Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks

CHIEN-LIANG FOK, GRUIA-CATALIN ROMAN, and CHENYANG LU
Washington University in St. Louis

This article presents Agilla, a mobile agent middleware designed to support self-adaptive applications in wireless sensor networks. Agilla provides a programming model in which applications consist of evolving communities of agents that share a wireless sensor network. Coordination among the agents and access to physical resources are supported by a tuple space abstraction. Agents can dynamically enter and exit a network and can autonomously clone and migrate themselves in response to environmental changes. Agilla's ability to support self-adaptive applications in wireless sensor networks has been demonstrated in the context of several applications, including fire detection and tracking, monitoring cargo containers, and robot navigation. Agilla, the first mobile agent system to operate in resource-constrained wireless sensor platforms, was implemented on top of TinyOS. Agilla's feasibility and efficiency was demonstrated by experimental evaluation on two physical testbeds consisting of Mica2 and TelosB nodes.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Design, Reliability, Algorithms

Additional Key Words and Phrases: Agent, mobile agent, wireless sensor network, middleware

ACM Reference Format:

Fok, C.-L., Roman, G.-C., and Lu, C. 2009. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. ACM Trans. Autonom. Adapt. Syst. 4, 3, Article 16 (July 2009), 26 pages. DOI = 10.1145/1552297.1552299 http://doi.acm.org/10.1145/1552297.1552299

1. INTRODUCTION

Because Wireless Sensor Networks (WSNs) are embedded and exposed to a dynamic world, many of its applications must adapt to changes in the physical environment or the user. For example, a WSN that tracks a wildfire must adapt to changes in the wildfire's position, and a WSN that provides a robot with extended situational awareness in its vicinity must adapt to the movements

This research is supported by the ONR under MURI research contract N00014-02-1-0715 and by the NSF under contract CNS-0520220.

Authors' address: C.-L. Fok, G.-C. Roman, C. Lu, Department of Computer Science and Engineering, Washington University, Saint Louis, MO 63130; email: {liang, roman, lu}@cse.wustl.edu.

DOI 10.1145/1552297.1552299 http://doi.acm.org/10.1145/1552297.1552299

of the robot. New programming models and middleware are needed to support application self-adaptability within WSNs.

To address this need, we developed *Agilla*, a middleware specifically designed to support self-adaptive applications on WSNs. Agilla structures an application in terms of one or more *mobile agents*, which are special processes that can explicitly migrate or clone from node to node while maintaining their state. To ensure that agents remain autonomous while enabling interagent coordination, Agilla provides localized tuple spaces that are remotely accessible. This allows agents to freely migrate while remaining able to coordinate with other agents.

Agilla's unique programming model enables higher degrees of application self-adaptability. By facilitating agent migration, an application can restrict itself to reside only on relevant nodes. As the environment changes, the application can self-adapt by migrating its agents to positions that best fulfill its goals. Moreover, since tuple spaces facilitate spatiotemporal decoupling between agents, each agent can be replaced without affecting the other agents in the network. This allows the set of mobile agents belonging to an application to evolve, enabling the application to adapt to changing requirements.

This article presents the Agilla programming model and the various engineering decisions that tailor Agilla to the WSN environment. Specifically, it makes the following primary contributions.

—We introduce for the first time mobile agents and tuple spaces as fundamental programming models for self-adaptive applications in WSNs.
—We present the design and implementation of Agilla, the first middleware system that supports mobile agents and interagent coordination in highly resource-constrained wireless sensor platforms. The Agilla middleware consumes only 57KB of code and 3.3KB of data memory on Mica2 platforms, and 45KB of code and 3.4KB of data memory on TelosB platforms.
—We evaluate the efficiency of Agilla via experiments on two physical WSN testbeds. The results show that Agilla agents can reliably migrate one hop every 0.3 seconds and perform remote tuple space operations within 55ms on the Mica2 platform.
—We demonstrate the generality and efficacy of the Agilla programming model for self-adaptive applications via three application case studies: wildfire tracking, cargo tracking, and robot navigation.

The remainder of the article is organized as follows. Section 2 discusses related work. Section 3 presents Agilla's model and explains how it is tailored to WSNs. Section 4 discusses the various engineering trade-offs necessary to cope with limited resources and an unreliable network. Section 5 presents the experimental results on Agilla's performance in terms of micro- and macrobenchmarks. Section 6 contains example applications that illustrate how Agilla can be used to simplify programming and increase network flexibility. The article ends with conclusions in Section 7.

2. RELATED WORK

A traditional approach for WSN adaption is to reprogram it over the wireless network. Systems that enable this can be divided based on what is reprogramed, that is, native code, interpreted code, or both. Two systems that reprogram native code are Deluge [Hui and Culler 2004] and MOAP [Stathopoulos et al. 2003]. They are designed to transfer large program binaries, enable the network to be arbitrarily reprogrammed, but incur high overhead and latency. To address this, SOS [Han et al. 2005], Contiki [Dunkels et al. 2004], and Impala [Liu and Martonosi 2003] are systems that enable partial reprogramming of binary code by providing a micro-kernel that supports dynamically linked modules. Since modules are relatively small, the cost of reprogramming is lower. Other systems that reprogram native code limit overhead by only sending the changes as determined by the diff [Reijers and Langendoen 2003] and rsync [Jeong 2005] algorithms.

Systems that reprogram interpreted code include Maté [Levis and Culler 2002], Application-Specific Virtual Machines (ASVM) [Levis et al. 2005], Melete [Yu et al. 2006], and SensorWare [Boulis et al. 2003]. In Maté and its successor ASVM, applications are divided into capsules that are flooded throughout the network. Each node stores the most recent version of a capsule and runs the application by interpreting the capsules using a Virtual Machine (VM). Since capsules are flooded throughout the network, Maté and ASVMs are installed network-wide. Melete improves upon Maté by enabling multiple applications to coexist within a sensor network. It does this by providing groups of capsules in which each group contains a different application. SensorWare allows users to dynamically inject mobile scripts into the network, enabling multiple applications to run concurrently. SensorWare scripts only provide weak mobility, that is, they must restart after each migration, and the system was implemented for the relatively powerful iPAQ 3670 platform.

Hybrid systems that reprogram both native and interpreted code include VM* [Koshy and Pandey 2005] and Dynamic Virtual Machine (DVM) [Balani et al. 2006]. Both systems allow the instruction set to change, but do not focus on how applications adapt.

The aforementioned reprogramming systems share a common feature: The decision on when and where to reprogram the network is determined centrally at a base station, often by a human operator. In contrast, Agilla provides a fundamentally different programming model based on mobile agents and tuple spaces that are especially well suited for self-adaptive applications in WSNs. Mobile agents can make adaptation decisions locally and autonomously *within* the network via migration (i.e., moving and cloning). Since network nodes are directly exposed to the environment, they can more quickly detect changes and better determine when software adaptation is necessary.

Mobile agents have been used in the Internet [Acharya et al. 1997; Baumann et al. 2002; Baldi and Picco 1998; Cabri et al. 2000; Cugola and Picco 2001; Gray 1997; Johansen et al. 1995; P.E.Clements et al. 1997; Peine and Stolpmann 1997] and their potential benefits are well established [Lange and Oshima 1999; Maes et al. 1999; Wu et al. 2004; Qi et al. 2001a; Tseng et al. 2004b; Tynan et al.

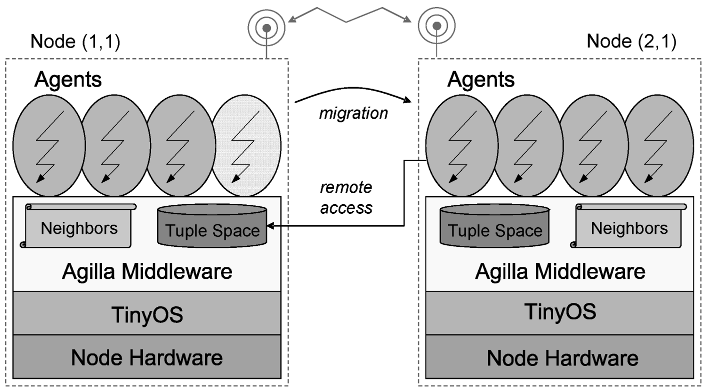

Fig. 1. The Agilla model.

2005; Wooldridge and Jennings 1995]. Since these systems are designed to run on Internet servers, efficient resource utilization is not their main focus. Mobile agents have also been used in wireless ad hoc networks. Systems that provide this include LIME [Murphy et al. 2006], Limone [Fok et al. 2004], and Smart Messages [Kang et al. 2004]. All three were designed for relatively resource-rich devices and are thus not appropriate for WSNs. For example, LIME supports tuple spaces that span multiple hosts incurring transactional costs [Carbunar et al. 2004], while Limone allocates a tuple space for each agent, rendering the cost of migration untenable. Smart Messages only supports a single thread of execution per node, meaning it cannot support multiple applications.

Mobile agents have previously been considered for use in wireless sensor networks. For example, mobile agents can perform certain operations like data integration [Qi et al. 2003, 2001a, 2001b] and tracking [Tseng et al. 2004a] better than traditional client/server mechanisms, and can be made energy efficient [Marsh et al. 2005; Tong et al. 2003]. While these efforts promote the use of mobile agents in wireless sensor networks, they were not actually deployed or evaluated in a real network. Instead, they were evaluated theoretically, in simulation, or using networks consisting of relatively resource-rich devices.

Agilla is the first system to bring the mobile agent programming model into a real wireless sensor network. By integrating the mobile agent and tuple space programming models, Agilla enables applications to be locally and autonomously self-adaptive. The Agilla programming model and middleware architecture meet the challenges unique to wireless sensor networks, for example, severe resource constraints and unreliable wireless connectivity. The novel specialization of the mobile agent and tuple space programming models combined with a careful engineering effort resulted in the working Agilla middleware system described in this article.

3. MODEL

Agilla's model, shown in Figure 1, is designed to facilitate adaptive behavior within a WSN. Each node supports multiple autonomous *mobile agents* that can move or clone across nodes while carrying their state. Mobile agent interactions are facilitated by two basic data abstractions on each node: a *neighbor list* and a *tuple space*. Agilla provides support for both *local* and *remote* tuple space operations. Furthermore, it provides specialized *reaction* primitives that enable agents to efficiently respond to changing state. Since the spatial orientations of WSN nodes are significant, Agilla addresses them by their *location*, which allows applications to focus on their objective (e.g., sense a phenomena at a particular location). Each of these features are described later in this section.

Agilla's model is designed for *localized* adaptation like fire tracking. It is *not* meant for data collection applications that require deployment across the entire WSN. Agilla is especially suitable for handling situations in which local decisions would significantly reduce the amount of data wirelessly transmitted. For example, in a fire tracking application, allowing mobile agents to autonomously and locally adapt to the changing location of the fire prevents having to coordinate the application from a base station. At the same time, it prevents the application from needing to be deployed on every node in the network.

3.1 Mobile Agents

Mobile agents are special processes that can autonomously migrate across nodes. Agilla provides two forms of migration, strong and weak, to support diverse application needs for self-adaptation. Strong migration transfers both the code and state, allowing the agent to resume execution at the destination. It is useful for performing computations that span multiple nodes. Weak migration only migrates the code. It exhibits less overhead since the state does not need to be transferred, but resets the agent.

When an agent migrates, it can either clone or move. If an agent is cloned, a copy of it arrives and starts executing at the destination while the original one resumes on the original node. If an agent is moved, it will no longer exist on the original node after it arrives at the destination.

An agent's life cycle begins when it is either injected into the network from a base station or cloned from another agent already in the network. Each agent executes autonomously performing application-specific tasks, and multiple agents may reside on the same node. When an agent completes its tasks, it dies freeing the computational resources it used.

3.2 Tuple Space

Agilla provides a *tuple space* [Gelernter 1985] on each node, as shown in Figure 1. A tuple space is a type of shared memory in which data is structured as tuples that are accessed via pattern-matching. This enables a decoupled style of communication in which the sender and receiver need not agree on a shared memory address, or even coexist, for communication to occur. Using tuple spaces, agents can function autonomously and migrate freely while still being able to communicate. It is further motivated by the fact that WSN applications continuously evolve and the agents may change over time, meaning a particular agent may not know with which other agents it must communicate.

A tuple space in Agilla does not span multiple nodes to avoid the overhead of keeping it consistent in a dynamic environment and to ensure scalability. If a shared data abstraction spanning multiple nodes is necessary, it must be built on top of Agilla's primitives. Although a tuple space is contained on a single node, it can be accessed by agents locally and remotely, and is augmented with reactions that enable agents to efficiently respond to changes in the tuple space state. Agilla tuple spaces also provide a convenient way for agents to discover properties of their environment. This is necessary because agents move and will encounter unfamiliar environments.

Agilla tuple spaces are accessible to agents residing on the same node via *local operations*, and to agents residing on different nodes using *remote operations*.

Local operations. An agent can save a tuple in the tuple space using operation `out`, and can either remove a tuple using operation `in` or read a tuple using operation `rd` (tuple space operations are named relative to the agent). The last two operations are blocking; they will wait until a matching tuple appears before allowing the agent to continue executing. Sometimes an agent may want to simply check whether a certain tuple exists and not block. To enable this, Agilla also provides *probing* operations `inp` (probing `in`), and `rdp` (probing `rd`). These operations are identical to `in` and `rd` except instead of blocking, they return `null` if a matching tuple does not exist.

All tuple space operations are performed atomically. This is feasible because a tuple space resides on a single node. A queue is used to serialize the operations, ensuring atomicity. If an operation blocks, it is placed in a separate wait queue to allow other operations to execute. When a tuple is inserted into the tuple space, the operations in the wait queue are placed in the regular queue, enabling them to search for a match. Thus, blocking operations are atomically executed at the time a matching tuple is found.

Remote operations. Agilla provides *remote tuple space operations* to enable agents residing on different nodes to communicate. They include remote versions of all nonblocking local operations, plus two special group operations, `routg` (remote group `out`) and `rrdpg` (remote probing group `rd`), that operate on all neighboring nodes. Remote operations are nonblocking to prevent an agent from deadlocking due to message loss.

Most remote tuple space operations rely on unicast communication. These operations are highly efficient since they only entail one network round-trip, a request and a reply. If an underlying multihop networking service is available, these operations may be performed on tuple spaces residing on nodes multiple hops away. The only exceptions are the group operations, which use single-hop broadcast. This is feasible since wireless is a broadcast medium. Group operations are performed on a best-effort basis.

Reactions. Reactions provide interrupt semantics and consist of a template and a *call-back function*, which is executed when the reaction *fires*. Prior to firing, a reaction must first be *activated* by a matching tuple in the local tuple space. When a reaction fires, a copy of the matching tuple is given to the agent, and the agent executes the reaction's call-back function. Reactions allow an agent to indicate its interest in tuples that match a particular template. They persist across agent migrations, but do not maintain history across migrations. Thus, if an agent migrates away and then back, it will rereact to all matching tuples in the local tuple space.

Agilla reactions can only react to tuples in the *local* tuple space. In addition, to conserve memory, the call-back functions are *not* executed atomically relative to activation. In Agilla, reactions *eventually* fire as long as the matching tuples remain within a tuple space. While it is possible for a tuple to be inserted and removed without a reaction firing, adopting these weaker semantics reduces middleware complexity and overhead, which is necessary given the limited resources on a WSN node.

3.3 Location-Based Addressing

The spatial orientation of WSN nodes is important because they are embedded within the environment. To capture this, Agilla addresses nodes by their location. Thus, instead of performing a `rout` on node 1, an agent performs it on a node at location (x, y). This simplifies programming by allowing the developer to focus on what an agent does (e.g., measure the temperature at location (x, y)) rather than how it does it (e.g., perform a search for the node at location (x, y), then measure the temperature at that node). The location coordinate system must ensure that every node has a unique address. If there is no node present at a particular location, an acceptable error bound must be specified. Agilla primitives can be generalized to enable operations on a region. For example, a fire detection node may need to clone itself onto *all* or *at least one* node in a particular geographic area.

The determination and selection of a destination location is application specific and must be specified by the application developer. For example, in an application that detects forest fires, the agents may perform a random walk. In another application involving a robot using mobile agents to discover its surroundings, the mobile agents may be restricted to the region surrounding the robot. The decision on how agents migrate impacts performance and should be a design criterion when developing an application. As a generic framework for developing WSN applications, Agilla does not control application-specific behavior like agent migration patterns.

The use of location-based addressing does not conflict with data-centric WSN applications [Intanagonwiwat et al. 2003]. Data-centric applications route based on content. The same behavior can be achieved using agents and tuple spaces. Specifically, agents can be programmed to analyze a tuple and send the tuple to a particular neighbor based on its content. Agilla uses location-based addressing because agents control their own movements and the most natural and meaningful way to specify a destination is by location.

3.4 Example

Figure 2 shows a portion of a FireTracker agent that notifies it of a fire's location. To reduce code size, Agilla agents utilize a stack architecture. Thus, the various push instructions shown on lines 1–4 place items on the stack. The

```
1: BEGIN  pushn fir
2:        pusht LOCATION
3:        pushc 2     // tuple <'fir', type:location> on stack
4:        pushc FIRE  // push reaction callback address
5:        regrxn      // register fire alert reaction
6:        wait        // wait for reaction to fire
7: FIRE   pop
8:        sclone      // strong clone to the node that detected the fire
9:        ...         // fire tracking code
```

Fig. 2. A portion of the `FireTracker` agent.

agent first registers a reaction sensitive to tuples containing the string "fir" and a location, and waits for the reaction to fire (lines 1–6). When a fire is detected, a matching tuple is inserted into the tuple space, causing the agent to execute the code beginning on line 7. Using the location stored within the tuple, the agent clones itself onto the node that detected the fire (line 8), and proceeds to form a dynamic perimeter around the fire. Note that FireTracker agents only reside on the nodes that are within the fire's vicinity, minimizing resource utilization. This differs from traditional deployments in which the application is installed onto every node in the network.

While Agilla agents are written in an assembly-like language, developing a higher-level language is a relatively straightforward extension of existing WSN scripting languages like TinyScript [Levis 2004]. For example, TinyScript supports functions, which is a convenient way to access Agilla operations. Thus, instead of writing the instructions shown on lines 1–5 of Figure 2 that register a reaction, a high-level language modeled after TinyScript could execute a function like `registerReaction(FIRE, ⟨"fir", TYPE:LOCATION⟩)`.

3.5 Scalability

Wireless sensor networks have the potential to grow in size [Murty et al. 2007], making scalability an important attribute to consider. In fact, one reason Agilla maintains a separate tuple space on each node is to ensure scalability. While scalability ultimately depends on the way an application is designed, Agilla encourages scalable designs by promoting localized interactions between agents via single-hop neighbor lists on each node. Using this neighbor list and per-node tuple spaces, application algorithms must be designed to only use local knowledge and interactions, thus ensuring scalability. For example, in the fire tracking application, each FireTracker agent only needs to check its clockwise and counterclockwise directions to form a perimeter around the fire. Complex application-wide behaviors like dynamic perimeter formation emerge through these simple local decisions.

3.6 Adaptation to Node Failures

Node failures are a frequent occurrence in wireless sensor networks. When a node fails, all tuples and mobile agents residing on that node are permanently lost. While this may cause application failure, Agilla provides the capability for applications to self-heal. Specifically, an application can self-heal by cloning or moving its agents onto the replacement node when it is installed. Unlike other in-network reprogramming systems, Agilla gives application developers control over the self-healing process. For example, in the fire tracking application, a node will fail when it catches on fire. The FireTracker agent self-heals by detecting this failure and cloning itself around the failed node to ensure the integrity of the perimeter. Note that in other systems like Trickle [Levis et al. 2004] and Deluge [Hui and Culler 2004], the developer has less control since the application is always flooded throughout the entire network.

3.7 Security

Security is important to some applications of WSNs [Perrig et al. 2004]. Mobile agents can introduce security risks due to their mobility. Existing techniques can be used to achieve various levels of security in Agilla. For example, wireless transmissions can be encrypted [Karlof et al. 2004], and each agent can be enclosed within a "sandbox" [Platon and Sei 2008]. More advanced techniques that are particularly useful in validating untrusted mobile code include code verification [Choi et al. 2007] and proof-carrying code [Necula 1997]. The current implementation of Agilla does not integrate all of these mechanisms, but they can be integrated to achieve the desired level of security.

4. MIDDLEWARE DESIGN AND IMPLEMENTATION

Agilla's middleware is designed to meet the unique challenges of WSNs like severe resource constraints and unreliable wireless networks. This section first presents the target platform, then describes Agilla's middleware architecture followed by its instruction set architecture.

4.1 Target Platform

Agilla was initially implemented on Mica2 nodes [Crossbow Technology 2005a], which have an 8-bit 7.38MHz Atmel ATmega128L microprocessor and a Chipcon CC1000 radio transceiver. The radio communicates at up to 38Kbps over a range of up to 100m [Zhao and Govindan 2003]. The nodes have only 128KB of ROM and 4KB of RAM. Since its initial implementation, Agilla has been ported to the MicaZ [Crossbow Technology 2005b], Tyndall 25mm [Tyndall National Institute 2005], and TelosB [Polastre et al. 2005] nodes, which have similar characteristics to the Mica2.

The severe resource constraints of WSN nodes require that Agilla be carefully engineered. While the operating system does not affect Agilla's basic programming model, it does affect the implementation. Agilla can be implemented on many different operating systems. For this study, TinyOS [Hill et al. 2000] is used. TinyOS does not support binary module updates like Contiki [Dunkels et al. 2004] or SOS [Balani et al. 2006]. However, this ability is complementary to Agilla and may enhance Agilla's flexibility. Through the careful engineering described in this section, Agilla was able to fit in the limited resources available, as shown in Table I.

Table I. Memory Availability and Size of Agilla

	Available		Agilla	
Platform	ROM	RAM	ROM	RAM
Mica2	128K	4K	57K	3.3K
TelosB	48K	10K	45K	3.4K

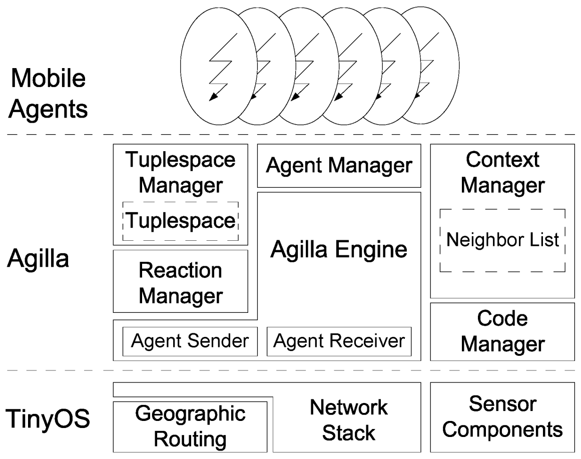

Fig. 3. Agilla's middleware architecture.

4.2 Middleware Architecture

Agilla's software architecture is divided into three layers, as shown in Figure 3. The highest contains the agents and is discussed further in Section 4.3. The middle contains the core Agilla middleware components, while the bottom is the operating system.

The middleware consists of several components that are orchestrated by an Agilla engine, which is the Virtual Machine (VM) kernel that controls the concurrent execution of all agents on a node. It implements a round-robin scheduling policy. If an agent executes a long-running instruction like `sleep`, `sense`, or `wait`, the engine immediately switches agents.[1] The VM's scheduler is implemented as a TinyOS task that is continuously posted. Each time the task is posted, it executes one instruction. By switching agents in a round-robin fashion, the effect is concurrent execution of all agents. Note that this design is due to TinyOS's execution model, which is based on tasks. If Agilla were implemented on an operating system supporting threads like Contiki or SOS, the VM scheduler could be replaced by multiple concurrent threads, each executing a different agent.

The Agilla engine also handles agent arrival and departure. When an agent migrates, Agilla divides it into multiple types of messages, as shown in Figure 4.

[1]A large body of work exists on scheduling policies that provide real-time guarantees or multiple priority levels [Stallings 2001]. These advanced scheduling policies may be incorporated into the Agilla engine.

Type	**Size (Bytes)**	**Content**
State	16	agent id, program counter, code size, condition code, stack pointer
Code	26	one instruction block
Heap	26	four variables and their addresses
Stack	26	four variables
Reaction	26	one reaction

Fig. 4. Messages used during migration.

The message sizes are based on restrictions imposed by TinyOS's network stack and do not limit the size of an agent. Multiple messages will be used if necessary. For example, if an agent has 30 bytes of code, two code messages will be used.

The highly unreliable nature of WSN wireless links may prevent agents from reliably migrating. This is a problem since a WSN application may fail or perform poorly if it looses an agent. To address this problem, an agent is migrated one hop at a time, and is acknowledged after each hop. This technique is used by other transport protocols within WSNs to increase reliability [Wan et al. 2002], but they focus on packet communication in general while Agilla applies the approach to agent migration in particular.

When an agent fails to migrate, it resumes on the sender node with an error flag set. The agent can then perform an application-specific response. While this may result in duplicate agents when the migration was actually successful, the possibility of duplicate agents is preferable to losing an agent. Consider the fire detection application; it is better to have duplicate fire warnings rather than no warnings at all. Of course, having duplicate agents wastes resources. To address this, agents can be programmed to detect when a certain number of identical agents are present, and terminate when this threshold is reached. Note that when an agent attempts to migrate a long distance, the agent may be resumed in the middle due to migration failure. The programmer must determine what to do in this situation.

The Agilla engine is supported by several components including the agent, code, context, tuple space, and reaction managers. Each manager provides a unique service that collectively forms Agilla's middleware. The agent manager maintains agent execution state. The code manager dynamically allocates code memory for each agent, and fetches instructions as the agent executes. The context manager keeps an updated list of neighboring nodes. The tuple space manager maintains the local tuple space and implements the nonblocking operations (both local and remote). Finally, the reaction manager stores the reactions registered by the local agents and determines when a reaction should fire.

Agilla's middleware is highly optimized to reduce memory consumption. Optimized components include the context, tuple space, and reaction managers. The context manager uses a simple beaconing mechanism for neighbor discovery. Provided enough resources and application requirements, more advanced protocols may be implemented [Whitehouse et al. 2004; Welsh and Mainland 2004; Woo et al. 2003]. The tuple space manager does not implement blocking

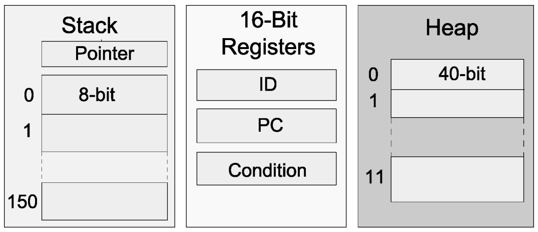

Fig. 5. The mobile agent architecture.

operations, which are implemented within the instructions themselves as described in Section 4.4. The reaction manager operates asynchronously with the tuple space manager. This simplification reduces memory consumption, but weakens the semantics of reactions slightly by only ensuring that a reaction will *eventually* fire, provided the matching tuple remains in the tuple space.

4.3 Agent

The architecture of an Agilla mobile agent is shown in Figure 5. It consists of a stack, heap, and various registers. A stack architecture is used to enable higher code density. Most Agilla instructions are a single byte. By default, the operand stack is 105 bytes, which is small enough to fit on the Mica2 platform. The heap is a random-access storage area that allows each agent to store 12 variables. It is accessed by instructions `getvar` and `setvar`. The operand stack and heap sizes are customizable based on memory availability.

The agent also contains three 16-bit registers: the agent's unique ID, the Program Counter (PC), and the condition code. The agent ID is unique and is maintained across move operations. A cloned agent is assigned a new ID. An agent ID is generated by concatenating the least significant byte of the host address with a monotonically increasing counter on the host.[2] The PC is the address of the next instruction. Finally, the condition code is a 16-bit register that records execution status. In addition to recording the results of comparison instructions, this register records whether a migration operation is successful, whether the agent is the original or clone, and if it failed, why.

4.4 Instruction Set Architecture (ISA)

Agilla's ISA is tailored to the unique properties of self-adaptive WSN applications (e.g., localized interactions), and to the mobile agent computing model (e.g., agent migration and the need for context information). Some of Agilla's unique instructions that achieve this are presented in Figure 6. A full listing is available on Agilla's Web site [Fok 2005]. Agilla's ISA can be divided into four

[2]Our current implementation assumes a maximum of 256 nodes and that each node may clone up to 256 agents. Future implementations that require higher limits may increase the size of the AgentID, or implement a mechanism that recycles old agent IDs belonging to agents that have have already died. To prevent duplicate IDs due to node failure, the counter can be stored in flash memory and restored when the node reboots.

Instruction	Description
`loc`	Pushes the current location onto the stack
`wait`	Stops agent execution, allows it to wait for a reaction
`smove`	Strong move
`wclone`	Weak clone
`getnbr`	Get a specific neighbor's address
`randnbr`	Get a random neighbor's address
`out`	Insert a tuple into the local tuple space
`inp`	Non-blocking find and remove tuple from the local tuple space
`rd`	Blocking find tuple in the local tuple space
`rout`	Insert a tuple into a remote tuple space
`rinp`	Non-blocking find and remove tuple from a remote tuple space
`regrxn`	Register a reaction on the local tuple space

Fig. 6. Noteworthy Agilla instructions.

categories: general purpose, extended, tuple space, and migration. General-purpose instructions enable agents to perform basic tasks like obtaining the neighbor list, sensing, periodically sleeping to conserve energy, and continuously repeating certain application-specific operations. Extended instructions are application specific. They enable an agent's efficiency to be significantly increased by changing the virtual-native code boundary [Levis et al. 2005]. For example, in the fire tracking agent, a helpful instruction is one that determines which neighbors are on fire. The remainder of this section discusses the tuple space and migration instructions.

Tuple space instructions. Tuple space operations allow an agent to interact with the tuple space on each host. These operations require a tuple or template parameter. This is done by pushing each field followed by the number of fields onto the stack. For example, in Figure 2, lines 1–3 push a template with two fields onto the stack.

Instructions `out`, `in`, `rd`, `inp`, and `rdp` access the local tuple space. The blocking `in` and `rd` operations are implemented by having the agent repeatedly try the probing `inp` or `rdp` operations. If no matching tuple is found, the agent is stored in a wait queue. When a tuple is inserted, the agents in this queue are notified and recheck for a match. The remote tuple space operations `rout` (remote `out`), `rinp` (remote probing `in`), `rrdp` (remote probing `rd`), `routg` (remote group out), and `rrdpg` (remote probing group `rd`) are nonblocking to avoid deadlock due to message loss and disconnection. The group operations are done in a best-effort basis using wireless broadcast. Applications must be engineered with these best-effort semantics in mind. `rrdpg` relies on a time-out to determine when to stop waiting for replies. The tuple space instructions also include `regrxn` (register reaction) and `deregrxn` (deregister reaction).

Migration instructions. Migration instructions allow an agent to move or clone to another node. Agilla provides four migration instructions: `smove` (strong move), `wmove` (weak move), `sclone` (strong clone), and `wclone` (weak clone). Strong operations transfer both the state and code, while weak operations only transfer the code. As mentioned in Section 3.1, strong migrations are more powerful since they allow an agent to resume where it left off prior to migrating, but

```
// The smove agent
1:     pushloc 5 1
2      smove            // move to node (5,1)
3:     pushloc 1 1
4:     smove            // move to node (1,1)
5:     halt

// The rout agent
1:     pushc 1
2      pushc 1          // tuple <value:1> on stack
3:     pushloc 5 1
4:     rout             // do rout on node (5,1)
5:     halt
```

Fig. 7. The agents that test `smove` (top) and `rout` (bottom).

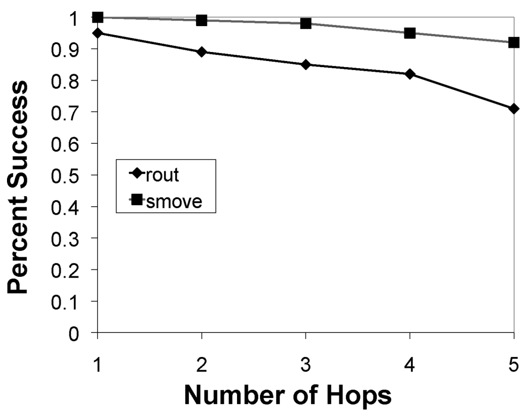

Fig. 8. `smove` vs. `rout` reliability.

incur higher overhead. Weak migrations are more efficient, but force agents to resume from the beginning upon arrival at the destination.

5. EMPIRICAL EVALUATION

This section presents micro-benchmarks demonstrating feasibility, followed by an application case study involving fire detection and tracking.

5.1 Micro-Benchmarks

The following micro-benchmarks are performed on a network of 25 Mica2 nodes arranged in a 5x5 grid. To achieve a multihop network, messages are filtered based on the grid topology. Geographic routing [Kim et al. 2005] is used for remote tuple space operations and agent migrations.

To test migration and tuple space operations, the agents shown in Figure 7 are used. The strong move (`smove`) agent moves to a remote node and back while the remote out (`rout`) agent places a tuple in a remote node's tuple space. The distance between the original and destination nodes is varied from 1 to 5 hops and each experiment is repeated 100 times. The latency of each successful execution and the number of failures are recorded. The results, shown in Figures 8

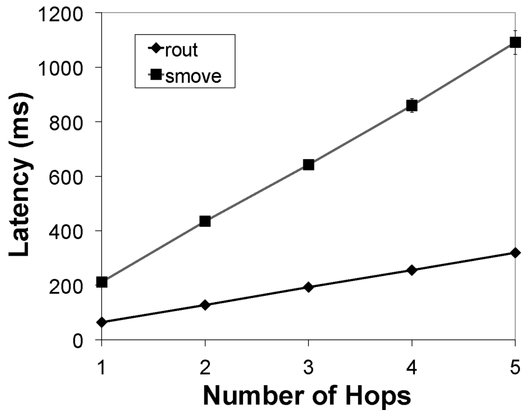

Fig. 9. `smove` vs. `rout` latency.

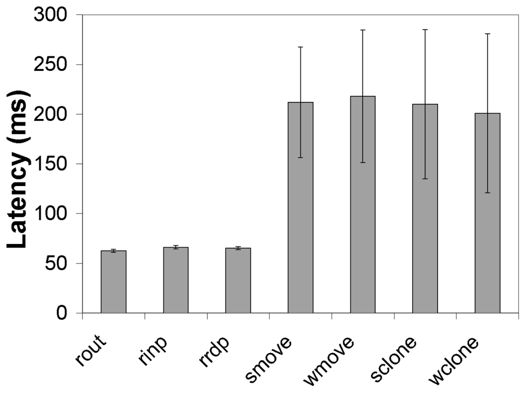

Fig. 10. Remote operation lantency.

and 9, indicate that as the distance increases, the probability of message loss also increases, which is reflected in a decrease in reliability. The reason `rout` is less reliable than `smove` is because, in these experiments, `rout` does not use ARQs for retransmitting lost packets. Note that confidence intervals are not given in Figure 8 since it shows the percent success across all experimental rounds. Figure 9 contains 95% confidence intervals calculated over the successful rounds (the error margins are difficult to see because they are so small, i.e., ±3.09ms for `rout` and ±43.79ms for `smove`).

The one-hop latencies of all remote operations are measured by timing each 100 times and finding the average. The results, shown in Figure 10, are similar to those for `rout` and `smove` and show that the agent migration instructions have higher overhead than the remote tuple space operations. The figure also shows that the strong and weak migration operations have approximately the same latency. This is because the additional overhead of transmitting a few more packets containing the agent's state is small relative to the cost of performing hop-by-hop migration. Note that the migration latencies have higher variance due to the use of ARQ. The results suggest that an agent can reliably migrate

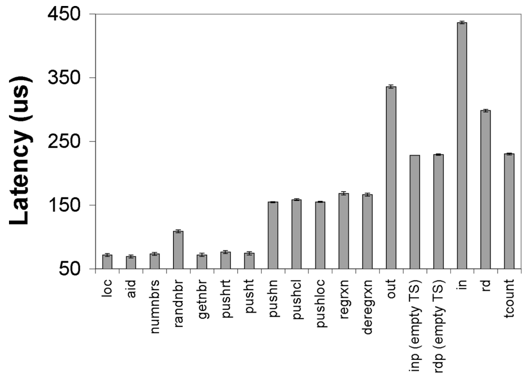

Fig. 11. Local operation latency.

one hop every 0.3s on the Mica2 platform. Assuming the radio range is 50m, this means an agent can migrate across a network at 600km/h (373mph), which is sufficient for tracking many interesting phenomena like fire.

Local operations unique to Agilla are also benchmarked. Each local instruction is benchmarked 100 times, and the average execution time is calculated. The results, shown in Figure 11, indicate that local operations execute quickly relative to remote operations. We did not directly compare Agilla's instructions with other WSN VMs like Maté [Levis and Culler 2002] because many of Agilla's instructions are higher level and do not have a corresponding instruction against which to compare. However, the latency of simpler Agilla instructions that execute within 100μs like get location (`loc`) and get agent ID (`aid`) are comparable to corresponding Maté operations [Levis and Culler 2002].

In summary, Agilla can perform one-hop remote tuple space operations within 55ms, and migration operations in 225ms per hop on Mica2 nodes. The migration latency scales linearly with the number of hops, and the additional overhead for reliable operations is justified by their resilience to message loss across multiple hops. This demonstrates the feasibility and efficiency of using mobile agents and tuple spaces in a WSN.

5.2 Macro-Benchmarks: Fire Detection and Tracking

This section evaluates how Agilla enables developers to create highly adaptive applications in a dynamic environment. A fire detection and tracking application is used, as shown in Figure 12. In this application, a WSN is deployed in a region susceptible to fires. `FireDetection` agents are deployed which patrol the network for fires. When they detect a fire, they notify a `FireTracker` agent, which moves to the fire and repeatedly clones itself to form a perimeter. The tracker agents autonomously adjust their numbers and locations to maintain a perimeter as the fire changes shape. The remainder of this section presents the implementation and evaluation of the fire detection and tracking application. The application consists of three types of agents: `Fire` agents that emulate fire, `FireDetection` agents, and `FireTracker` agents.

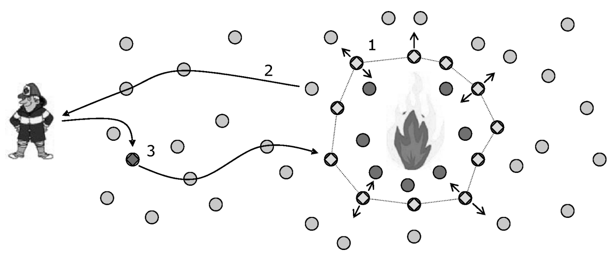

Fig. 12. An overview of the fire detection and tracking application. When a fire breaks out, detection agents sense the fire (1) and send a message to a base station (2), which injects a tracker agent into the network (3). This agent migrates to the fire and clones itself to form a perimeter. The perimeter is continuously adjusted based on the fire's behavior.

```
1:  BEGIN        pushn fir
2:               pushc 1
3:               out                 // insert fire tuple
4:  BLINK_RED    pushc 25
5:               putled              // toggle red LED
6:               pushc 1
7:               sleep               // sleep for 1/8 second
8:               rjump BLINK_RED
```

Fig. 13. The static `Fire` agent.

5.2.1 *Fire Agents.* `Fire` agents emulate fire by inserting the string "fir" into the local tuple space. Fire is detected by searching for these tuples. Two types of `Fire` agents are used: *static* and *dynamic*. A static `Fire` agent does not move and serves as a baseline on how quickly the `FireTracker` agent can form a perimeter around a fire. Its code is shown in Figure 13. Lines 1–3 insert the fire tuple, while lines 4–8 continuously blink the red LED. By monitoring the LEDs, the application's state is determined. Dynamic `Fire` agents model a fire that spreads. It is implemented in a mere 47 bytes of instructions and is available on Agilla's Web site [Fok 2005].

5.2.2 *Fire Detection Agents.* `FireDetector` agents search for fire tuples, and notify the `FireTracker` agent when a fire is found.[3] The agent's code is shown in Figure 14. It searches for a fire tuple once per second (lines 1–7), and inserts a tuple containing its location and the string "fir" onto the node hosting the `FireTracker` agent if a fire tuple is found (lines 8–14). After it notifies the base station, it dies freeing its resources (line 15).

5.2.3 *Tracking Agents.* `FireTracker` agents form and maintain a perimeter around the fire. They insert a tuple containing the string "trk" into the local tuple space. Each `FireTracker` agent periodically checks its clockwise and counterclockwise neighbors (relative to the position of the fire) for this tuple. If

[3]In a real deployment, fire can be detected using sensors that give more details about the fire. This would enable more sophisticated fire-tracking algorithms that consider the unique characteristics of the fire.

```
BEGIN     pushn fir
          pushc 1
          rdp                  // Search for fire tuple
          rjumpc FIRE          // jump if fire tuple found
SLEEP     pushc 8
          sleep                // sleep for 1 second
          rjump BEGIN          // repeat
FIRE      pop
          pop                  // pop off fire tuple
          loc
          pushn fir
          pushc 2              // tuple <"fir", location>
          pushloc 1 1          // assume fire tracker is at (1,1)
          rout                 // send tuple to fire tracker
          halt                 // die
```

Fig. 14. A `FireDetector` agent.

```
1:   REG_RXN     pushn fir
2:               pushc 1
3:               pushc  RXN_FIRED
4:               regrxn              // register the reaction
5:               ...                 // tracking code omitted
6:   RXN_FIRED   pushc 9
7:               putled              // turn off LEDs
8:               pushn trk
9:               pushc 1
10:              inp                 // remove tracker tuple
11:              halt                // die
```

Fig. 15. The reaction registered by the `FireTracker` agent.

they exist, the perimeter is considered intact. Otherwise, the agent clones itself onto the neighbor that is missing the tuple to reestablish the perimeter. This is repeated until the perimeter is fully formed. If the perimeter is breached, the `FireTracker` agents next to the breach will detect the missing neighbor and attempt to reform the perimeter via cloning. Note that perimeter formation is an emergent behavior based on local decisions, which ensures scalability. The actual code for creating and maintaining this perimeter is omitted due to space constraints and begins on line 5 of Figure 15. It consists of 58 lines of code, and is included with Agilla's distribution [Fok 2005].

If a node hosting a `FireTracker` agent catches on fire, the `FireTracker` agent must die. This is achieved by registering a reaction that kills the agent when a fire tuple appears. The code for this is shown in Figure 15. Lines 1–4 register a reaction sensitive to fire tuples, while lines 6–11 define the call-back function, which turns off the LEDs (lines 6–7), removes the tracker tuple (lines 8–10), and kills the agent (line 11).

The life cycle of a `FireTracker` agent is shown in Figure 16. It works by repeatedly checking whether the perimeter is breached, and cloning itself to recreate the perimeter if necessary. The periodic checking allows the perimeter to be adjusted as the fire spreads, thus achieving self-adaptation. The `FireTracker`

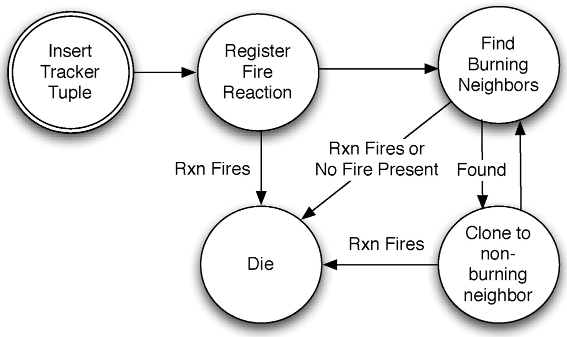

Fig. 16. The life cycle of a `FireTracker` agent.

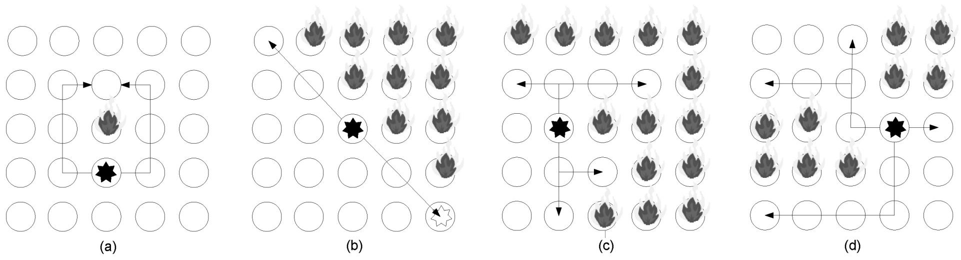

Fig. 17. The static fire test scenario, the star is the initial position of the `FireTracker` agent.

agent was implemented in only 101 bytes of code, demonstrating the expressiveness of the Agilla programming model.

The `FireTracker` agent was evaluated using both static and dynamic fires, using the same network as was used in the micro-benchmarks. For the static fire tests, static `Fire` agents are used to form fires of various shapes, and a `FireTracker` agent is injected onto a node next to the fire. The time to form a perimeter is recorded. The types of fires used are shown in Figure 17. The starting location of the `FireTracker` agent is marked with a black star, and the arrows indicate where the agent must clone to form the perimeter. Note that in scene B, node (5,1) in the lower-right corner also has a star. This is to evaluate how the starting location of the `FireTracker` agent impacts efficiency.

The results of the static fire tests are shown in Figure 18. In most cases the perimeter is formed within 3 seconds. Scene A took longer because its shape limits the number of `FireTracker` agents that can spread in parallel. For example, when a `FireTracker` agent is at node (2,2), which is in the lower-left corner, it is the only agent that can clone to (2,3). To test this, we reran scene B with the `FireTracker` agent initialized at node (5,1), which is in the lower-right corner. The results, shown in Figure 18, indicate that the initial point of `FireTracker` significantly impacts performance.

To evaluate Agilla's ability to maintain a perimeter around a spreading fire, four `FireDetector` agents are injected into the network at the positions marked with a star in Figure 19. A dynamic `Fire` agent is then injected into node (5,5). Note that `FireDetector` agents must first detect the fire and notify

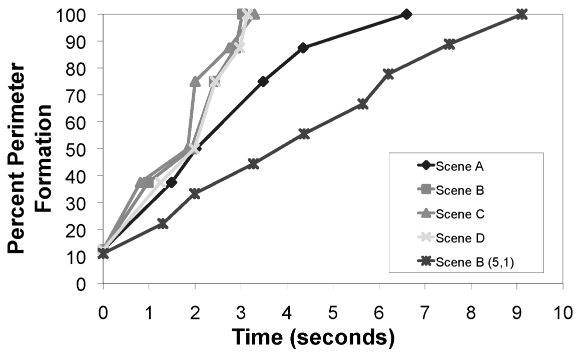

Fig. 18. The rate of forming a perimeter around static fires.

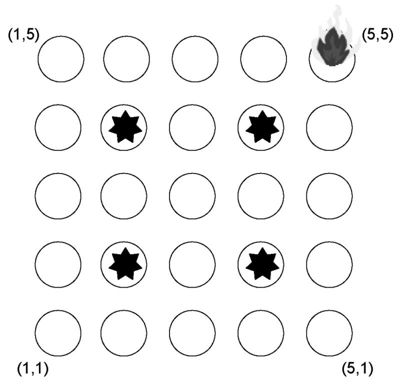

Fig. 19. Dynamic fire test settings.

the `FireTracker` agent before perimeter formation can begin. The `FireTracker` agent is initially at (1,1). Two tests are run: one with a slow `Fire` agent that spreads one hop every 7 seconds, another with a fast one that spreads every 5 seconds. The results, shown in Figure 20, indicate that the `FireTracker` agent does a reasonable job maintaining a perimeter around the slow fire, but has difficulty with the fast one due to the network being partitioned by the fire. The reason why both converge to 100% is because as the fire spreads, the network eventually becomes saturated. While it is true that in our limited network, every node will eventually be on fire, the `FireTracker` agents were able to at least partially form a perimeter long before it engulfs every node in the network.

This case study shows that Agilla provides a convenient programming model for implementing highly adaptive applications in a dynamic environment. With Agilla, we created a nontrivial self-adaptive application (fire-tracking) with minimal effort, while achieving sufficient application-level performance.

6. MORE APPLICATION EXAMPLES

To further demonstrate the efficacy of the Agilla programming model, we present the experiences of developing two other applications using Agilla. They

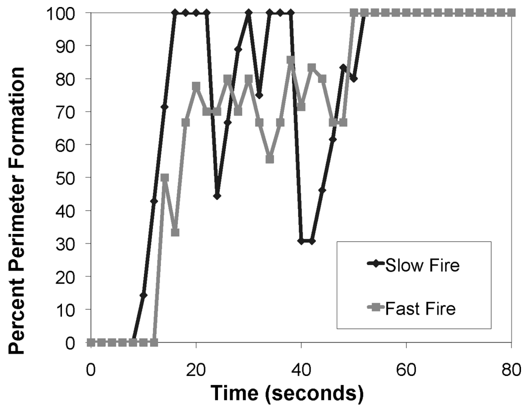

Fig. 20. Dynamic fire perimeter formation.

include cargo container monitoring and navigation in a dynamic environment. These projects were collaborative and are only outlined here. They are presented as case studies that further demonstrate Agilla's effectiveness in supporting self-adaptive applications.

6.1 Monitoring Cargo Containers

Container Security Devices (CSDs) can reduce the risk of importing cargo containers by continuously monitoring every container and forming a wireless ad hoc network for delivering security alerts. However, developing an application on top of CSDs is difficult due to the highly dynamic environment and the number of organizations involved. Agilla can facilitate CSD applications by injecting an agent that continuously monitors the sensors for anomalies. Alarms can be raised by inserting a tuple describing the event into the local tuple space. When the ship arrives at a port, Agilla agents belonging to the U.S. Customs and Border Patrol can search for these tuples to determine which containers need to be inspected.

Our group, along with corporate partners, deployed an Agilla network on a mock cargo container testbed. Each container is given a Mica2 node with a MTS310 sensor board for sensing light, temperature, and vibration, and a speaker for emitting an audible alert. Mobile agents load the manifest lists, detect security breaches, and query the manifest lists and security events. Tuples are encrypted, preventing eavesdropping and fake tuples. To prevent migration failures due to wireless contention, the mobile agents that canvas the entire network (e.g., search agents) clone in one direction, and move in a perpendicular direction.

Unlike RFIDs, CSDs can perform local computations. In this evaluation, security mobile agents detect impacts by monitoring the accelerometer, and intrusions by monitoring the light sensor. When an anomaly is detected, they save an alert tuple in the local tuple space. Later, CBP mobile agents search for the alert tuples to determine which containers need to be inspected.

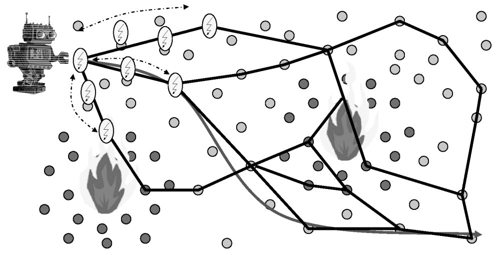

Fig. 21. The robot navigation problem. A roadmap graph is overlaid on the WSN and mobile agents are used to query the temperature along the edges, which helps the robot navigate around the fires.

The cargo shipping application has been implemented and demonstrated at SenSys 2005 [Hackmann et al. 2005]. It establishes the need for a self-adaptive software infrastructure and illustrates how Agilla meets this need.

6.2 Navigation in a Dynamic Environment

Consider a robot that needs to travel through a region while avoiding fires, as shown in Figure 21. Since the robot's on-board sensors have limited range, it uses the WSN and a navigation service [Bhattacharya et al. 2006] to find a safe route around the fire. To ensure efficiency, the possible routes are restricted to those on a roadmap [Bayazit et al. 2002], which is shown as an overlay graph in Figure 21. The problem is how to determine which paths in the roadmap are safe to traverse.

Assuming safe paths have maximum temperatures below a certain threshold, the robot deploys network exploration agents to determine the maximum temperature along a path. Once injected, the agent repeatedly clones itself onto nodes along the path, and sends back temperature information. To limit overhead, the agents spread themselves out and filter data coming from further down the edge by filtering all temperature readings less than the local temperature. Once the robot receives the maximum temperature along the edges of the overlay graph, it can decide which route to take.

The navigation service was implemented in only three days and evaluated using a testbed consisting of 17 Mica2 motes and an ActiveMedia Pioneer-3 DX robot [Bhattacharya et al. 2006]. Empirical results demonstrate the feasibility of using Agilla in this highly dynamic application. The implementation demonstrates how Agilla enables nontrivial in-network processing techniques to be implemented as mobile agents.

7. CONCLUSION

This article presents Agilla, a mobile agent middleware specifically designed for resource-constrained WSNs. It is the first working system to bring mobile agents and a tuple-space-based coordination model into WSNs. Agilla enhances network flexibility and supports adaptive applications through mobile agents that coordinate via localized tuple spaces. We have implemented Agilla on TinyOS and multiple WSN hardware platforms. Empirical results on

Mica2 and TelosB nodes demonstrate the feasibility and efficiency of Agilla on resource-constrained WSNs. Experiences with three adaptive applications demonstrate the expressiveness of Agilla's programming model.

ACKNOWLEDGMENT

We would like to thank colleagues at Boeing for collaborating on the cargo tracking application, and S. Bhattacharya, N. Atay, G. Alankus, and B. Bayazit for collaborating on the robot navigation application.

REFERENCES

ACHARYA, A., RANGANATHAN, M., AND SALTZ, J. 1997. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds. vol. 1222. Springer-Verlag, 111–130.

BALANI, R., HAN, C.-C., RENGASWAMY, R. K., TSIGKOGIANNIS, I., AND SRIVASTAVA, M. 2006. Multi-level software reconfiguration for sensor networks. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT'06)*. ACM, 112–121.

BALDI, M. AND PICCO, G. P. 1998. Evaluating the trade-offs of mobile code design paradigms in network management applications. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*. IEEE Computer Society, 146–155.

BAUMANN, J., ROTHERMEL, H. K., STRASSER, M., AND THEILMANN, W. 2002. Mole: A mobile agent system. *Softw. Pract. Exper. 32*, 6, 575–603.

BAYAZIT, O. B., LIEN, J.-M., AND AMATO, N. M. 2002. Roadmap-based flocking for complex environments. In *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications (PG'02)*. 104–121.

BHATTACHARYA, S., ATAY, N., ALANKUS, G., LU, C., BAYAZIT, O. B., AND ROMAN, G.-C. 2006. Roadmap query for sensor network assisted navigation in dynamic environments. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 17–36.

BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. B. 2003. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*. ACM, New York, 187–200.

CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. 2000. MARS: A programmable coordination architecture for mobile agents. *Internet Comput. 4*, 4, 26–35.

CARBUNAR, B., VALENTE, M. T., AND VITEK, J. 2004. Coordination and mobility in CoreLime. *Math. Structures Comput. Sci. 14*, 3, 397–419.

CHOI, Y.-G., KANG, J., AND NYANG, D. 2007. Proactive code verification protocol in wireless sensor network. Lecture Nodes in Computer Science, vol. 4706, 6, Springer, 1085–1096.

CROSSBOW TECHNOLOGY. 2005a. Mica2 wireless measurement system. http://www.xbow.com/ Products/productdetails.aspx?sid=174.

CROSSBOW TECHNOLOGY. 2005b. MicaZ wireless measurement system. http://www.xbow.com/ Products/productdetails.aspx?sid=164.

CUGOLA, G. AND PICCO, G. 2001. Peerware: Core middleware support for peer-to-peer and mobile systems. Tech. rep., Politecnico di Milano.

DUNKELS, A., GRONVALL, B., AND VOIGT, T. 2004. Contiki: A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*. IEEE Computer Society, 455–462.

FOK. C.-L. 2005. Agilla Website. http://mobilab.wustl.edu/projects/agilla.

FOK, C.-L., ROMAN, G.-C., AND HACKMANN, G. 2004. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages (Coordination'04)*, R. DeNicola et al., Eds. Lecture Notes in Computer Science, vol. 2949. Springer-Verlag, 135–151.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1, 80–112.

GRAY, R. 1997. Agent Tcl. *Dr. Dobb's J. Softw. Tools 22*, 3, 18–71.

Hackmann, G., Fok, C.-L., Roman, G.-C., Lu, C., Zuver, C., English, K., and Meier, J. 2005. Demo abstract: Agile cargo tracking using mobile agents. In *Proceedings of 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys'05)*, 303.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*. ACM, 163–176.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGPLAN Not. 35*, 11, 93–104.

Hui, J. W. and Culler, D. 2004. 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 (SenSys'04)*. ACM, 81–94.

Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., and Silva, F. 2003. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw. 11*, 1, 2–16.

Jeong, J. 2005. Incremental network programming for wireless sensors. M.S. thesis, Electrical Engineering and Computer Science Department, University of California, Berkeley.

Johansen, D., van Renesse, R., and Schneider, F. B. 1995. An introduction to the TACOMA distributed system. version 1.0. Tech. rep. 95-23, University of Tromsø, Tromsø, Norway.

Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., and Iftode, L. 2004. Smart messages: A distributed computing platform for networks of embedded systems. *The Comput. J.* (Special Issue on Mobile and Pervasive Computing). *47*, 475–494.

Karlof, C., Sastry, N., and Wagner, D. 2004. Tinysec: A link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM, 162–175.

Kim, Y.-J., Govindan, R., Karp, B., and Shenker, S. 2005. Geographic routing made practical. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*. USENIX Association, 217–230.

Koshy, J. and Pandey, R. 2005. VMSTAR: Synthesizing scalable run-time environments for sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys'05)*. ACM, 243–254.

Lange, D. B. and Oshima, M. 1999. Seven good reasons for mobile agents. *Comm. ACM 42*, 3, 88–89.

Levis, P. 2004. The tinyscript language.
http://www.cs.berkeley.edu/~pal/mate-web/files/ tinyscript-manual.pdf.

Levis, P. and Culler, D. 2002. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. ACM, 85–95.

Levis, P., Gay, D., and Culler, D. 2005. Active sensor networks. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI'05)*. USENIX Association, 343–356.

Levis, P., Patel, N., Culler, D., and Shenker, S. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, 2–2.

Liu, T. and Martonosi, M. 2003. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, 107–118.

Maes, P., Guttman, R. H., and Moukas, A. G. 1999. Agents that buy and sell. *Comm. ACM 42*, 3, 81–ff.

Marsh, D., O'Kane, D., and O'Hare, G. M. P. 2005. Agents for wireless sensor network power management. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE Computer Society, 413–418.

Murphy, A. L., Picco, G. P., and Roman, G.-C. 2006. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol. 15*, 3, 279–328.

Murty, R., Gosain, A., Tierney, M., Brody, A., Fahad, A., Bers, J., and Welsh, M. 2007. Citysense: A vision for an urban-scale wireless networking testbed. Tech. rep. 13-07, Harvard University.

NECULA, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM, 106–119.

P.E.CLEMENTS, PAPAIOANNOU, T., AND EDWARDS, J. 1997. Aglets: Enabling the virtual enterprise. In *Proceedings of the International Conference on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement*.

PEINE, H. AND STOLPMANN, T. 1997. The architecture of the Ara platform for mobile agents. In *Proceedings of the 1st International Workshop on Mobile Agents*. R. Popescu-Zeletin and K. Rothermel Eds. Lecture Notes in Computer Science, vol. 1219, 50–61.

PERRIG, A., STANKOVIC, J., AND WAGNER, D. 2004. Security in wireless sensor networks. *Comm. ACM 47*, 6, 53–57.

PLATON, E. AND SEI, Y. 2008. Security engineering in wireless sensor networks. *Progress Inf. 5*, 3, 49–64.

POLASTRE, J., SZEWCZYK, R., AND CULLER, D. 2005. Telos: Enabling ultra lower-power wireless research. In *Proceedings of the IPSN'05 SPOTS*. ACM and IEEE, 364–369.

QI, H., IYENGAR, S. S., AND CHAKRABARTY, K. 2001a. Multi-resolution data integration using mobile agents in distributed sensor networks. *IEEE Trans. Syst. Man Cybernet. – Part C 31*, 3, 383–391.

QI, H., WANG, X., IYENGAR, S. S., AND CHAKRABARTY, K. 2001b. Multi-sensor data fusion in distributed sensor networks using mobile agents. In *Proceedings of 5th International Conference on Information Fusion.* 11–16.

QI, H., XU, Y., AND WANG, X. 2003. Mobile-agent-based collaborative signal and information processing in sensor networks. In *Proceedings of the IEEE 91*, IEEE 1172–1183.

REIJERS, N. AND LANGENDOEN, K. 2003. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications (WSNA'03)*. ACM, 60–67.

STALLINGS, W. 2001. *Operating Systems* 4th, Ed. Prentice Hall.

STATHOPOULOS, T., HEIDEMANN, J., AND ESTRIN, D. 2003. A remote code update mechanism for wireless sensor networks. Tech. rep. CENS-TR-30, UCLA.

TONG, L., ZHAO, Q., AND ADIREDDY, S. 2003. Sensor networks with mobile agents. In *Proceedings of the Military Communications International Symposium*. 688–693.

TSENG, Y.-C., KUO, S.-P., LEE, H.-W., AND HUANG, C.-F. 2004a. Location tracking in a wireless sensor network by mobile agents and its data fusion strategies. *Comput. J. 47*, 4, 448–460.

TSENG, Y.-C., KUO, S.-P., LEE, W.-W., AND HUANG, C.-F. 2004b. Location tracking in a wireless sensor network by mobile agents and its data fusion strategies. *Comput. J. 47*, 4, 448–460.

TYNAN, R., RUZZELLI, A. G., AND P., O. G. M. 2005. A methodology for the development of multi-agent systems on wireless sensor networks. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*.

TYNDALL NATIONAL INSTITUTE. 2005. The 25mm cube module.
http://www.tyndall.ie/research/mai-group/25cube_mai.html.

WAN, C.-Y., CAMPBELL, A. T., AND KRISHNAMURTHY, L. 2002. PSFQ: A reliable transport protocol for wireless sensor networks. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*. ACM, 1–11.

WELSH, M. AND MAINLAND, G. 2004. Programming sensor networks using abstract regions. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, 3–3.

WHITEHOUSE, K., SHARP, C., BREWER, E., AND CULLER, D. 2004. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*. ACM, 99–110.

WOO, A., TONG, T., AND CULLER, D. 2003. Taming the underlying challenges of reliable multi-hop routing in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*. ACM, 14–27.

WOOLDRIDGE, M. AND JENNINGS, N. 1995. Intelligent agents: Theory and practice. *IEEE Trans. Knowl. Eng. Rev. 10*, 2, 115–152.

WU, Q., RAO, N., BARHEN, J., IYENGAR, S. S., VAISHNAVI, V., QI, H., AND CHAKRABARTY, K. 2004. On computing mobile agent routes for data fusion in distributed sensor networks. *IEEE Trans. Knowl. Data Eng. 6*, 16, 740–753.

Yu, Y., Rittle, L. J., Bhandari, V., and LeBrun, J. B. 2006. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys'06)*. ACM, 139–152.

Zhao, J. and Govindan, R. 2003. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*. ACM, 1–13.

Received July 2008; accepted May 2009