING
2:    pushc 1                         // length of template (1 field)
3:    pushloc UART_X UART_Y           // base station's location
4:    rrdp                            // rrdp(base station, <STRING>)
```

Fig. 11. WSN Discovery Code Snippet

Agimone is populated with the following components, as shown in Figure 10:

- The `AgimoneAgent`s are specific Limone agents which allow Agilla tuples and agents to traverse the IP network. These agents serve as the basis for the Agimone integration layer. Each base station hosts one `AgimoneAgent`.
- The Agilla and Limone tuple spaces, as described above.
- The Limone registry allows remote WSN discovery. Each application shares a single Limone registry. This registry is optional on IP networks where multicast broadcasts are supported.

In the remainder of this section, we will describe the services Agimone provides in further detail.

A. WSN Discovery

Since new WSNs are being formed and destroyed as the applications evolve, it is often necessary for agents in the WSNs to be aware of these changes at run-time. This is accomplished using a *WSN advertisement* scheme. Each base station's `AgimoneAgent` encapsulates information about the corresponding WSN in a WSN advertisement message. This WSN advertisement describes the WSN's properties to Agilla agents. Since different applications may be interested in different properties of the WSNs, this advertisement is application-specific. For example, agents that comprise a cargo tracking application may only be interested in knowing the location of each network. Thus, the WSN advertisements contain a 3-character string describing their locations, such as "dok" (dock) or "shp" (ship).

When a new WSN connects to the IP network, its corresponding `AgimoneAgent` begins beaconing a well-known Limone registry with messages containing this WSN advertisement. The Limone registry in turn forwards these advertisements to other hosts on the Limone network. Similarly, the Limone registry notifies Limone agents when hosts leave the Limone network. `AgimoneAgent`s use these notifications to store up-to-date copies of all other WSN advertisements in their base station's Agilla tuple space.

Agilla agents can access the base station's tuple space by performing remote tuple space operations with the special destination address (UART_X, UART_Y). Thus, they can select an appropriate WSN advertisement using a **rrdp** operation. The example in Figure 11 shows such a **rrdp** operation, which tries to locate any WSN advertisement containing just a string. If this operation succeeds, then a tuple containing a matching WSN advertisement is placed on the top of the Agilla agent's operand stack.

B. Migration Across WSNs

Once an Agilla agent has a WSN advertisement on the top of its operand stack, it can proceed to migrate to the WSN with the assistance of the `AgimoneAgent`. The agent migration procedure is detailed in Figure 12. WSN advertisements are distributed in Steps 1 and 2, and placed in the base stations' tuple space in Step 3. The Agilla agent selects one of these WSN advertisements in Step 4.

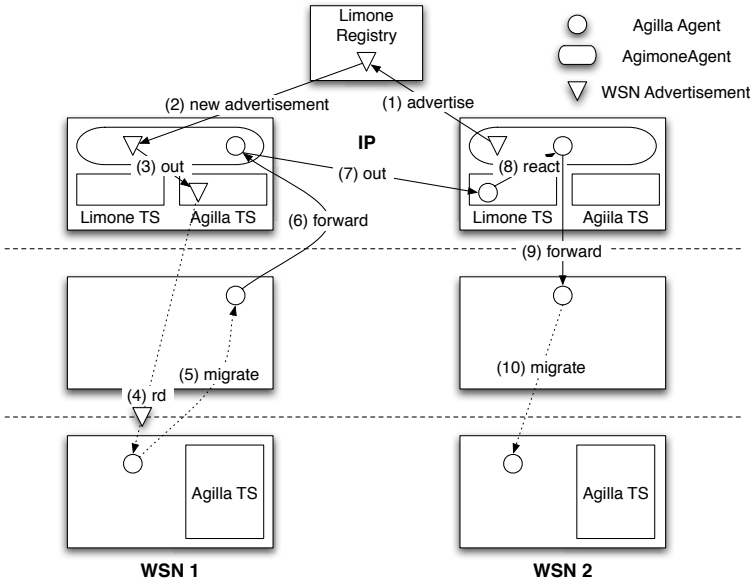

Fig. 12. Agilla Agent Migration Across Different WSNs

```
1:              pusht STRING                    // type STRING
2:              pushc 1                         // length of template (1 field)
3:              pushloc UART_X UART_Y           // base station's location
4:              rrdp                            // rrdp(base station, <STRING>)
5:              rjumpc ADFOUND
6:              halt                            // if matching tuple not found, halt
7: ADFOUND      pushloc UART_X, UART_Y          // base station's location
8:              smove                           // strong migrate the agent to base station
```

Fig. 13. Migration Code Snippet

Once an Agilla agent has selected a satisfactory WSN advertisement, it performs a strong migration to the WSN gateway, as shown in Step 5. Sample code to perform this operation is listed in Figure 13. This migration request is forwarded to the `AgimoneAgent` executing on the base station in Step 6. The `AgimoneAgent` extracts the destination WSN advertisement from the top of the agent's operand stack. It then encapsulates the Agilla agent into a Limone tuple of the form <Agent: *(encapsulated agent)*>. In Step 7, it places this tuple into the tuple space of the `AgimoneAgent` residing in the destination network.

When the `AgimoneAgent` initializes on the base station, it installs a reaction on its tuple space that notifies it of new tuples in the form <Agent: Agilla Agent>. Thus, in Step 8, the `AgimoneAgent` on the base station of the destination WSN is notified of the arrival of the encapsulated Agilla agent. It extracts the Agilla agent from its tuple space and injects it into the WSN gateway in Step 9. In Step 10, the agent migrates into the new WSN, where it resumes its computation.

C. Cross-Middleware Interactions Via Tuple Spaces

Because of the limited computational powers of wireless sensors, Agilla agents may wish to use devices on the IP network as a computational resource. By the same token, a Limone agent may wish to exploit the sensing resources of a remote WSN. Both goals can be achieved by providing Limone agents with access to the Agilla tuple space that resides on each base station. This gives both types of agents a common data space for exchanging messages. However, directly exposing the Agilla tuple

space API to Limone agents has some undesirable side effects. For example, Limone agents would only be able to make their Agilla tuple spaces accessible to the WSN if they had some form of direct communication with the sensors. If a Limone agent resided on a host outside of the communication range of the desired WSN, or if the host were simply not equipped with a WSN gateway, then it could not interact with the Agilla agents in that WSN.

Instead, the `AgimoneAgent` exposes each base station's Agilla tuple space to the Limone network by wrapping it in the Limone tuple space API. Other Limone agents communicate with Agilla agents by performing remote tuple-space operations on `AgimoneAgent`'s Limone tuple space. The `AgimoneAgent` captures the Limone tuple space API calls and translates them to their Agilla equivalents, which it forwards to the Agilla tuple space API. Hence, when a Limone agent places tuples in the `AgimoneAgent`'s tuple space, these tuples are made available to any Agilla agent in the corresponding WSN. Similarly, tuples placed into the base station's tuple space by Agilla agents are made available to remote Limone agents. These Limone agents need not have a WSN gateway attached to their host, since the `AgimoneAgent` will interact with the WSN on their behalf.

However, translating Limone tuple space operations to their Agilla counterparts is not always straightforward. As discussed earlier in Section III, there are restrictions placed on the contents of Agilla tuples and templates that are not present in Limone. For example, a Limone agent may attempt to place the tuple <ID: 3.14, Flag: "mark"> in the tuple space. Since Agilla does not have a floating-point data type, there is no way to convert this Limone operation into an Agilla tuple space operation. Clearly, there does not always exist a 1-to-1 mapping from Limone tuples and templates to their Agilla equivalents, due to the disparities between nodes in the WSN and those on the IP network.

To resolve this problem, the `AgimoneAgent`'s tuple space uses Limone's rejection mechanism to filter incoming tuple space operations. This mechanism allows agents to reject any operations issued on their tuple space from remote agents. The `AgimoneAgent` places the following restrictions on all incoming tuples and templates:

- Fields cannot be named arbitrarily. Instead, the field names must impose a numerical order on the fields, as required by Agilla. That is, exactly one field must be named "1", exactly one field must be named "2", etc.
- Fields may only use the data types recognized by Agilla.
- The only constraint functions which are allowed are `DefaultConstraintFunction` (i.e., match by type) or `EquivalencyConstraintFunction` (i.e., match exactly by value).

If a Limone tuple space operation contains a tuple or template that does not conform to these specifications, it cannot be converted to an Agilla tuple space operation. Any such tuple space operations will be rejected by the `AgimoneAgent`. Conforming operations are converted to their Agilla counterparts and forwarded to the Agilla tuple space. The results of the operation are converted from Agilla tuples to Limone tuples (using the naming and data type conventions specified above) and sent back to the originator of the request.

As discussed earlier, the `AgimoneAgent`'s tuple space is also used to migrate encapsulated agents. The `AgimoneAgent` includes a special case to accept tuples containing these agents. However, since these tuples do not conform to the restrictions listed above, they are immediately discarded after the enclosed agent is injected into the network.

D. Implementation Details

Agilla and Limone have been implemented and deployed on a wide variety of hardware. Agilla consists of two parts: a NesC-based portion that is installed on sensors, and a Java-based `AgentInjector` that is installed on base stations. Since RAM and ROM are at a premium on many sensor devices, Agilla is necessarily compact: it consumes 49.66KB of flash ROM and 3.07KB of RAM. Agilla has been ported to several different sensor architectures, including MICA2, MICAZ, and Tyndall25. For this paper, we used a CVS snapshot of Agilla 3.0, which can be downloaded from [4].

The Limone and Agimone packages are developed in Java according to the Java 2 Micro Edition Personal Profile 1.0 [29] specification. This allows Limone and Agimone to be deployed on devices like PDAs and Stargates which cannot host full Java 2 Standard Edition runtimes. Since Personal Profile is a subset of the Java 2 Standard Edition specification, Limone and Agimone can also be deployed on desktop and laptop computers. Limone was designed for deployment on storage-constrained devices like PDAs; hence, the .JAR file containing its Java bytecode consumes only 132KB of storage space. Agimone is even more compact: its .JAR file consumes less than 13KB of storage space. Since Agimone is implemented as a Limone agent, it operates on any platform supported by Limone. We have successfully deployed Limone to many different operating systems, including Windows Mobile, Windows XP, Linux, Solaris, and Mac OS X.

V. Case Study: Cargo Tracking

Using the architecture described in the previous section, we can implement a wide range of complex WSN applications. Cargo tracking is an example of an application that is well-suited for implementation using Agimone. As discussed in Section II, cargo containers can be equipped with sensors that form WSNs in localized clusters. Many of these containers are located in remote warehouses and vehicles. Thus, it is paramount that users be able to interact with these WSN clusters without needing to be within the WSN's communication range. This can be achieved by connecting the WSNs' base stations together using a common IP network, then deploying Agimone to these base stations so that queries may traverse either network as needed.

In this section, we present a prototype application that uses mobile agents to track cargo. This application was previously implemented using Limone and Agilla [15], and deployed on a testbed of 12 MICA2 motes. In the first version of this program, the two middleware were poorly-integrated. Queries and responses were passed across middleware boundaries using a custom Limone agent developed specifically for this application. This agent had to be modified and re-deployed each time a new type of query was added to the application, adding significant overhead to the development process. For this paper, we have augmented parts of the cargo tracking application to take advantage of Igillimone's capabilities. The modified agent code, which we will present in this section, is much cleaner as a result.

A. Watchdog Agents

Sensors attached to shipping containers can be equipped with various inexpensive sensor boards. These boards can be monitored for unusual readings to try to detect attempted intrusions into the containers. For example, a user could "arm" a container by deploying a watchdog agent that collects light data from a sensor installed on the inside of a cargo container. If the container is opened, then the sensor sends an alert tuple to the nearest base station, which in turn propagates the tuple to

```
1:                pushloc [destination]
2:                smove                  // move from base station to destination sensor
3: LOOP           pushc PHOTO
4:                sense
5:                pushcl 850
6:                cgt
7:                rjumpc OPENED          // if light sensor reading > 850, goto OPENED
8:                rjump LOOP             // else goto LOOP
9: OPENED         pushc 17
10:               pushc 1
11:               out                    // out(<"al1">)
12:               addr
13:               pushn al1
14:               pushc 2
15:               pushloc UART_X UART_Y
16:               rout                   // rout(UART, <"al1", address>)
17:               halt
```

Fig. 14. Light Sensor Watchdog Code

```
public void agentInit() {
    ETemplate alertTemplate = new ETemplate();
    alertTemplate.addConstraint(new EConstraint("1", AgillaString.class,
        new DefaultConstraintFunction()));
    alertTemplate.addConstraint(new EConstraint("2", AgillaValue.class,
        new DefaultConstraintFunction()));
    // Make a template for tuples with a field "1" containing an Agilla STRING
    // and a field "2" containing an Agilla VALUE

    Reaction resultsReaction = new Reaction(new ReactivePattern(new ProfileSelector(),
        Reaction.ONCE_PER_TUPLE, alertTemplate), this);
    getRR().registerReaction(resultsReaction);
    // Install a reaction that notifies us when a matching tuple is produced
}

public void reactsTo(ReactionEvent re) {
    ETuple t = re.tuple();
    AgillaString alertType = (AgillaString)t.getField("1").getValue();
    AgillaValue alertSource = (AgillaValue)t.getField("2").getValue();
    // Extract the contents of the tuple from the event

    ...
}
```

Fig. 15. Client-Side Alert Reaction Code

security personnel over the IP network. Light sensor readings are only an example of the kinds of environmental readings that could be used: many other kinds of intrusions can be detected, limited only by the kinds of sensor boards that are commercially available.

We have implemented two prototype agents that monitor the sensor's accelerometer and light readings, respectively. In the interest of space, we will only examine the latter agent here. Figure 14 provides the code for the light-monitoring agent. Its implementation is fairly straightforward, owing to the features offered by Agimone discussed in Section IV. The agent first moves to the sensor that it intends to monitor. It loops, repeatedly reading the light sensor until its reading goes above a fixed threshold. When this happens, it places a tuple in its own tuple space that records the event. It also places an alert tuple in the base station's tuple space.

Because Agilla tuple space on the base station of each WSN is exposed to the IP network as a Limone tuple space, it is easy for clients on the IP network to gain access to the alerts that the sensors create. Figure 15 gives an example of how a remote client on the IP network can react to the creation of these tuples. In the client's `agentInit()` method, we create a Limone template that describes the alert tuple's format: a string in the first field that contains the alert type, and an integer value in the second field that contains the sensor's address. The client registers this reaction with the Limone middleware. Limone

automatically forwards this reaction registration to all other Limone agents on the IP network, including the `AgimoneAgent` that “owns” the Agilla tuple space on each base station.

When the Agilla agent places an alert tuple in the base station’s tuple space, the reaction fires. Limone invokes the client Limone agent’s `reactsTo()` method when this happens. In this method, we extract the two fields from the tuple that created the event. We can then do whatever kind of processing we desire with this information (e.g., display it in a GUI, log it to disk, and notify security personnel).

The functionality provided by Agimone reduces the effort needed to develop this application to a minimum. The Agilla agent contains only 17 lines of code, and yet is able to interface with the sensor’s low-level sensing and communication hardware. Further, the Limone client requires only 7 lines of code to automatically receive notification messages sent by the sensors, and another 4 lines of code to extract its contents. As a result, this agent was developed in only a few hours. A comparable application written using NesC would likely require much more code, as well as custom Java or C code to interface with the base station over the IP network. An application of this size would likely also need significant debugging after development. In contrast, Agimone provides applications with a reusable and robust middleware platform that exhibits a high degree of network integration.

B. Intrusion Search Agent

The watchdog agents keep a record of all anomalous behavior in the sensors’ tuple spaces. A user, such as a shipping company or a port authority, may want to search all the containers for possible tampering. Consider a scenario where containers are being moved between a ship and a loading dock, each of which has a corresponding base station. These locations are out of WSN communication range with each other, so the boxes in each location form a separate WSN. The WSNs are bridged by an long-range IP network, such as 802.11b or Ethernet, between their respective base stations.

Though users can search both WSNs for intrusions simultaneously, a comprehensive search may be unnecessarily expensive. Ideally, the scope of such a search should be determined at runtime. For example, assume that the containers on the ship are far more likely to be tampered with than the containers on the dock. Rather than searching both WSNs separately, it is better to only search the containers on the ship first. If one of the containers on the ship has been tampered with, then the search should automatically expand to the containers on the dock, in order to determine the scope of the security breach.

This behavior can be achieved by deploying a mobile agent that spreads across a single WSN, looking in each sensor’s tuple space for a recorded anomaly, and notifying the user as records are found. If one is found, then the agent migrates to the other WSN. Once there, it repeats the search on the second WSN.

The code fragment shown in Figure 16 attempts to migrate the agent between WSNs. Again, this code is fairly straightforward, building on the simple examples given in Section IV. Our base stations’ advertisement tuples contain a single field with the string “dok” or “shp”. Accordingly, our agent performs a **rrdp** operation on the base station’s tuple space, using a template that matches tuples containing a single string. If a matching tuple is found, then Agilla places the tuple on the top of the agent’s operand stack. The agent then attempts to issue an **smove** request to migrate to the base station. As described in Section IV-B, the agent will migrate over the IP network to the WSN corresponding to the advertisement tuple at the top of its operand stack.

```
1:                pusht STRING
2:                pushc 1
3:                pushloc UART_X UART_Y
4:                rrdp        // rrdp(UART, <string>)
5:                rjumpc ADFOUND
6:                halt        // if matching tuple not found, halt
7: ADFOUND        pushloc UART_X UART_Y
8:                smove       // strong move to UART
9:                rjumpc MIGRATEOK
10:               halt        // if migration failed, halt
11: MIGRATEOK     pushcl START
12:               jumps       // goto START
```

Fig. 16. Intrusion Search Migration Code Snippet

```
1: START      pushn mrk
2:            pushcl [search id]
3:            pushc 2
4:            rdp         // rdp(<search ID, "mrk">)
5:            rjumpc FOUND
6:            rjump NOMARK
7: FOUND      halt
8: NOMARK     pushn mrk
9:            pushcl [search id]
10:           pushc 2
11:           out         // out(<search ID, "mrk">)
```

Fig. 17. Intrusion Search Preamble

If the migration is successful, then the agent will jump back to the start of its execution and search the destination network for intrusions.

To prevent agents from traversing the same WSN twice, the agent's code begins with the snippet listed in Figure 17. Each query is given a unique 16-bit numeric ID. When the agent arrives in a WSN, it attempts to read a "marker" tuple containing this ID from the base station's tuple space. If it succeeds, then it knows that the network has already been traversed, and it dies. Otherwise, it places the marker in the base station's tuple space to prevent any further copies from traversing the WSN. Since each marker includes the query's unique instance ID, later queries will not be stopped by the markers left by a previous query.

Like in the last example, the client can use Limone's reaction mechanism to automatically aggregate the results sent from the WSNs. A single reaction, like the one in Figure 15, will collect all matching tuples that are placed in any base station's tuple space, regardless of which WSN produced the tuple.

Again, Agimone's functionality and reusability greatly simplify the development of this application. Agilla agents can migrate between WSNs using only 23 lines of Agilla code. No additional support code had to be written or installed on the base station to handle these migrations.

VI. Performance Evaluation

We evaluated our system by deploying it on two WSNs connected by an IP network. The WSNs are composed of MICA2 motes and are separated by using different radio channels. Each WSN has a single gateway that is attached to an IBM R40 laptop via a 115.2Kbps serial link. The laptops are connected via a standard 100Mbps wired Ethernet network. Since they are on the same subnet, a Limone registry is not required; they discover each other using regular multicast beacons. The laptops are identically configured with a 1.5GHz Intel Pentium M processor, 512MB of RAM, Windows XP and Java Standard Edition 5.0. All latencies are measured using Java's `System.nanoTime()` method which uses the system's most accurate timer. This section presents micro-benchmarks examining the primitives that cross network boundaries. These benchmarks can be divided into three categories: tuple space operations, agent migration operations, and overall performance.

A. Tuple Space Operations

In these benchmarks, we measure the cost of tuple space operations which cross middleware boundaries. Specifically, we evaluate operations **rinp**, **rrdp**, and **rout**. These operations can be executed in both directions; they may be performed by the

```
1: BEGIN      pushcl 100
2: START      copy
3:            pushc 0
4:            ceq
5:            pushc END
6:            jumpc // jump to END after doing n rrdp(uart)
7:            pusht string
8:            pushc 1
9:            pushcl uart
10:           rrdp // do a rrdp(uart)
11:           rjumpc FOUND
12:           pushc 25
13:           putled // toggle red LED if rrdp fails
14:           rjump CONTINUE
15: FOUND     pop
16:           pop // remove tuple
17: CONTINUE  dec
18:           pushc START
19:           jumps // jump back to START
20: END       pushcl uart
21:           smove // migrate to the base station
22:           halt
```

Fig. 18. The **rrdp** Benchmark Agent.

```
1: BEGIN      pushcl 100
2: START      copy
3:            pushc 0
4:            ceq
5:            pushc END
6:            jumpc // jump to END after doing n rrdp(uart)
7:            pusht string
8:            pushc 1
9:            pushc 1
10:           cpull // set condition = 1
11:           rjumpc FOUND
12:           pushc 25
13:           putled // toggle red LED if rrdp fails
14:           rjump CONTINUE
15: FOUND     pop
16:           pop // remove tuple
17: CONTINUE  dec
18:           pushc START
19:           jumps // jump back to START
20: END       pushcl uart
21:           smove // migrate to the base station
22:           halt
```

Fig. 19. The **rrdp** Baseline Agent.

Operation (Mote-to-PC)	latency (ms)
rinp	10.64 ± 0.15
rrdp	10.35 ± 0.06
rout	10.37 ± 0.07

Operation (PC-to-Mote)	latency (ms)
rinp	10.98 ± 0.17
rrdp	11.26 ± 0.19
rout	10.85 ± 0.07

Fig. 20. The Latency of Remote Tuple Space Operations

AgimoneAgent on the tuple space belonging to the WSN gateway (PC-to-Mote), or by an Agilla agent on the base station's tuple space (Mote-to-PC).

Mote-To-PC. The first set of benchmarks determine the latency of the remote tuple space operations when the initiator is an Agilla agent residing on the WSN gateway and the destination is the attached base station. Since modifying the implementation of Agilla may affect the results, we inferred the latency by injecting two agents, a benchmark and baseline, onto the gateway. The benchmark agent performs a tuple space operation 100 times before migrating back to the base station. The baseline agent is identical to the first, except it performs a dummy operation in place of the tuple space operation. This allows us to isolate the cost of the tuple space operation from the rest of the overhead associated with executing the agent. For example, the agents used to evaluate **rrdp** are shown in Figures 18 and 19. Notice that on line 10, the benchmark agent performs **rrdp** whereas the baseline agent only performs **cpull**, which is a local operation whose overhead is negligible, since it is several orders of magnitude lower [12]. Thus, by comparing the lifespan of the two agents, the latency of **rrdp** can be calculated.

Three sets of agents were produced, one for each operation. Each benchmark of 100 operations is repeated 10 times, and the average latency is calculated. The results are shown in Figure 20. They show that all operations consume approximately 10 to 11 ms on average, with the **rinp** operation being slightly more expensive than the other two.

PC-To-Mote. The second set of benchmarks determine the latency of remote tuple space operations when the initiator is the AgimoneAgent on the base station, and the destination is the gateway. Since these operations start and end on the same device, their latency can be directly measured. The base station's tuple space is initialized by populating it with tuples for the **rinp** and **rrdp** operations to consume. For each instruction, we measured the time it takes to execute them 20 times, and

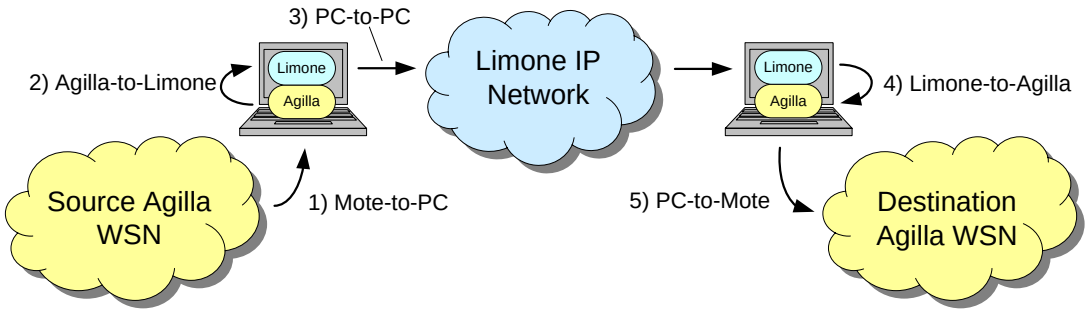

Fig. 21. The Five Stages of an Inter-WSN Agent Migration Operation.

divided the result by 20 to get the average. This process is repeated 20 times; the results are shown in Figure 20. The cost of these operations is roughly the same, regardless of direction.

B. Agent Migration Operations

Agent migration operations enable agents located in one WSN to migrate across an IP network into another WSN. These operations include **smove** (strong move) and **sclone** (strong clone). They are identical in that both operations capture the agent's state and code and transfer them to the destination WSN. They differ in that **smove** terminates the agent on the source node, whereas **sclone** does not. Since both operations are identical in terms of performance, only **smove** is considered in the benchmarks that follow.

From an Agilla agent's perspective, an inter-WSN agent migration occurs in an atomic step, e.g., by executing instruction **smove**. However, as discussed in Section IV, there are in reality many steps involved. We group these steps into the five distinct stages shown in Figure 21. During stage one (Mote-To-PC), the migrating agent moves from the source mote onto the attached base station. During stage two (Agilla-to-Limone), the agent is encapsulated within a Java object and passed to the `AgimoneAgent`. This is the point where the agent passes from the Agilla realm into the Limone realm. During stage three (PC-to-PC), the agent is migrated from the `AgimoneAgent` at the source base station to the tuple space located at the destination. During stage four (Limone-to-Agilla), the destination's `AgimoneAgent` reacts to the agent tuple, extracts the migrating agent, and passes the agent to Agilla's `AgentInjector`. This is when the agent passes from the Limone realm back into the Agilla realm. Finally, during stage five (PC-to-Mote), the agent is injected into the destination WSN. Each of these stages is considered separately in the benchmarks that follow.

Unless otherwise stated, the benchmarks use the bouncing agent shown in Figure 22. The bouncing agent consists of 16 bytes of state, and 20 bytes of code for a total of 36 bytes, or two TinyOS messages. When it arrives on a mote, it queries the WSN gateway for an advertisement tuple consisting of a string (lines 1-4). If it finds an advertisement, it migrates to the WSN being advertised (lines 13-16). If it fails to find such a tuple, it toggles the mote's red LED, sleeps for a half second, and repeats (lines 7-12). Note that since the `AgimoneAgent` does not include the local WSN's advertisement in the base station's tuple space, the bouncing agent is guaranteed to only discover a remote WSN's advertisement tuple, and will thus

```
1: BEGIN   pusht string
2:         pushc 1
3:         pushcl uart
4:         rrdp        // search for advertisement tuple
5:         pushc FOUND
6:         jumpc       // jump to FOUND if found
7:         pushc 25
8:         putled      // blink red LED if not found
9:         pushc 4
10:        sleep
11:        pushc BEGIN
12:        jumps
13: FOUND  pushcl uart
14:        smove       // migrate to remote WSN
15:        pushc BEGIN
16:        jumps       // after moving, repeat
```

Fig. 22. The Bouncing Agent Use for Evaluating Inter-WSN Migration Performance

Stage	Name	Unfiltered Latency	Filtered Latency	# Points Filtered
1	Mote-to-PC	36.12 ± 1.19ms	35.56 ± 0.58ms	2
2	Agilla-to-Limone	1.03 ± 0.16ms	$307.11 \pm 1.59\mu$s	182
3	PC-to-PC	19.45 ± 0.26ms	19.11 ± 0.15ms	19
4	Limone-to-Agilla	1.13 ± 0.16ms	$830.05 \pm 2.26\mu s$	12
5	PC-to-Mote	28.16 ± 5.92ms	23.72 ± 0.56ms	3

Fig. 23. The Latency of Each Agent Migration Stage

bounce between two WSNs indefinitely.

A summary of all the results are shown in Figure 23. Note the column listing the number of points filtered. These were outlier points with values orders of magnitude above the mean. They are most likely caused by inaccuracies in Java's `System.nanoTime()` method and garbage collection. Since these points are relatively sparse, we filtered them out when presenting the graphs below. The latencies of both the filtered and unfiltered data are listed. Details of each stage are now presented.

Stage 1: Mote-to-PC. Measuring the latency of an agent moving from the gateway onto the attached base station is challenging because it starts and ends on different systems with clocks that are difficult to synchronize. To overcome this problem, we used a technique similar to that used in measuring the Mote-to-PC tuple space operations. Specifically, when an agent initiates a migration, it first performs a **rrdp** on the base station's tuple space to find the destination WSN's advertisement tuple. We exploit this by measuring the time from when the **rrdp** request is received to when the agent arrives on the base station. Since the latency of the **rrdp** is known from the benchmarks presented in Section VI-A, the latency of the mote-to-pc migration can be calculated. The experiment is repeated 1000 times; the results are shown in Figure 24. Note that the figure shows filtered data where extraneous points with values several orders of magnitude above the average are removed. With the extraneous points, the average latency is 41.29 ± 1.13ms. Without them, the average latency is 40.73 ± 0.52ms.

In this benchmark, we begin measuring the time from when the **rrdp** operation arrives at the base station. Thus, the results of the **rrdp** operation must return to the sensor before the migration actually begins. We can approximate the time spent actually migrating by subtracting half the cost of a mote-to-PC **rrdp** operation, or 5.18 ± 0.03ms, from the benchmark results. This gives us a migration latency of 36.12 ± 1.19ms for the unfiltered results, and 35.56 ± 0.58ms for the filtered results.

Stage 2: Agilla-to-Limone. When the migrating agent arrives on the PC, it is encapsulated in a Java `Agent` object and passed to the `AgimoneAgent`. Since this only involves invoking the constructor for `Agent` and a method call on `AgimoneAgent`,

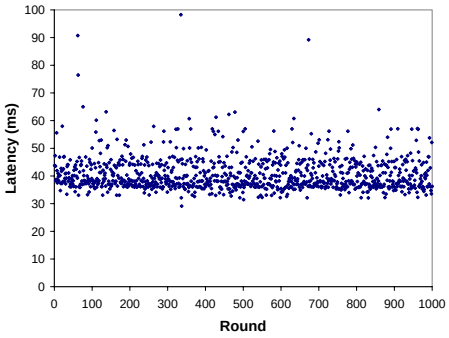

Fig. 24. Stage 1 Latency (Mote-to-PC)

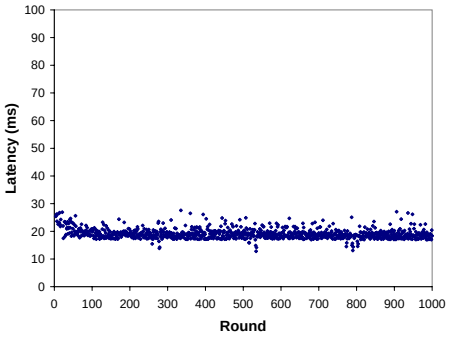

Fig. 25. Stage 3 Latency (PCtoPC)

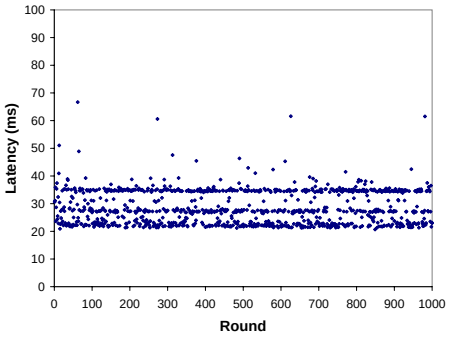

Fig. 26. Stage 5 Latency (PC-to-Mote)

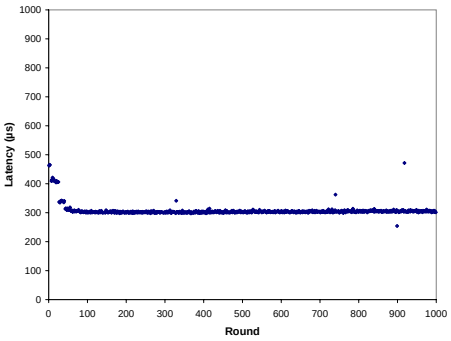

Fig. 27. Stage 2 Latency (Agilla-to-Limone)

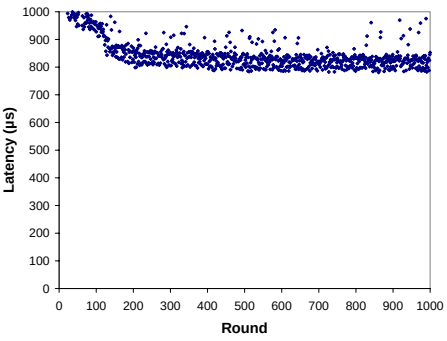

Fig. 28. Stage 4 Latency (Limone-to-Agilla)

the cost of passing the agent across middleware boundaries should be negligible relative to the other stages. To measure this, a bouncing agent is injected into the network and the time from when the last migration message is received on the base station, to when `AgimoneAgent`'s callback method begins executing is recorded. This experiment is repeated 1000 times; the results are shown in Figure 27. Note that the results included 182 extraneous data points, which is more than any other stage. This is most likely stemming from inaccuracies in Java coupled with the very small reading. With the spikes the average latency is 1.03 ± 0.16ms. With the spikes filtered out, the average latency is $307.11 \pm 1.59\mu$s. The large difference in filtered and unfiltered results is due to the number of extraneous points in the data set.

Stage 3: PC-to-PC. When the `AgimoneAgent` receives the migrating agent, it saves the agent within a Limone tuple and places it in the destination base station's tuple space using Limone's **rout** operation. The latency of migrating an agent over the IP network is measured by bouncing an agent between two `AgimoneAgents`. The total time is halved to obtain the one way latency. This process is repeated 1000 times; the results are shown in Figure 25. With the extraneous points, the average latency is 19.45 ± 0.26ms. With the spikes filtered out, the average latency is 19.11 ± 0.15ms.

Stage 4: Limone-to-Agilla. When the tuple containing the migrating agent is placed within the destination base station's tuple space, the `AgimoneAgent` on the base station reacts to this tuple by extracting the agent and passing it to Agilla's `AgentInjector`. Like Agilla-to-Limone, this process occurs entirely on the base station, so its latency should be negligible relative to the other stages. For this stage, the time from when the tuple is placed into the tuple space to when the agent is passed to the `AgentInjector` is recorded. This experiment is repeated 1000 times; the results are shown in Figure 28. With the extraneous points, the average latency is 1.13 ± 0.16ms. Without the extraneous points, the average latency is $830.05 \pm 2.26\mu$s.

As expected, this is negligible relative to the other stages. Notice that this stage is significantly longer than Agilla-to-Limone. This is because the Limone middleware must perform some computations to decide which local agents to propagate the reactions to, which involves consulting several local data structures, e.g., a reaction registry.

Stage 5: PC-to-Mote. When the `AgentInjector` receives the migrating agent, it injects the agent into the local WSN. Like Mote-to-PC, measuring this latency directly is difficult. The latency is indirectly calculated by taking advantage of the fact that, upon injection, the bouncing agent immediately performs a **rrdp** on the base station's tuple space searching for a network advertisement tuple. For each round, the time from when the agent is injected to when **rrdp** request arrives is recorded. This experiment is repeated 1000 times; the results are shown in Figure 26. With the extraneous points, the average latency is 33.33 ± 5.86ms. Without the extraneous points, the average latency is 28.89 ± 0.50ms.

As with stage 1, we include half of a Mote-to-PC **rrdp** operation in this benchmark. Subtracting half the cost of this operation, we get 28.16 ± 5.92ms for the unfiltered results, and 23.72 ± 0.56ms for the filtered results. This is slightly less than stage 1 because the majority of the processing is done by the sender, which in this stage is a laptop as opposed to a sensor.

C. Overall Performance

The following benchmarks evaluate the latency of common sequences of operations. The first set of benchmark (In-and-Out) answers the following question: what is the minimum amount of time it will take for an agent to enter a WSN and return? In a real-world scenario, the agent will perform some application-specific operations while in the network. However, for evaluation purposes, we inject an agent that immediately migrates back onto the base station after it arrives on a mote. The second set of benchmarks (End-to-End) evaluates how long it takes for an agent residing in one WSN to migrate into another WSN and back. The previous section analyzed each stage of the migration process individually; theoretically, the In-and-Out and End-to-End delays will be the sum of their individual stages. These benchmarks serve to validate the results of previous benchmarks.

While Agimone simplifies programming and increases network flexibility, its use of virtual machines results in some overhead. To quantify this overhead, a native-code implementation of In-And-Out and End-to-End is tested and the results are plotted along with the Agimone results. Instead of using agents, the native code implementation only exchanges data. However, the amount of data exchanged is the equal to the amount of data and code sent by the mobile agent solution.

Finally, all of the previous benchmarks used the same 36-byte bouncing agent. To gain insight on how the size of the agent affects inter-WSN migration latency, the last benchmark (Exploding-Agent) repeats the End-to-End experiment, but with successively larger agents.

In-and-Out. This benchmark measures the time from when a bouncing agent is injected onto the gateway to when it returns. Note that when the agent arrives on the gateway, it must first perform a **rrdp** on the base station's tuple space to determine the destination WSN's advertisement tuple. Thus, this benchmark measures the aggregate of the Mote-to-PC, PC-to-Mote, Limone-to-Agilla, and Agilla-to-Limone migration operations, and the Mote-to-PC tuple space operation. The benchmark is repeated 1000 times and the results are shown in Figure 29. The average In-and-Out latency is 62.18 ± 6.09ms unfiltered, 56.56 ± 0.27ms filtered. This is approximately the aggregate of the constituent stages (stages 1, 2, 3, and 4).

The native implementation of In-and-Out consists of a Java application on the base station that queries the attached gateway

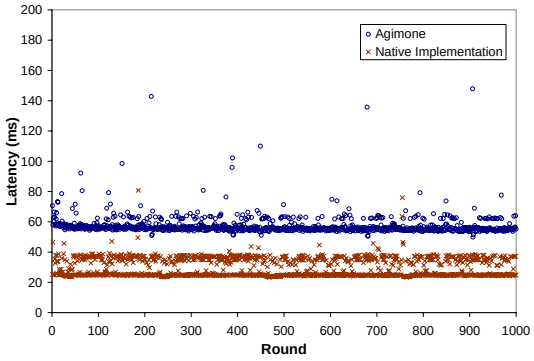

Fig. 29. The In-and-Out Agent Migration Latency.

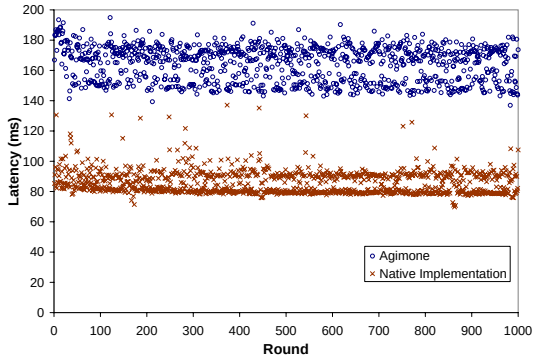

Fig. 30. The End-to-End Migration Latency

sensor by sending it two TinyOS packets totalling 36 bytes. Each message is acknowledged before the next message is sent. Once the gateway receives the query, it responds with another two TinyOS packets totalling 36 bytes. The benchmark starts the timer when the query is issued, and stops it when the response is received. This process is repeated 1000 times; the results are shown in Figure 29. The native implementation has an average latency of 30.09 ± 0.51ms, which is 27.39ms faster than the Agimone implementation.

End-to-End. The End-to-End latency is measured by injecting a bouncing agent into the WSN and recording the times when the `AgimoneAgent` passes it to the `AgentInjector` (stage 4 of the migration). The intervals between these times represent the agent's round-trip time. The benchmark monitors the bouncing agent as it makes 1000 round-trips; the results are shown in Figure 30. The average end-to-end round trip time is 179.19 ± 9.96ms unfiltered, and 164.75 ± 0.96ms filtered. This closely matches the aggregate of the various stages involved.

The native implementation of End-to-End does everything In-And-Out does except, after receiving the query results from the local gateway, the base station sends a 36-byte packet over the IP network to a remote base station. When the remote base station receives this packet, it queries its local gateway and sends a 36-byte reply. The benchmark starts the timer just prior to quering the local gateway, and stops it when the reply from the remote base station is received. This process is repeated 1000 times; the results are shown in Figure 30. The native implementation has an average latency of 86.36 ± 2.15ms, which is 78.39ms faster than the Agimone implementation.

Exploding-Agent. Until now, all aforementioned benchmarks used the relatively small bouncing agent. To gain insight on how the size of the agent affects the latency of inter-network migration, this benchmark repeats End-to-End with successively larger agents. The agent size is increased by inserting variables onto the agent's heap, stack, and reaction registry. Each agent is allowed to bounce between two WSNs 10 times, and the agent's round-trip time is recorded. The results are shown in Figure 31. They show that the cost of agent migration is roughly linear with the agent's size. As the size of the agent increases, its migration time is dominated more by the slow mote-to-mote and mote-to-PC stages.

The benchmarks presented in this section provide a general overview of Agimone's performance and overhead. All inter-network tuple space operations, regardless of direction, take about 10.5ms. **rinp** takes slightly longer than **rout** and **rrdp**

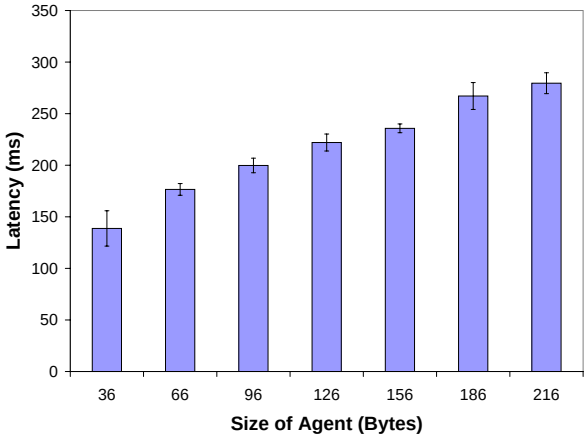

Fig. 31. End-to-End Latency vs. Size of Agent

because it has to remove a tuple, which involves more bookkeeping. A mobile agent takes about 82.5ms to migrate from one WSN to another. Of this, approximately 60ms is spent moving to and from the WSN and its base station, and 20ms is spent traversing the IP network. The latency of migrating into a WSN and back is about 60ms. Of this, most of the time ($>$57ms) is spent on the serial link between the base station and WSN gateway. The actual transition from Agilla to Limone takes about 307μs, while going from Limone to Agilla takes about 830μs. The overhead of Agimone compared to native code varies depending on the task. In the two operations presented, In-And-Out and End-to-End, there was a 27.39ms and 78.39ms increase in execution time relative to native code, respectively. Native code, however, is not nearly as flexible as mobile agents, and presumably requires more development time. Finally, an agent's migration latency increases linearly with its size.

VII. Related Work

There are a number of middleware systems that increase the flexibility of WSNs by enabling in-network reprogramming. They include XNP [5], Deluge [18], Maté [21], SensorWare [6], Impala [23], and Smart Messages [20]. There are also a number of coordination middleware designed for IP networks. They include JEDI [9], LIME [26], and MARS [7]. These middleware systems are either targeted for WSNs, or IP networks, but not the integration of heterogeneous networks. Recent efforts at building systems that integrate the two types of networks are more closely related to our work.

The Hourglass [28] and Stream-based Overlay Networks (SBONs) [27] systems cooperate to integrate multiple WSNs with the Internet. Specifically, they form an overlay network over the Internet consisting of servers that are connected to various WSNs. The system routes data streams generated by nodes within a WSN to applications on the Internet that require the data. The servers within the overlay network provide resource registration and discovery services. They are capable of dynamically adapting to network conditions by installing stream operators like data filters and aggregators to the source, for example, to reduce network usage in times of congestion. The Hourglass-SBON system differs from Agimone by focusing on delivering data streams generated within WSNs to consumers on the Internet. It is tailored specifically to data-streaming applications. Agimone, on the other hand, is a general-purpose middleware system that focuses on providing a seamless infrastructure for mobile agents to migrate and cooperate across WSNs and IP networks.

Tenet [14] provides a two-tiered architecture where the lower tier consists of resource-poor sensors and the higher tier consists of relatively powerful computers connected via an IP network. The higher tier computers are *masters* that control the sensors, which are directly addressable and contain a library of generic functions. Masters send sensors tasks that activate these functions in an application-specific way. Tenet restricts the sensors to perform simple functions and organize into routing trees anchored at the masters. It uses a well-tested routing structure within the resource-poor sensors and moves much of the complexity associated with application development onto second-tier devices that can be more easily debugged, thus simplifying application development. Unlike Agimone, Tenet reprograms the sensors using tasks that cannot control where they are executed or carry state across nodes. Therefore, Agimone provides a more flexible infrastructure for deploying adaptive applications. Also Tenet uses messages as the basic unit of communication. Message passing tightly couples communicating entities; both must be present for information to be exchanged. Agilla uses tuple spaces that decouple communication between agents residing on different networks further ensuring system flexibility.

SERUN [22] uses a three-level network architecture where the lowest level consists of cheap low-power sensors that gather data, the middle level consists of more capable *microservers* that process the data, and the highest level consists of PC-class systems where end-users can issue queries. When a query is issued, a task is sent to a microserver. The microserver runs the task by querying one or more sensors and processing the data according to the instructions within the task. To save energy, SERUN codifies the tasks as multiple publish-subscribe components, and forming a service composition graph on each microserver. When a task is received, the microserver attempts to add the task's components to the graph so as to reuse as much sensor data as possible. SERUN differs from Agimone in that it moves much of the application-specific code away from the low-power sensors and onto the microservers, and its tasks cannot autonomously migrate across microservers.

IrisNet [13] diverges from traditional WSNs consisting of resource-poor devices by proposing an Internet-scale sensor network consisting of desktop PCs with low-cost sensors, e.g., web cams. The IrisNet platform provides a query service for obtaining sensor data from anywhere on the Internet. Functionally, it is similar to TinyDB [24] in that it treats the network as a database, and allows users to query for data. However, IrisNet is larger scale and operates on relatively powerful machines. It is tailored to meet the unique challenges associated with an Internet-scale application, e.g., security, privacy, data robustness, and saleability. IrisNet differs from Agimone by transforming the IP network into a giant sensor network. Agimone is complimentary to IrisNet in that it may provide a mechanism for seamlessly integrating an IrisNet with resource-poor WSNs.

VIII. Conclusion and Future Work

In this paper, we have presented Agimone, a middleware system for integrating WSNs over the Internet and other IP networks. We have implemented an efficient integration layer that integrates Agilla and Limone, two existing mobile agent middleware platforms. By adapting an existing cargo tracking application to use Agimone, we have demonstrated how developers can easily take advantage of the functionality we provide. Our empirical performance data demonstrates the efficiency of our middleware on existing sensor and base station hardware. Tuples can cross middleware boundaries in approximately 10.5ms, and agents can migrate between WSNs and back in 82.5ms. Though there is some runtime overhead associated with using mobile agents as compared to native code, the increase in developer productivity outweighs this performance penalty for all but the most

time-critical of applications.

There are two specific areas in Agimone that merit continued development. First, though Agilla's mobile agent paradigm greatly simplifies WSN application development, agents must be developed directly in Agilla bytecode. This procedure can be disorienting at first for developers who are not accustomed to working at a bytecode level. WSN application development could be made even more straightforward by implementing a high-level language which compiles to Agilla bytecode.

Second, the Limone registry used for centralized service discovery is not designed with scalability in mind. It is unclear how well the registry will perform when many agents are connected. This shortcoming can be resolved by adapting a more robust device discovery mechanism, like WSDL, to implement the Limone device discovery API. By implementing these new device discovery providers, WSN applications could be more easily deployed to large IP networks like the Internet.

ACKNOWLEDGMENT

This research is supported by the Office of Naval Research under MURI research contract N00014-02-1-0715 and by the the NSF under NOSS contract CNS-0520220. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors. We would also like to thank Boeing Corporation for their support on an earlier version of the cargo tracking application.

REFERENCES

[1] `http://platformx.sourceforge.net/`.

[2] http://sourceforge.net/cvs/?group_id=28656.

[3] http://mobilab.wustl.edu/projects/agilla.

[4] `http://mobilab.wustl.edu/projects/agilla/download/index.html`.

[5] `http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf`.

[6] BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys* (May 2003), USENIX, pp. 187–200.

[7] CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. MARS: A programmable coordination architecture for mobile agents. *Internet Computing 4*, 4 (2000), 26–35.

[8] CHESHIRE, S. DNS-based service discovery. Tech. rep., Apple Computer, Inc., 2005.

[9] CUGOLA, G., NITTO, E. D., AND FUGGETTA, A. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering 27*, 9 (September 2001), 827–850.

[10] CULLER, D., ESTRIN, D., AND SRIVASTAVA, M. Overview of sensor networks. *IEEE Computer 37*, 8 (2004), 41–49.

[11] FOK, C.-L., ROMAN, G.-C., AND HACKMANN, G. A Lightweight Coordination Middleware for Mobile Computing. In *Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004)* (February 2004), R. DeNicola, G. Ferrari, and G. Meredith, Eds., no. 2949 in Lecture Notes in Computer Science, Springer-Verlag, pp. 135–151.

[12] FOK, C.-L., ROMAN, G.-C., AND LU, C. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)* (June 2005), IEEE, pp. 653–662.

[13] GIBBONS, P., CARP, B., KE, Y., NATH, S., AND SESHAN, S. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing* (October-December 2003), 22–33.

[14] GOVINDAN, R., KOHLER, E., ESTRIN, D., BIAN, F., CHINTALAPUDI, K., GNAWALI, O., RANGWALA, S., GUMMADI, R., AND STATHOPOULOS, T. Tenet: An architecture for tiered embedded networks. Tech. Rep. CENS-TR-56, UCLA CENS, November 2005.

[15] HACKMANN, G., FOK, C.-L., ROMAN, G.-C., LU, C., ZUVER, C., ENGLISH, K., AND MEIER, J. Demo abstract: Agile cargo tracking using mobile agents. In *Proceedings of the 3rd Annual Conference on Embedded Networked Sensor Systems (SenSys'05)* (November 2005), ACM, p. 303.

[16] HE, T., KRISHNAMURTHY, S., STANKOVIC, J. A., ABDELZAHER, T., LUO, L., STOLERU, R., YAN, T., GU, L., HUI, J., AND KROGH, B. Energy-efficient surveillance system using wireless sensor networks. In *MobiSYS '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2004), ACM Press, pp. 270–283.

[17] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems* (2000), pp. 93–104.

[18] HUI, J., AND CULLER, D. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004), ACM Press, pp. 81–94.

[19] JUANG, P., OKI, H., WANG, Y., MARTONOSI, M., PEH, L. S., AND RUBENSTEIN, D. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGPLAN Not. 37*, 10 (2002), 96–107.

[20] KANG, P., BORCEA, C., XU, G., SAXENA, A., KREMER, U., AND IFTODE, L. Smart messages: A distributed computing platform for networks of embedded systems. *Special Issue on Mobile and Pervasive Computing, The Computer Journal 47* (2004), 475–494.

[21] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM Press, pp. 85–95.

[22] LIU, J., CHEONG, E., AND ZHAO, F. Semantics-based optimization across uncoordinated tasks in networked embedded systems. Tech. Rep. MSR-TR-2005-46, Microsoft Research, One Microsoft Way, Redmond, WA 98075, April 2005.

[23] LIU, T., AND MARTONOSI, M. Impala: A middleware system for managing autonomic, parallel sensor systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2003).

[24] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD Int. Conf. on Management of Data* (2003), pp. 491 – 502.

[25] MALAN, D., FULFORD-JONES, T., WELSH, M., AND MOULTON, S. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *Proceedings of the MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004)* (June 2004).

[26] PICCO, G., MURPHY, A., AND ROMAN, G.-C. LIME: Linda meets mobility. In *Proc. of the 21st Int'l. Conf. on Software Engineering* (May 1999).

[27] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., , AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06, to appear)* (April 2006).

[28] SHNEIDMAN, J., PIETZUCH, P., LEDLIE, J., ROUSSOPOULOS, M., SELTZER, M., AND WELSH, M. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Tech. Rep. TR-21-04, Harvard, 2004.

[29] SUN MICROSYSTEMS, INC. Personal profile. `http://java.sun.com/products/personalprofile/index.jsp`.

[30] W3C-XML-ACTIVITY-ON-XML-PROTOCOLS. W3c recommendation: Web services description language 1.1. http://www.w3.org/TR/wsdl, October 2003.

[31] ZHAO, J., AND GOVINDAN, R. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. of the ACM SenSys* (2003).