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, St. Louis MO 63130-4899, USA

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. Agimone is the first system that allows mobile agents to migrate between sensor and IP networks. 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. We also provide an empirical evaluation that demonstrates the efficiency of Agimone using two WSNs consisting of Mica2 motes connected by an IP network.

1 Introduction

Wireless sensor networks (WSNs) consist of tiny sensors embedded within the environment. Many applications require that sensor nodes be deeply embedded in areas where they are difficult to physically access, such as scattered in forests, making it is impractical to physically gather the nodes in order to collect data or deploy new applications. This necessitates WSN systems in which the nodes operate for very long periods of time without physical access. Thus, data collection and application deployment is done over wireless networks. These long system lifetimes also mandate the flexibility 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 [1] operating system hardwires software components. Once deployed, application behavior can only be marginally tweaked by changing specific parameters defined prior to deployment. To complicate matters, the sensors' power consumption 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

range and reliability be sacrificed [2]. These limitations impose severe restrictions on the complexity and scope of WSN applications.

Many of these restrictions can be eased by logically combining multiple, physically disconnected WSNs using a common IP network. For example, WSNs can be used for cargo tracking and monitoring by attaching sensors to individual cargo containers. However, 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 [3], can act as gateways between the IP network and their respective WSNs. By coordinating these disjoint networks to act as one logical network, sophisticated WSN applications can be developed. This way, 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 task, since WSNs are constantly forming and reshaping as the application evolves. Hence, WSN nodes must be able to determine the availability of other WSNs at run-time. Further, agent transactions across hosts should not be 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 Agilla [4], a middleware for wireless sensors. Limone [5], 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 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 2, we discuss the shortcomings of the state-of-the-art and explain the motivation behind our general-purpose integration layer. Section 3 provides a brief overview of the programming models used by Agilla and Limone. Section 4 describes Agimone's architecture. Section 5 presents a cargo tracking application that highlights the capabilities and expressiveness of Agimone. A performance evaluation is provided in Section 6. We discuss related middleware systems in Section 7. Finally, we conclude in Section 8.

2 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 realized using a middleware system that integrates IP networks and WSNs.

2.1 Cargo Tracking: A Motivating Application

Consider the problem of cargo tracking. 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 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 all of these users' needs ahead-of-time, and so deploying a single monolithic application on each sensor is infeasible. Mobile agents are invaluable for this scenario. Each authorized user can deploy custom mobile agents to query the sensors on the containers.

However, the limited radio range of individual sensors forces WSNs to form physically-localized clusters. If we rely solely on the sensors' radios, users must interact individually with each of these clusters. This requirement is unreasonable and greatly limits the sophistication of WSN applications. Instead, the current state-of-the-art is to deploy base stations in each cluster. These base stations are connected together using a common IP network. This provides an infrastructure which WSN applications can exploit for inter-WSN interactions. It also provides a means for sensors to interact with clients on the IP network.

2.2 Challenges

Though these capabilities are essential, they are difficult to satisfy. The sensors that populate WSNs have vastly different capabilities from the devices connected to the IP network, preventing the deployment of a uniform software layer across all devices. Today, complex WSN applications consist of separate software support platforms for WSNs and the IP network. Application-specific software is

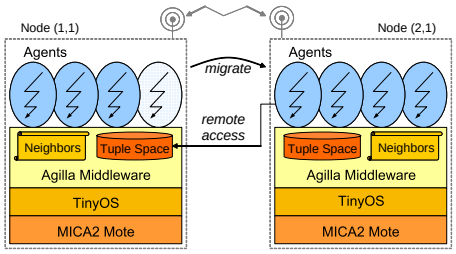

Fig. 1. The Agilla Architecture

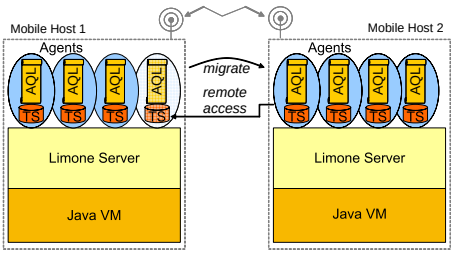

Fig. 2. The Limone Architecture

used to pass messages and translate queries between these two classes of devices. However, writing this support layer requires programming experience with both types of devices. Also, this layer must often be modified and redeployed when application features or protocols change. This is unacceptable for applications which have a constantly-evolving set of capabilities, like cargo tracking.

In this work, we present a middleware platform that supports seamless integration of WSNs and IP networks into a uniform software platform. Our middleware's services facilitate the development of WSN applications which exploit the IP network as a resource for computation and communication. Mobile agents in a WSN are provided 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 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 application described above.

3 Background

This section provides a brief overview of the programming models offered by Agilla and Limone. More details on the implementation are available in [4] and [5].

3.1 Agilla

Agilla programs are 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. An agent may execute special instructions that allow it to interact with the environment and move across nodes. Multiple agents can coexist on a 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 [6], but tailored to the mobile agent paradigm. Like Maté, most Agilla instructions fit in a single byte. Agilla is available for Mica2, MicaZ, and Tyndall25 nodes and is distributed through TinyOS' source respository. See Agilla's website [7] 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. Tuple spaces offer many of the same programming benefits as shared data systems, but with far less message-passing.

Each sensor in the WSN has a single local tuple space. 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. **rd** and **in** operations respectively remove and copy tuples from a tuple space; these operations are parameterized by *templates* that specify the forms of matching tuples. In Agilla, tuple and template fields contain one of a handful of well-known 16-bit data types (integer, string, sensor reading, etc.). Alternatively, a template's field may contain a type (e.g., `VALUE` or `STRING`) rather than a specific value. This indicates that any value of the corresponding type is acceptable. The code snippets in Figures 3 and 4 demonstrate the **out** and **rd** operations, respectively.

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

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

The **rd** or **in** operations block until a matching tuple is available. Agents may also perform *probing* (non-blocking) tuple removals and copies using a different set of primitives. *Remote* tuple space primitives manipulate tuple spaces residing on remote sensors. 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. These operations are described in further detail in [4].

Agilla agents may move or clone onto other hosts in the WSN using either *weak* or *strong* migration operations. 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.

3.2 Limone

Limone provides a similar agent-based programming model using tuple spaces for inter-agent communication. Its architecture is shown in Figure 2. Limone supports the same primitive tuple space operations as Agilla, as well as an analogous reaction mechanism. 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 by their Agilla counterparts. Fields in Limone tuples are indexed by a user-specified name and can contain any Java data type of any size. Similarly, Limone templates are more flexible than Agilla templates. In addition to matching by name and type, Limone templates use *constraint functions* to provide a fine-grained way to specify matching values. For example, the constraint <"ID", Integer, GreaterThanConstraint(10)> matches fields named "ID" that contain an Integer greater than 10. Most constraints use either DefaultConstraintFunction (match any value of the correct type), or EquivalencyConstraintFunction (match only the specified value). Figures 5 and 6 provide code which demonstrate the syntax of Limone's **out** and **rd** operations.

```
ETuple t = new ETuple();
t.addField(new EField("ID", 15));
// Field <ID: 15>
t.addField(new EField("Flag",
    "mark"));
// Field <Flag: "mark">
getTS().out(t);
// out(<ID: 15, Flag: "mark">
```

Fig. 5. Limone **out** Code Snippet

```
ETemplate t = new ETemplate();
t.addConstraint(new EConstraint("ID",
    Integer.class, new DefaultConstraint-
    Function()));
// Match field ID containing any Integer
t.addConstraint(new EConstraint("Flag",
    String.class, new Equivalency-
    ConstraintFunction("mark")));
// Match field flag containing "mark"
ETuple tuple = getTS().rd(t);
// rd a tuple matching this template
```

Fig. 6. Limone **rd** Code Snippet

4 Architecture of Agimone

We have constructed the Agimone architecture (shown in Figure 7) 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 WSN node 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 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 discover each other using beacons where multicast routing is supported, or a centralized service directory elsewhere. We have implemented a simple Limone service registry that is suitable for a small number of agents.

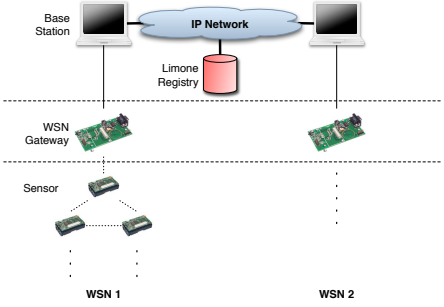

Fig. 7. Agimone Network Architecture

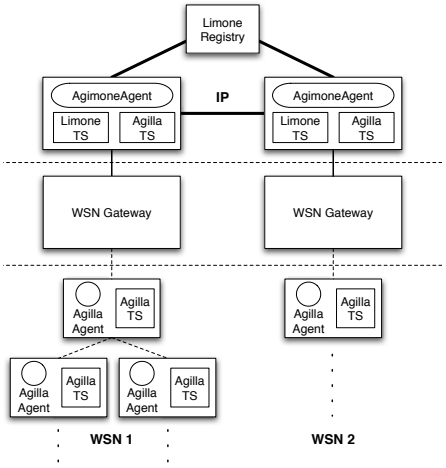

Fig. 8. Agimone System Components

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 greater scalability can use a more sophisticated protocol, like WSDL [8].

Agimone is populated with the following components, as shown in Figure 8:

- 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.

In the rest of this section, we will describe Agimone's services in further detail.

4.1 WSN Discovery

Since new WSNs are formed and destroyed as the applications evolve, it is often necessary for agents in the WSNs to be aware of these changes at runtime. 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 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 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` beacons a well-known Limone registry with messages containing its WSN advertisement. The Limone registry forwards these advertisements to other Limone agents. Similarly, the registry notifies Limone agents when hosts leave the 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.

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

Fig. 9. WSN Discovery Code Snippet

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

Fig. 10. Migration Code Snippet

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 code in Figure 9 places any available WSN advertisement containing a string on top of the Agilla agent's operand stack.

4.2 Migration Across WSNs

Using these advertisements, Agilla agents can select other WSNs and migrate to them with the assistance of the `AgimoneAgent`. This procedure is detailed in Figure 11. WSN advertisements are distributed in Steps 1 and 2, and placed in the base stations' Agilla tuple space in Step 3. The Agilla agent selects one of these WSN advertisements in Step 4 and places it on top of its operand stack.

Once the Agilla agent has an acceptable advertisement on its operand stack, it performs a strong migration to the WSN gateway, as shown in Step 5. Sample code to perform this operation is listed in Figure 10. 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 Limone tuple space of the destination network's `AgimoneAgent`.

On initialization, `AgimoneAgent` installs a reaction on its tuple space that notifies it of tuples in the form <Agent: Agilla Agent>. Thus, in Step 8, the

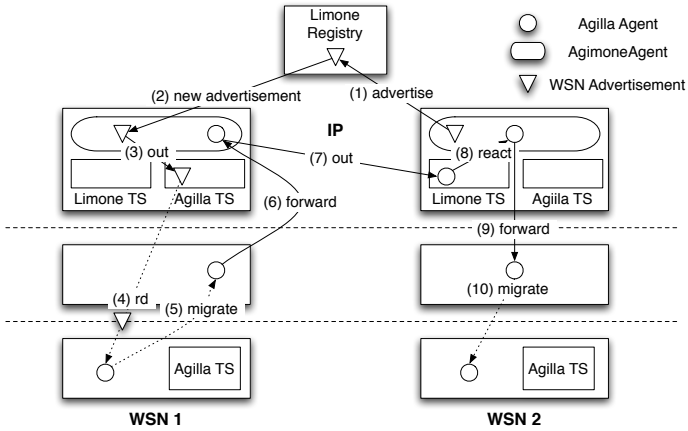

Fig. 11. Agilla Agent Migration Across Different WSNs

`AgimoneAgent` on the destination base station is notified of the tuple's arrival. It extracts the agent from the tuple and injects it into the WSN gateway in Step 9. In Step 10, the agent migrates to the new WSN, where it resumes computation.

This process involves several transactions across WSNs and the IP network. However, this is transparent to the Agilla agent developer, who only invokes a single migration operation to the base station. Thus, developers can leverage the Limone network's infrastructure while still using the familiar Agilla APIs.

4.3 Cross-Middleware Interactions Via Tuple Spaces

So far, we have only considered the IP network as a way for distant WSNs to interact. However, it can also be used to support interactions between devices in a WSN and devices on the IP network. Because of the limited computational powers of wireless sensors, Agilla agents may wish to use devices on the IP network as a computational resource. Likewise, a Limone agent may wish to exploit the sensing resources of a remote WSN. Both goals can be achieved by giving Limone agents access to the Agilla tuple space that resides on each base station, providing both types of agents with 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, though Limone agents can reside on any host in the IP network, they would only be able to interact with a WSN if they reside on a base station within its radio range.

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 this Limone tuple space. The `AgimoneAgent` translates these operations to their Agilla equivalents and forwards them to the Agilla API. Hence, any tuples placed by Limone agents into this tuple space are available to

Agilla agents in the corresponding WSN, and vice-versa. These Limone agents need not have a WSN gateway attached to their host to interact with the WSN, since an `AgimoneAgent` will communicate with the WSN on their behalf. Limone and Agilla agents interact with this shared tuple space using their respective APIs. So, developers who are only familiar with one of these systems can still leverage resources made available by the other, without first learning a new API.

However, as discussed earlier in Section 3, there are restrictions on Agilla tuples and templates that do not exist in Limone. For example, a Limone agent may try to place the tuple <ID: 3.14, Flag: "mark"> in the `AgimoneAgent`'s tuple space. Since Agilla does not have a floating-point data type, there is no way to convert this Limone tuple to an equivalent Agilla tuple. To resolve this problem, the `AgimoneAgent` uses Limone's rejection mechanism to filter incoming tuple space operations. This mechanism allows agents to reject any remote operations issued on their tuple space. The `AgimoneAgent` places the following restrictions on all incoming tuples and templates:

- Fields cannot be named arbitrarily. 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 must contain Agilla data types.
- The only constraint functions are DefaultConstraintFunction (i.e., match by type) or EquivalencyConstraintFunction (i.e., match by exact value).

The `AgimoneAgent` will reject all non-conforming operations, since they have no Agilla equivalents. Conforming operations are converted to their Agilla counterparts and forwarded to the Agilla tuple space. The results are converted from Agilla tuples to Limone tuples (using the conventions specified above) and sent back to the request's originator.

4.4 Implementation Details

Agilla and Limone have been implemented and deployed on a wide variety of hardware. Agilla has two parts: a NesC-based portion that is installed on sensors, and a Java-based `AgentInjector` that is installed on base stations. Since storage is at a premium on many sensors, Agilla is necessarily compact: it consumes 49.66KB of flash ROM and 3.07KB of RAM. Agilla supports 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 [9].

The Limone and Agimone packages are developed in Java according to the J2ME Personal Profile 1.0 [10] specification. This allows deployment on devices like PDAs and Stargates which cannot host full Java Standard Edition runtimes, as well as on desktop and laptop computers. Limone was designed for deployment on storage-constrained devices like PDAs: its bytecode distribution consumes only 132KB of storage space. Agimone is even more compact: it consumes 13KB of storage space. Agimone operates on any platform supported by Limone, which includes Windows Mobile, Windows XP, Linux, Solaris, and Mac OS X.

5 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 one such application that is well-suited for implementation using Agimone. As discussed in Section 2, cargo containers can be equipped with sensors that form WSNs in localized clusters. Many of these containers are located in remote warehouses and vehicles. So, users must be able to interact with these clusters without needing to be within the WSN's communication range. This can be achieved by connecting the WSNs' base stations using a common IP network, then deploying Agimone on them so that queries may traverse either network as needed.

In this section, we present a prototype application that uses mobile agents to track cargo. Our group had developed a similar application (demonstrated at SenSys '05 [11]) using a custom Limone agent to marshal messages between the sensor and IP networks. This custom agent had to be repeatedly modified and redeployed as our application's feature set evolved, greatly complicating development efforts. These difficulties motivated the creation of Agimone and a complete redesign of the application around it, resulting in much cleaner code overall and a simpler deployment process. Although Agimone was motivated by the cargo tracking application, we emphasize that Agimone is a *general purpose* middleware with a uniform programming model that can be used for a broad class of applications that need to integrate multiple WSNs and the IP network.

In the interest of space, we provide here a brief overview of two agents that are part of this application. More in-depth information about the application, including sample code, may be found in [12].

5.1 Watchdog Agents

Sensors attached to shipping containers can be equipped with various inexpensive sensor boards which can be used to detect attempted intrusions into the containers. As a demonstration of this potential, we have implemented two prototype agents that monitor the sensor's accelerometer and light readings, respectively. These agents loop, repeatedly reading the sensor until an unusual reading is detected. When this happens, an event is recorded in the local tuple space, and an alert tuple is placed in the base station's tuple space.

The `AgimoneAgent` on the base station automatically exposes these alert tuples to the Limone network. Remote Limone clients on the IP network can register reactions for these tuples. Limone automatically notifies these clients when any new alerts are generated. We can then do whatever processing we desire with these alerts (e.g., log it to disk and notify security personnel).

As a testament to Agilla's expressiveness, the watchdog agent that monitors the light sensor contains only 17 lines of code. The Limone client requires only 11 lines of code to automatically receive alerts and extract their contents. The Agilla agent and the Limone client were developed in only a few hours.

5.2 Intrusion Search Agent

A user, such as a shipping company or a port authority, may later want to search all the containers for any tampering recorded by the watchdog agents. Consider a scenario where containers are being moved between a ship and a loading dock, each of which has a corresponding WSN and base station. These base stations are connected by an IP link, e.g., Ethernet or 802.11b. Though users can search both WSNs simultaneously, a comprehensive search may be unnecessarily expensive. Ideally, the scope of such a search should be determined at runtime. For example, the user may know that containers on the ship are far more susceptible to tampering than the dock. So, the search for tampered containers should begin on the ship. If one of these containers has been tampered with, then the search should automatically expand to the dock, in order to determine the scope of the security breach.

We have developed a sample Agilla agent which consults WSN advertisements at runtime to locate WSNs and apply this searching policy. This involved adding only 23 lines of code to the previous Agilla agent. Owing to Agimone's flexibility, the Limone client used to monitor the watchdog agents' alerts required no modifications to support this new agent's alerts. Further, no additional support code had to be deployed to the base stations to support inter-WSN migrations.

6 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 attached to an IBM R40 laptop via a 115.2Kbps serial link. The laptops are connected via a 100Mbps wired Ethernet link. Since they are on the same subnet, discovery is performed using multicast beacons rather than a Limone registry. The laptops are configured with a 1.5GHz Intel Pentium M processor, 512MB of RAM, Windows XP and Java Standard Edition 5.0. 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, and overall performance.

We have not compared the performance of Agimone to any other middleware systems. This is because to date no comparable systems exist: Agimone is currently the only middleware which supports the interaction of mobile agents across WSNs joined by an IP network. In this section, we focus on the cost of the inter-WSN operations supported by Agimone. The interested reader may consult [4] for a detailed discussion of Agilla's intra-WSN performance.

6.1 Tuple Space Operations

In the first set of benchmarks, we evaluate the cost of the tuple space operations **rinp**, **rrdp**, and **rout** across middleware boundaries. These operations may

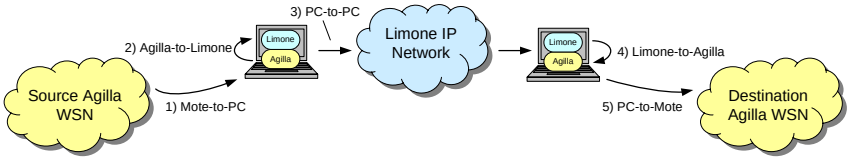

Fig. 13. The Five Stages of an Inter-WSN Agent Migration Operation.

be performed by the `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). In the interest of brevity, we only provide here a brief overview of the benchmarks. The interested reader may find more technical details in [12].

Mote-To-PC. The first set of benchmarks determine the latency of an Agilla agent on the WSN gateway accessing `AgimoneAgent`'s Agilla tuple space. We created three benchmark agents, each of which performs one of the remote tuple space operations (**rinp**, **rrdp**, and **rout**) 100 times, over which the mean was calculated. Each benchmark was repeated 100 times. The operations have an average latency of 10 to 11 ms, as shown in shown in Figure 12.

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. 12. The Latency of Remote Tuple Space Operations

PC-To-Mote. The second set of benchmarks repeats the same operation in the opposite direction. In this case, since the latency can be directly measured, each experiment calculates the latency of one operation execution. Figure 12 shows the average results from 100 runs of each benchmark. The mean latency of PC-to-Mote tuple space operations is 10 to 11 ms.

6.2 Agent Migration Operations

As discussed in Section 3.1, agent migrations enable agents located in one WSN to migrate across an IP network into another WSN. From an Agilla agent's perspective, an inter-WSN agent migration occurs by invoking a single operation. However, as discussed in Section 4, there many steps involved which are transparent to the agent. In this set of benchmarks, we identify five distinct stages involved in migrating a 36-byte agent across WSNs, as shown in shown in Figure 13, and measure the cost of each stage. Again, we refer the interested reader to [12] for more in-depth technical details.

The results of these benchmarks are shown in Figure 14. All benchmark results are presented as an average of 1000 runs. Note that stage 2 has a significant difference between mean and median latency. This difference is caused by sparse points with values orders of magnitude above the mean, which we suspect are caused by the process being interrupted by the OS or Java's garbage collector.

Stage 1: Mote-to-PC. Here, the agent moves from the source mote to the base station. We measured this procedure by deploying an agent which searches the `AgimoneAgent`'s tuple space for a WSN advertisement and then attempts to migrate to the base station. The mean latency of this stage is 36.12 ± 1.19ms.

Stage 2: Agilla-to-Limone. In this tage, the agent passes from the Agilla middleware on the base station to the Limone middleware. The cost of this operation should be negligible, since it only involves a few local method calls. This is borne out by our tests; the mean latency is 1.03 ± 0.16ms.

Stage	Mean Latency	Median Latency
1	36.12 ± 1.19ms	33.73ms
2	1.03 ± 0.16ms	303.95μs
3	19.45 ± 0.26ms	18.77ms
4	1.13 ± 0.16ms	834.74μs
5	28.16 ± 5.92ms	22.28ms

Fig. 14. The Latency of Each Agent Migration Stage (Average of 1000 Runs)

Stage 3: PC-to-PC. In this stage, the `AgimoneAgent` encapsulates the migrating agent into a Limone tuple and places it in the destination `AgimoneAgent`'s tuple space. We timed this stage by repeatedly migrating an agent between two base stations, then halving the round-trip time. This stage had a mean latency of 19.45 ± 0.26ms.

Stage 4: Limone-to-Agilla. In this stage, the `AgimoneAgent` extracts the encapsulated agent from the Limone tuple and passes it to Agilla's `AgentInjector`. Like stage 2, this only involves a few local method calls, so the latency should be negligible. We recorded the time between placing the tuple in the tuple space to passing the agent to the `AgentInjector`. The mean latency is 1.13 ± 0.16ms; as expected, this is negligible relative to other stages.

Stage 5: PC-to-Mote. In the final stage, the agent is injected into the destination WSN. Similarly to stage 1, we measured this latency by migrating an agent which immediately reads an advertisement tuple from the base station, and measuring the time between injection and receiving the tuple space request. The mean latency of this stage is 28.16 ± 5.92ms.

6.3 Overall Performance

The last set of benchmarks evaluate the latency of common sequences of operations. The In-and-Out benchmark measures the cost of migrating in and out of the same WSN. The End-to-End benchmark evaluates the cost of migrating from one WSN to a different WSN and back. These two benchmarks use the same 36-byte agent and are repeated 1000 times.

While Agimone simplifies programming and increases network flexibility, its use of virtual machines results in some overhead. We quantify this overhead by comparing the first two benchmarks above with native-code implementations. To isolate the cost of message-passing from execution, the native implementations exchange 36-byte data messages in place of 36-byte mobile agents.

In-and-Out. This benchmark injects an agent which migrates repeatedly between two WSNs, and measures the cost of moving the agent in and out of one WSN. When the agent is injected into the WSN, it immediately performs a **rrdp** to find the other WSN's advertisement, and then attempts to migrate to it.

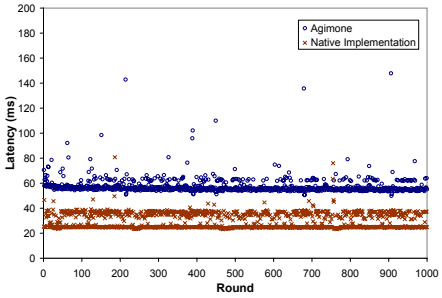

Fig. 15. The In-and-Out Agent Migration Latency.

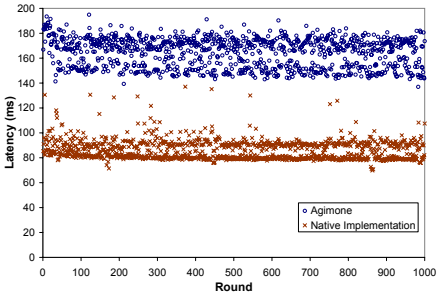

Fig. 16. The End-to-End Migration Latency

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 results of this benchmark are shown in Figure 15. The mean In-and-Out latency is 62.18 ± 6.09ms, with a 55.76ms median. This is approximately the aggregate of the constituent stages (1, 2, 3, and 4).

The native implementation of In-and-Out is a Java application that sends a 36-byte query to the attached gateway sensor in two TinyOS packets; the sensor immediately sends 36 bytes of data back. The benchmark measures the time from sending the request to receiving the response. The native implementation has a mean latency of 30.09 ± 0.51ms, and a median latency of 26.29ms.

End-to-End. The End-to-End latency is measured by injecting the same agent and recording its round-trip time over the IP network. The results are shown in Figure 16. The mean round trip time is 179.19 ± 9.96ms, with a median of 167.96ms. This closely matches the sum of the various stages involved.

The native implementation of End-to-End adds to the In-And-Out benchmark by sending a 36-byte packet over the IP network to a remote base station after receiving a response from the WSN. The remote base station sends a 36-byte reply. The benchmark measures the time from querying the sensor node to receiving a response from the remote base station. The native implementation has an mean latency of 86.36 ± 2.15ms, and a median latency of 84.38ms.

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. A mobile agent takes about 85.9ms to migrate from one WSN to another. Of this, approximately 65ms 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. Most of this time (>57ms) is spent on the serial link between the base station and WSN gateway. The actual transition from Agilla to Limone is less than 1ms in either direction. 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 32.09ms and 92.83ms 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.

7 Related Work

There are many middleware systems that increase WSN flexibility by enabling in-network reprogramming. They include XNP [13], Deluge [14], Maté [6], SensorWare [15], Impala [16], and Smart Messages [17]. There are also coordination middleware like LIME [18], and MARS [19] that are designed for IP networks. These middleware systems are either targeted for WSNs, or IP networks, but not the integration of both. Recent systems that integrate IP and sensor networks are more closely related to Agimone.

The Hourglass [20] and Stream-based Overlay Networks (SBONs) [21] systems form an overlay network over the Internet out of servers connected to various WSNs. The system routes data streams generated within WSNs to applications on the Internet. The system also provides resource registration and discovery services to servers. Servers dynamically adapt to network conditions by installing stream operators like data filters and aggregators on the source, e.g., to reduce network congestion. Hourglass-SBON focuses on delivering data streams generated within WSNs to consumers on the Internet. Agimone, on the other hand, is a general-purpose middleware system that supports agent migration and coordination across WSNs and IP networks, as well as data sharing.

Tenet [22] provides a two-tiered architecture partitioned into resource-poor sensors and relatively powerful computers connected via an IP network. The higher-tier computers directly control sensors, which service them using well-established protocols. This moves much of the application development onto more-powerful computers, simplifying debugging. Unlike Agimone, Tenet's tasks cannot relocate autonomously or carry state across nodes. Therefore, Agimone provides a more flexible infrastructure for deploying adaptive applications. Also, Tenet uses message passing as its basic communication paradigm, which easily fails in the face of transient link failures. Agilla uses tuples for all inter-agent communication, which survive temporal communication failures.

SERUN [23] uses a three-level network architecture divided into inexpensive data-gathering sensors, data-processing *microservers*, and PC-class systems where end-users can issue queries. When a query is issued, a task is sent to a microserver that queries one or more sensors and processes the data according to the task's instructions. 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 [24] diverges from traditional WSNs by proposing an Internet-scale sensor network consisting of desktop PCs with low-cost sensors, e.g., web cams. IrisNet provides a query service for obtaining sensor data from anywhere on the Internet. Functionally, it is similar to TinyDB [25] in that it treats the network as a database. However, since IrisNet operates on relatively powerful machines rather than embedded sensors, it is best suited for applications where sensing

capabilities are secondary to computational resources. In this sense, IrisNet is complementary to Agimone rather than an alternative.

8 Conclusion

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 layer that integrates Agilla and Limone, two existing mobile agent middleware platforms. By designing a cargo tracking application that uses 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. 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.

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. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
2. Zhao, J., Govindan, R.: Understanding packet delivery performance in dense wireless sensor networks. In: Proc. of the ACM SenSys. (2003)
3. (`http://platformx.sourceforge.net/`)
4. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS'05), IEEE (2005) 653–662
5. Fok, C.L., Roman, G.C., Hackmann, G.: A Lightweight Coordination Middleware for Mobile Computing. In DeNicola, R., Ferrari, G., Meredith, G., eds.: Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004). Number 2949 in Lecture Notes in Computer Science, Springer-Verlag (2004) 135–151
6. Levis, P., 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, ACM Press (2002) 85–95

7. (http://mobilab.wustl.edu/projects/agilla)
8. W3C-XML-Activity-On-XML-Protocols: W3c recommendation: Web services description language 1.1. http://www.w3.org/TR/wsdl (2003)
9. (http://mobilab.wustl.edu/projects/agilla/download/index.html)
10. (http://java.sun.com/products/personalprofile/index.jsp)
11. Hackmann, G., Fok, C.L., Roman, G.C., Lu, C., Zuver, C., English, K., Meier, J.: Demo abstract: Agile cargo tracking using mobile agents. In: Proceedings of the 3rd Annual Conference on Embedded Networked Sensor Systems (SenSys'05), ACM (2005) 303
12. Hackmann, G., Fok, C.L., Roman, G.C., Lu, C.: Agimone: Middleware support for seamless integration of sensor and ip networks. Technical Report WUCSE-05-56, Washington University in St. Louis Department of Computer Science and Engineering (2005)
13. (http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf)
14. Hui, J., 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, ACM Press (2004) 81–94
15. Boulis, A., Han, C.C., Srivastava, M.: Design and implementation of a framework for efficient and programmable sensor networks. In: Proc. of MobiSys, USENIX (2003) 187–200
16. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2003)
17. Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., 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
18. Picco, G., Murphy, A., Roman, G.C.: LIME: Linda meets mobility. In: Proc. of the 21^{st} Int'l. Conf. on Software Engineering. (1999)
19. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4**(4) (2000) 26–35
20. Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., Welsh, M.: Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard (2004)
21. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., , Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proc. of the 22nd International Conference on Data Engineering (ICDE'06, to appear). (2006)
22. Govindan, R., Kohler, E., Estrin, D., Bian, F., Chintalapudi, K., Gnawali, O., Rangwala, S., Gummadi, R., Stathopoulos, T.: Tenet: An architecture for tiered embedded networks. Technical Report CENS-TR-56, UCLA CENS (2005)
23. Liu, J., Cheong, E., Zhao, F.: Semantics-based optimization across uncoordinated tasks in networked embedded systems. Technical Report MSR-TR-2005-46, Microsoft Research, One Microsoft Way, Redmond, WA 98075 (2005)
24. Gibbons, P., Carp, B., Ke, Y., Nath, S., Seshan, S.: Irisnet: An architecture for a worldwide sensor web. IEEE Pervasive Computing (2003) 22–33
25. Madden, S., Franklin, M., Hellerstein, J., 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) 491 – 502