Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu
Washington University in St. Louis
Saint Louis, Missouri 63130
{liang, roman, lu}@cse.wustl.edu

Abstract

Wireless sensor networks (WSNs) are difficult to program and usually run statically-installed software limiting its flexibility. To address this, we developed Agilla, a new middleware that increases network flexibility while simplifying application development. An Agilla network is deployed with no pre-installed application. Instead, users inject mobile agents that spread across nodes performing application-specific tasks. Each agent is autonomous, allowing multiple applications to share a network. Programming is simplified by allowing programmers to create agents using a high-level language. Linda-like tuple spaces are used for inter-agent communication and context discovery. This preserves each agent's autonomy while providing a rich infrastructure for building complex applications, and marks the first time mobile agents and tuple spaces are used in a unified framework for WSNs. Our efforts resulted in an implementation for MICA2 motes and the development of several applications. The implementation consumes a mere 41.6KB of code and 3.59KB of data memory. An agent can migrate 5 hops in less than 1.1 seconds with 92% reliability. In this paper, we present Agilla and provide a detailed evaluation of its implementation, an empirical study of its overhead, and a case study demonstrating its use.

1 Introduction

Wireless sensor networks (WSNs) have been under development for many years and are about to gain widespread use as technology improves, prices drop, and new applications are developed. They will proliferate because they can penetrate application areas for which traditional networks are inadequate. Instead of using a few sensing stations connected by wires or one-hop wireless links, WSNs consist of a multitude of tiny sensors attached to battery-powered microprocessors that opportunistically form a multi-hop wireless ad hoc network. The miniature size and number of these sensors enable micro-sensing at unprecedented resolutions and scales. Several examples of WSN deployments include habitat monitoring, microclimate research, surveillance, medical care, and structural monitoring [11]. However in order for WSNs to go mainstream, they must be cheaper, easier to use, and more flexible.

The existing software for WSNs is not flexible enough to meet the demands of many applications. Most WSNs are limited in that the application must be installed prior to deployment, and, once deployed, they can only be marginally tweaked to adapt to changing circumstances. Also, there are many situations where the application itself must change. For example, suppose a sensor network is deployed in a forest for detecting fires. When a fire is detected and the fire fighters arrive, they may want to reprogram the network with a search and rescue application. While it may be possible to integrate the two and install both, this is not scalable.

One way of increasing network flexibility is to allow in-network reprogramming. Two technologies that allow this are Maté [20], and SensorWare [6]. Both have disadvantages. In Maté, applications are divided into capsules that are flooded throughout the network. Each node stores the most recent version of each capsule and runs the application by interpreting the instructions within them. Maté does not allow a user to control where an application is installed. This limits the network to run a single application at a time. SensorWare allows users to dynamically inject mobile scripts into the network. This enables multiple applications to run concurrently, but the scripts only support weak mobility and have fixed points of entry. Also, the system was implemented for the relatively resource rich iPAQ 3670 platform.

To address the problems listed above, we have developed a new middleware called **Agilla**. Instead of relying on traditional fixed-location programs, Agilla adopts a mobile agent-based paradigm where programs are composed of agents that can migrate across nodes. Mobile agents are dynamic, localized, and intelligent. Each agent is, in ef-

fect, a virtual machine with dedicated instruction and data memory. As an agent executes, special instructions allow it to interact with the environment and move from one node to another. Multiple agents can coexist on a single node. Linda-like tuple spaces [14] are used to ensure each agent is autonomous while still able to communicate. They offer a shared memory model where the datum is a tuple that is accessed via pattern matching. This allows one agent to insert a tuple containing a sensor reading and another to later retrieve it without the two knowing each other, thus achieving a high level of decoupling.

Agilla provides many inherent benefits. In-network reprogramming is achieved since new agents can be injected and old agents can die. Multiple applications can coexist since agents belonging to different applications can coexist. An agent world-view can ease application development by diverting focus from complex distributed algorithms to individual agent behavior. For example, instead of worrying about how nodes must coordinate to track an intruder, a mobile agent programmer can think of an agent *following* the intruder by repeatedly migrating to the node that best detects it. Finally, by allowing post-deployment reprogramming, Agilla transforms WSNs into general-purpose computing platforms that are open for the public to use.

Mobile agents have been used for many years on the Internet and their benefits are known. Some middleware systems that provide agents include Agent Tcl [15], Ara [24], Java Aglets [23], Mole [5], Sumatra [3], TACOMA [17], PEERWARE [10], and MARS [9]. They have been successfully used in data mining [19], e-commerce [21], and network management applications [4]. Mobile agents have been shown to benefit other types of networks as well. LIME [22] was the first successful attempt at using mobile agents and tuple spaces in wireless ad hoc networks. It allowed agents to access a global tuple space containing the aggregate data of all nodes within range. It provided a high level of atomicity but required a predictable network. Since many agents have specific interests, EgoSpaces [18] was created to allow agents to focus only on relevant data. Both LIME and EgoSpaces required relatively powerful nodes and stable network connectivity. Since some networks do not meet this criteria, Limone [12] was created as a lightweight version of LIME. It eliminated all assumptions about the network, but provided weaker guarantees. This paper explores whether the use of mobile agents and tuple spaces in WSNs is technically feasible and beneficial to a WSN application developer. To our knowledge, nothing like this has been attempted to date.

WSNs are notoriously difficult to work with primarily due to their extremely limited resources. Each node consists of a relatively slow processor, little memory, and an unreliable low-bandwidth radio with limited range. WSNs often have long deployment intervals during which it is likely for the demands of the user, or the users themselves, to change. Applications must be flexible to adapt to a changing context, network topology, and user needs. Agilla helps applications achieve this.

This paper makes four contributions. First, it explores the benefits of using mobile agents and tuple spaces as a foundation for developing new WSN applications. Second, it examines the technical challenges associated with designing Agilla and tailoring it to fit the salient properties of WSNs. Third, it demonstrates the feasibility of using mobile agents and tuple spaces in existing WSNs through the development of middleware. Finally, it evaluates the performance of Agilla in terms of easing application development and overhead. These contributions provide valuable engineering lessons for future efforts related to software developments in the area of WSNs.

The remainder of the paper is organized as follows. Section 2 presents Agilla's model and explains how it was tailored to the unique properties of WSNs. Section 3 discusses the various engineering tradeoffs that had to be made to cope with limited resources and an unreliable network. Section 4 presents the experimental results on Agilla's efficiency and reliability. Section 5 contains a case study that illustrates how Agilla makes programming a sophisticated fire detection application simpler. The paper ends with conclusions in Section 6.

2 Model

This section first presents a motivating example and then describes Agilla's model in light of this application.

2.1 Motivating Example

In the remote arid forests of central Arizona, lighting ignites a fire that quickly spreads with the prevailing winds. The remoteness of the region would allow the fire to burn undetected for hours, virtually ensuring that it will soon rage out of control. Fortunately, the USDA Forest Service had recognized this area as highly incendiary and deployed a WSN for detecting fire. As the fire grows, nearby sensors quickly detect it and spawn tracking agents that swarm around the fire collecting information about the exact location of the flames. The tracking agents form a dynamic perimeter jumping away from the fire as it draws too near, and cloning themselves onto neighbors to encompass the growing fire. Simultaneously, they notify a base station that forwards the warning via the Internet to the nearest fire fighters a hundred miles away. By the time they arrive, the entire region is engulfed burning with such intensity so as to be seen and felt from miles away.

The fire fighters act quickly and make it their first priority to evacuate the area. They inject search-and-rescue agents

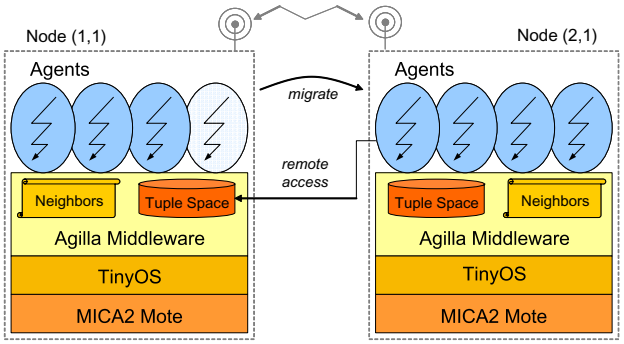

Figure 1. The Agilla model

into the network that spread and repeatedly clone themselves scouring the region looking for lost hikers trapped by the flames. Some of these agents find a group of children and coordinate with the other agents to form a path of greatest safety that the rescuers, carrying PDAs to access the path information, use to reach the children and bring them to safety. Once everyone is safe, the fire fighters query the tracking agents for the precise location and dynamics of the fire. From this data, they are able to predict the fire's behavior and strategically control its movements preventing it from approaching populated areas where property can be damaged and people injured.

Once the fire has died, the tracking agents also die leaving only small fire detection agents that periodically searches for fire. The minuscule resource consumed by these agents allow other applications to run, which biologists do by injecting state-of-the-art habitat monitoring agents for learning about the life cycle of coyotes.

2.2 The Agilla Model

The Agilla model is shown in Figure 1. Each node supports multiple agents and maintains a tuple space and neighbor list. The tuple space is local and is shared by the agents residing on the node. Special instructions allow agents to remotely access another node's tuple space. The neighbor list contains the address of all one-hop nodes. Agents can migrate carrying their code and state, but do not carry their own tuple spaces.

An Agilla application consists of numerous autonomous agents, possibly of different types, scattered throughout a network. For example, in the motivating example, there are fire detection agents, tracking agents, and search-and-rescue agents. Given all these agents, there must be some coordination mechanism that allows them to communicate. Agilla provides this through tuple spaces. Agilla tuple spaces offer a shared memory model where the datum is a tuple. Tuples adhere to a strict format and are accessed by pattern matching via templates. A tuple is an ordered set of fields where each field has a type and value. Types may include integers, strings, locations, and sensor readings. Tuples are accessed using templates that are also ordered sets of fields. Templates are unique in that their fields may contains wild cards that match by type. To extract a tuple from a tuple space, the agent needs to provide a template that matches the tuple. A template matches a tuple if they have the same number of fields, and each field in the tuple matches the corresponding field in the template.

Tuple spaces provide a high level of decoupling that ensures each agent remains autonomous and provides a convenient way for an agent to discover its context. For example, since each node may have different sensors, Agilla places special tuples into each node's tuple space indicating what type of sensors are available. If a node has a thermometer, Agilla would insert a "temperature tuple" into its tuple space. These tuples are pre-defined, allowing an agent to discover what type of sensors are available. Other context information stored in the tuple space include the location and the number of co-located agents and their identities. Tradeoffs had to be made regarding what information to store in the tuple space versus providing special accessor instructions, and is discussed further in Section 3.

Agilla tuple spaces provide operations `out`, `in`, `rd`, `inp`, and `rdp`. They are atomic and operate over the local tuple space. `out` inserts a tuple. `in` and `rd` are blocking operations that remove and copy a tuple from the tuple space, respectively. If a match does not exist, the executing agent blocks until a match appears. `inp` and `rdp` are the same as `in` and `rd` except they do not block.

Like many other systems [8, 12, 18, 22], Agilla adds *reactions* to its tuple spaces. Reactions allow an agent to tell Agilla that it is interested in tuples that match a particular template. When the matching tuple is placed into the tuple space, the agent is notified, allowing it to immediately respond. Without reactions, an agent would either have to block or poll waiting for the tuple to appear, both of which are inefficient. Agilla reactions are strictly local, an agent can only react to tuples in the local tuple space.

A tuple space ensures that each agent runs autonomously by allowing them to communicate in a decoupled fashion. For example, suppose there is a fire detection and habitat monitoring agent residing on the same node when fire is detected. The fire detection agent inserts a fire tuple into the local tuple space to indicate the presence of fire and activates a tracking agent before dying. The habitat monitoring agent reacts to this tuple, and voluntarily kills itself to free additional resources. Notice how the fire detection agent does not need to know who received the fire tuple, the sending and reception can occur at different times, and reception can occur even if the sender is no longer present. This spatial and temporal decoupling ensures each agent operates autonomously.

Agilla agents also need to coordinate with agents residing on remote nodes. For example, in the motivating example, the tracking agents need to coordinate to ensure the perimeter is not breached. Agilla allows agents to coordinate across nodes by introducing special remote tuple space operations. These include `rout`, `rinp`, and `rrdp`. They are synonymous with `out`, `inp`, and `rdp` except they take an additional location parameter that specifies on which node to perform the operation. Only probing operations are provided to prevent an agent from blocking forever due to message loss. In the example above, the tracking agents would periodically perform `rrdp` operations to ensure neighboring tracking agents are still alive.

Note that Agilla does *not* support tuple spaces that span across nodes. Instead, it supports *local* tuple spaces where each node maintains a distinct and separate tuple space. The dedicated remote tuple space instructions described above rely on unicast communication with the specific node hosting the tuple space. Hence, a remote tuple space operation entails the transmission of only two messages, a request and a reply, and is scalable to networks of any size.

Agilla assumes each node knows its physical location. This is reasonable since sensor data is often meaningless unless the location at which it was obtained is known. Nodes may acquire their location through GPS or any number of localization techniques [7]. Agilla provides one-hop neighbor discovery using beacons. The one-hop neighbor information is stored in an acquaintance list and is continuously updated by Agilla. Agents can access this list using special instructions `numnbr` and `getnbr`. Geographic routing allows an agent to access the tuple space on any node.

The idea behind Agilla is to initially deploy a network without any application installed. Agents that implement application behavior can later be injected, effectively reprogramming the network. An agent's life cycle begins when it is either injected into the network by the user, or cloned from another agent already in the network. An agent contains its own instruction memory, data memory, program counter, operand stack, and heap. Agilla executes each agent as an autonomous virtual machine and supports multiple agents on a node. Each agent employs a stack-architecture. Along with all the usual instructions that enable general-purpose computing and inter-agent communication, an agent can execute special instructions that move or clone it from one node to another. They include `smove`, `wmove`, `sclone`, and `wclone`. When an agent moves, it carries its state and code and resumes executing on the new node. When it clones, it copies its state and code to another node and resumes executing on both. The first letter of the migration instruction specifies whether the operation is *weak* or *strong*. In a weak operation, only the code is transferred. The program counter, heap, and stack are reset and the agent resumes running from the beginning. In a strong operation, everything is transferred and the agent resumes executing where it left off. An agent can move or clone itself to any node regardless of the number of hops away. The multi-hop migration is handled by the underlying middleware and is transparent to the user. When an agent completes its task it dies, allowing Agilla to free its resources and use them for other agents. An agent dies by executing the `halt` instruction.

```
1: BEGIN  pushn fir
2:        pusht LOCATION
3:        pushc 2
4:        pushc FIRE
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
```

Figure 2. The FIRETRACKER agent

WSNs rely heavily on spatial information. For example, a collection of temperature readings is not useful if it is not known from where the readings were obtained. For this reason, Agilla identifies nodes based on their location rather than their network address. A node's location is its address. Thus, instead of performing a `rout` operation on node 1, an agent performs it on a node at (x,y). Agilla addresses all nodes by their location. To account for slight errors in location, Agilla allows an error ϵ when specifying the address. By using location as addresses, Agilla primitives can be easily generalized to enable operations on a region. For example, a fire detection node can clone itself on all nodes in a geographic area, or alternatively it can clone itself to at least one node in the region.

To solidify Agilla's model, Figure 2 shows the FIRETRACKER agent mentioned in the motivating example. Recall that FIRETRACKER agents swarm around the fire forming a dynamic perimeter, a complex process that consumes lots of resources. To minimize overhead, the application uses lightweight FIREDETECTOR agents during idle periods, and spawn heavier-weight FIRETRACKER agents only when needed. Figure 2 demonstrates how a FIRETRACKER agent is notified. When a FIRETRACKER agent is injected into a node, it registers a reaction sensitive to `FireAlert` tuples and waits for the reaction to fire. This is done by lines 1-6. When a FIREDETECTOR agent detects fire, it performs a `rout(FireAlert)` on the node hosting the FIRETRACKER agent, which reacts to the tuple by executing the code starting from line 7. Notice that on line 8, the agent clones itself at the node that detected the fire. Once there, it will continue to clone and spread out to form a dynamic barrier around the flames.

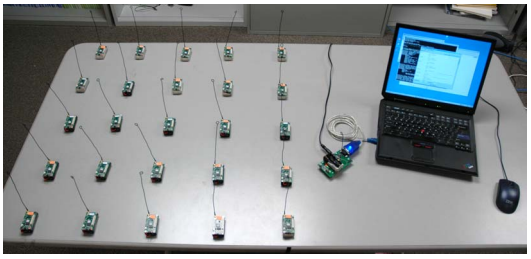

Figure 3. Our experimental test bed with 25 motes and a laptop as the base station

3 Engineering Effort

This section discusses the engineering effort behind Agilla. It starts with an overview of the sensor network's hardware and operating system. It then presents the architecture of Agilla's middleware followed by that of the agent. It ends with a discussion of the agent's instruction set.

3.1 Implementation Platform

Agilla has been implemented and tested on MICA2 motes [1], as shown in Figure 3. These motes have an 8 MHz Atmel ATmega128L 8-bit microprocessor connected to a Chipcon CC1000 radio transceiver. The radio communicates at up to 38 Kbps over a range of 100m, though the actual amounts vary substantially based on the environment [25]. They have 128KB of instruction and 4KB of data memory. MICA2 motes are representative of a typical device used in WSNs. Developing applications for them is challenging primarily due to the limited amount of data memory and a highly unreliable low-bandwidth radio.

The WSN communicates with a relatively powerful base station with access to the Internet. Our platform uses a laptop. It has a MIB510 interface board that forms a bridge between the WSN and Internet. The laptop runs a Java application that allows a user to interact with the WSN by injecting agents and performing remote tuple space operations. It also starts an RMI server that allows anyone on the Internet to remotely access the sensor network.

MICA2 motes run a simple but highly concurrent operating system called TinyOS [16]. TinyOS applications are divided into components that are arranged in a hierarchy. A main challenge with using TinyOS is the lack of dynamic memory management. All variables must be declared statically. While this simplifies compile-time analysis, it also makes the meager 4KB of data memory more precious. As pointed out in [20], TinyOS has a high learning curve, which is compounded by the limited resources and unreliable radio. In addition, the hard-wiring of TinyOS components makes it difficult to develop flexible applications. To change a program's behavior, the new behavior must either be pre-coded, or the mote needs to be retrieved and reprogrammed. Having a middleware that provides a higher-level programming abstraction that hides these complexities allow programmers to quickly implement, test, and deploy their applications.

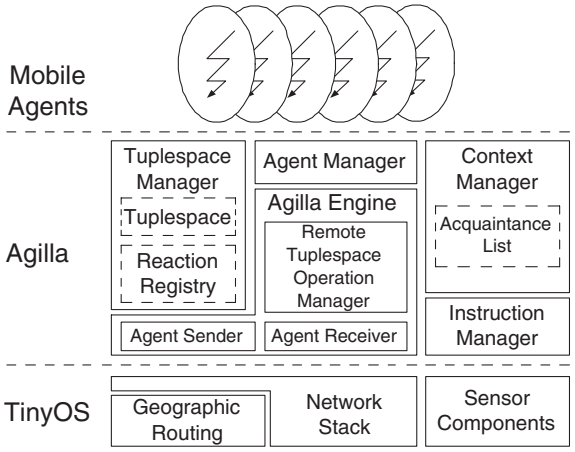

Figure 4. Agilla's middleware architecture

3.2 Agilla Architecture

Agilla's architecture, shown in Figure 4, is divided into three layers, the highest containing the agents that are discussed further in Section 3.3. The middle layer contains the core Agilla middleware components, while the bottom is TinyOS. Agilla's core middleware consists of an agent, context, instruction, and tuple space manager and an engine that orchestrates them.

Agent Manager. The agent manager maintains each agent's context. It is responsible for allocating memory for an agent when it arrives and de-allocating it when it leaves or dies. It is also responsible for determining when an agent is ready to run, and notifies the Agilla engine when this occurs. By default the agent manager can handle up to 4 agents. This is easily changed and is primarily limited by processor speed and the amount of memory available.

Context Manager. The context manager determines its location as well as that of its neighbors. It uses beacons to discover neighbors and stores the neighbor locations in an acquaintance list that is accessible to an agent via special instructions (e.g., `numnbrs`, `getnbr`, and `randnbr`). We use dedicated instructions rather than the tuple space because they are frequently used. Allowing an agent to know its location and that of its neighbors is vital. In the motivating example, FIREDETECTOR agents need to tell FIRETRACKER agents where they are. FIRETRACKER agents will then need to know the location of their neighbors to dynamically adjust the perimeter.

Instruction Manager. Since TinyOS does not provide dynamic memory allocation, Agilla implements one that is

tailored to the severe resource limitations of MICA2 motes. When an agent arrives, it specifies the amount of instruction memory it requires, and the instruction manager allocates the minimum number of 22 byte blocks necessary to store the agent's code. We found that 22 byte blocks are a good compromise between internal fragmentation and undue forward pointer overhead. When agents are running, the instruction manager retrieves the next instruction to execute. When an agent migrates, it packages up the agent's code into the minimal number of messages. By default, the instruction manager is allocated 440 bytes (20 blocks). With a few exceptions, an instruction is one byte meaning an agent can have up to 440 instructions.

The alternative to using a byte code manager is to allocate a fixed amount of memory to each agent. The disadvantage of this is waste if an agent does not use all of its allotted memory, and it also limits the maximum size of an agent's program.

Tuple Space Manager. The tuple space manager implements all of the non-blocking tuple space operations (e.g., `out`, `inp` and `rdp`) and reactions, and manages the contents of the local tuple space and reaction registry. The blocking operations are implemented within the agent, as described in Section 3.3. The tuple space manager dynamically allocates memory for each tuple. By default, it is allocated 600 bytes and a tuple may contain up to 25 bytes worth of fields. This ensures a tuple can fit within the 27 byte payload of a single TinyOS message. To prevent internal fragmentation and the need for forward pointers, the 600-bytes are allocated linearly. When a tuple is removed, all following tuples are shifted forward. While this may result in more memory swapping, it is simple. We leave a more in-depth investigation of efficient tuple space implementations as future work.

The tuple space manager remembers the reactions registered by each agent by storing them within the reaction registry. Whenever a tuple is inserted, it checks the registry for a match. If the new tuple matches a reaction's template, the tuple space manager notifies the agent manager, which updates the agent's program counter to execute the reaction's code. During a migration, the tuple space manager packages up all reactions registered by an agent so they can be transferred along with the agent. When an agent arrives, it automatically restores all of the agent's reactions. By default the reaction registry is allocated 400 bytes, allowing it to remember up to 10 reactions.

Agilla Engine. The Agilla engine serves as the virtual machine kernel that controls the concurrent execution of all agents on a node. It implements a simple round-robin scheduling policy where each agent can execute a fixed number of instructions before switching context. The default number of instructions is 4, which is the same as in Maté. Naturally, if an agent executes a long-running instruction like `sleep`, `sense`, or `wait`, the engine immediately switches context.

Type	Size (Bytes)	Content
State	20	program counter, code size, condition code, stack pointer
Code	28	one instruction block
Heap	32	four variables and their addresses
Stack	30	four variables
Reaction	36	one reaction

Figure 5. Messages used during migration

The Agilla engine also handles the arrival and departure of agents. This is particularly difficult due to the highly unreliable nature of MICA2's radio. It is compounded by the fact that an agent cannot be sent in a single message. When an agent migrates, Agilla divides it into numerous types of messages as shown in Figure 5. At a minimum, a migration requires two messages: one state and one code. Many agents require more since they have data in their stack and heap, and have registered reactions. If a single message is lost, the migration operation will fail. To help minimize this problem, agents are migrated one hop at a time, and each message is acknowledged. We tried using end-to-end communication where messages are not acknowledged till they reach the final destination, but found that the high packet-loss probability over multiple links made this unacceptably prone to failure. If a one-hop acknowledgement is not received within 0.1 seconds, the message is retransmitted. This repeats up for four times. If the operation stalls for over 0.25 seconds, the receiver aborts. If the sender detects a failure, it resumes the agent running on the local machine with the condition code set to zero. While this may result in duplicate agents, the alternative is to simply kill the agent. We decided that having duplicate agents in the network is preferable. Consider the motivating example; it is better to have duplicate warnings that there is a fire rather than no warnings at all.

Remote tuple space operations are also handled by the Agilla Engine. To perform a remote tuple space operation, a request containing the instruction and template is sent to the destination node. When the destination receives it, it performs the operation on its local tuple space and sends back the result. Unlike agent migration operations, we used end-to-end communication for remote tuple space operations and do not use acknowledgements. This is because they are usually done on nearby nodes, a request can fit in one message, and the operational semantics are not broken if a message is lost. To reduce the effects of message loss, the initiator timeouts after 2 seconds and re-transmits the request at most twice.

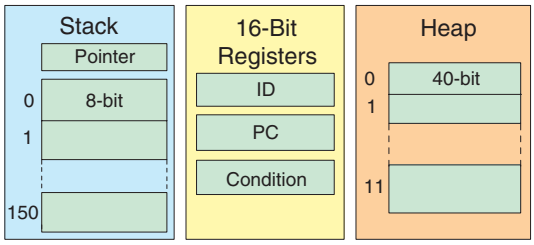

Figure 6. The mobile agent architecture

3.3 Agent Architecture

The agent architecture is shown in Figure 6. It consists of a stack, heap, and various registers. Mobile agents use a stack architecture because it allows most instructions to be a single byte (a few consume 3 bytes for pushing 16-bit variables onto the stack). The heap is a random-access storage area that allows an agent to store up to 12 variables. It is accessed by the `getvar` and `setvar` instructions.

The agent also contains three 16-bit registers: one containing the agent's ID, another with the program counter (PC), and the last with the condition code. The agent ID is unique to each agent and is maintained across move operations. A cloned agent is assigned a new ID. The PC is the address of the next instruction. It is modified by the jump instructions and is used by the code manager to fetch the next instruction. When a reaction fires, the reaction manager changes the PC to point to the first instruction of the reaction's code. To allow an agent to resume executing where it was when the reaction fired, the original PC is stored on the stack. Finally, the condition code is a 16-bit register that records execution status. For example, the instruction `ceq` sets the condition to be 1 if the top two variables in the stack are equal.

3.4 Agent Instruction Set Architecture (ISA)

Agilla's ISA is based on that of Maté. However, there are many differences that are necessary for supporting agent mobility and tuple spaces. Some instructions unique to Agilla are shown in Figure 7. A full listing is available at [2]. Agilla's ISA can be divided into three categories: general purpose, tuple space, and migration instructions.

General purpose instructions. Agilla's general purpose instructions are nearly identical to that of Maté. They include, among many others, `add`, `halt`, `putled`, `or`, `rand`, `sense`, `eq`, `pop`, and `pushc`. New instructions used by Agilla are `sleep`, `rjump`, `rjumpc`, `aid`, and `pushcl`. These enable an agent to achieve sophisticated behavior without using multiple components, which is necessary in Maté. For example, an Agilla agent can perform some application-specific actions, sleep, and jump back to repeat. Maté can only achieve this in its timer capsule.

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) be placed onto the stack as a parameter. This is done by pushing each field followed by the number of fields. For example, in Figure 2, lines 1-3 pushes a template with two fields onto the stack.

Dedicated instructions `out`, `in`, `rd`, `inp`, and `rdp` are provided for accessing the local tuple space. The blocking `in` and `rd` operations are implemented by having the agent repeatedly trying to `inp` or `rdp` a tuple. If the probe fails, the agent's context is stored in a wait queue until a tuple is inserted. When this occurs, the agents in this queue are notified and can re-check for a match. The remote tuple space operations `rout`, `rinp`, and `rrdp` are non-blocking to account for message loss and disconnection. If the operation is successful, the resulting tuple is placed onto the stack and the condition is set to 1. The tuple space instructions also include `regrxn` and `deregrxn`. They allow an agent to register and deregister a reaction, respectively. Both instructions require a template and value be pushed onto the stack, where the value is the address of the first instruction of the reaction's code.

Migration instructions. The migration instructions allow an agent to move or clone from one node to another, possibly multiple hops away. Agilla provides four migration instructions: `smove`, `wmove`, `sclone`, and `wclone`. They were discussed in Section 2.2.

4 Performance Evaluation

This section evaluates Agilla. It first examines the reliability of remote tuple space and agent migration operations, and then investigates the overhead of Agilla's instructions. Our MICA2 network is arranged in a 5x5 grid as shown in Figure 3. Each node is assigned an (x,y) coordinate based on its grid position, where the node in the lower-left corner has a location of (1,1). To simulate multi-hop routing, we modified TinyOS's network stack to filter out all messages except those from immediate neighbors based on the grid topology. For geographic routing, we implemented a simple best-effort greedy-forwarding algorithm that forwards messages to the neighbor closest to the destination.

WSNs are notorious for having highly unreliable wireless links. This is problematic since agents migrate using multiple messages and the operation will fail if just one message is lost. Agilla uses acknowledgements and timers to retransmit if it suspects message loss. To avoid excessive overhead, this is only done during migration and not during remote tuple space operations. To test the reliability, the agents shown in Figure 8 are injected into node (0,0). The `smove` agent moves to a remote node and back while the `rout` agent places a tuple in a remote node's tuple space. Each agent is run 100 times for 1-5 hops. The latency of

Instruction	Opcode	Parameters	Return Values	Description
loc	0x01	n/a	[location]	Pushes host's location onto the stack
wait	0x0b	n/a	n/a	Stops agent execution, allows it to wait for a reaction
smove	0x1a	[location]	n/a	Strong move
wclone	0x1d	[location]	n/a	Weak clone
getnbr	0x20	[value]	[location]	Get a neighbor's address
out	0x33	[tuple]	n/a	Insert a tuple into the local tuple space
inp	0x34	[template]	[tuple]?	Non-blocking find and remove tuple from tuple space
rd	0x37	[template]	[tuple]	Blocking find tuple in tuple space
rout	0x39	[location], [tuple]	n/a	Insert a tuple into a remote tuple space
rinp	0x3a	[location], [template]	[tuple]?	Non-blocking find and remove tuple from remote tuple space
regrxn	0x3e	[template], [value]	n/a	Register a reaction on the local tuple space

Figure 7. Noteworthy Agilla instructions

```
// The smove agent
1:    pushloc 5 1
2     smove          // strong move to mote at (5,1)
3:    pushloc 0 0
4:    smove          // strong move to mote at (0,0)
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 mote (5,1)
5:    halt
```

Figure 8. The agents that test `smove` (top) and `rout` (bottom)

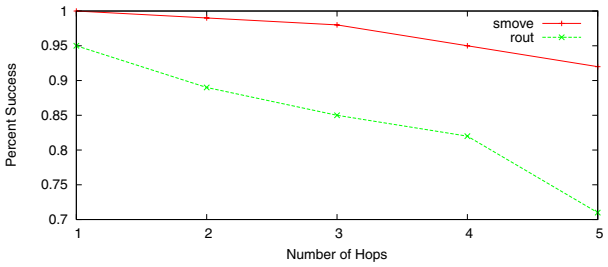

Figure 9. The reliability of `smove` vs. `rout`

each successful execution, and the number of failures are recorded (`smove` results are halved to account for the double migration). The results, shown in Figures 9 and 10, indicate that both operations perform well across short distances. However, as the distance increases, the probability of a message being lost also increases, which is reflected in a decrease in reliability. The results show that `smove` is more reliable than `rout`, but has higher latency. Since successful migration is vital, we feel the additional overhead of the agent migration protocol is justified. There is clearly a tradeoff between latency and reliability.

To determine whether `rout` and `smove` are representative of the other remote tuple space and agent migration in-

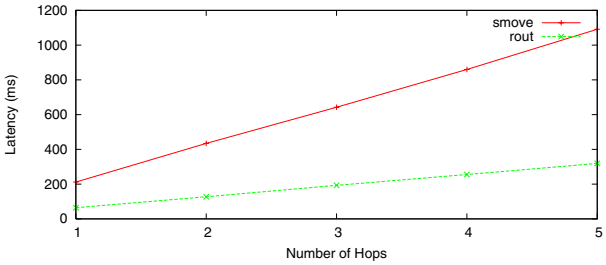

Figure 10. The latency of `smove` vs. `rout`

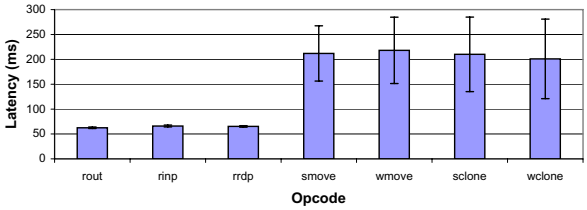

Figure 11. The latency of remote operations.

structions, we found the one-hop execution time of all these instructions by timing each 100 times and finding the average. The results, shown in Figure 11, indicate that `rout` and `smove` are representative, and that agent migration instructions have significantly higher overhead than remote tuple space operations. Note that migration operations have higher variance. This makes sense since they employ retransmit timers in the event of message loss. The results also suggest that the quickest an agent can migrate is once every 0.3 seconds. Assuming the radio range is around 50m, this means an agent can migrate across a network at 600km/h or 373mph, which is sufficient for tracking many interesting events like fire.

We now benchmark local operations unique to Agilla. Like Maté, Agilla executes each instruction as a separate task. To determine the execution times of these instructions, we disabled the radio and timed how long it took to execute each 1000 times, then repeated it 100 times. We calculated the average execution time of each instruction and the results, shown in Figure 12, indicate that there are three general classes of local operations. The first class has the

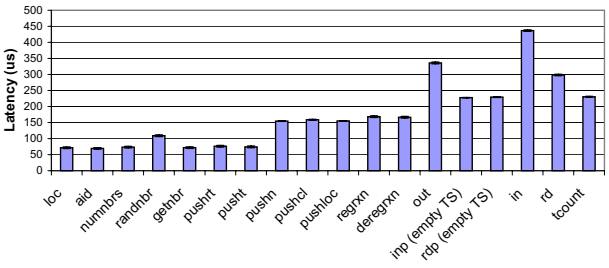

Figure 12. The latency of local operations.

least execution time and include the `loc`, `aid`, `numnbrs`, and various `push` instructions. These instructions simply push a value onto the stack and do not perform any calculation. They take about 75μs. The second class of instructions take longer because they either perform additional memory accesses (e.g., `pushn`, `pushcl`, `pushloc`, `regrxn`, and `deregrxn`), or perform simple computations (e.g., `randnbr`). These instructions take around 150μs to execute. The last group of instructions cost the most and consist of tuple space operations. However, they still execute fairly quickly averaging 292μs. Note that the blocking tuple space operations take slightly longer than the non-blocking ones. This makes sense since blocking operations need to undergo the additional overhead of checking whether the non-blocking equivalent failed and blocking until a result is found. Also note that `in` takes longer than `rd`, which makes sense since it requires modifying the state of the tuple space.

Agilla can perform one-hop remote tuple space operations in about 55ms, and migration operations in 225ms. The execution time scales linearly with the number of hops, and the additional overhead for migration operations is justified by their resilience to message loss across multiple hops. Local operations take between 60-440μs. This demonstrates the feasibility and efficiency of using mobile agents and tuple spaces in a representative WSN. We did not directly compare Agilla's instructions with other sensor network middleware like Maté because many of Agilla's instructions are higher level and do not have a corresponding instruction with which to compare to. However, the latency of simpler Agilla instructions like `loc` and `aid` that execute within 100μs are comparable to those of Maté.

5 Usability Case Study

This section provides a case study on Agilla's ability to simplify application development. Returning to the motivating example given in Section 2.1, we see that Agilla enables the development of complex applications by using agents as the basic unit of execution and modularity, and tuple spaces as a means of communication. We have developed a prototype that consists of two types of agents: 1) a

```
... // bootstrapping code omitted
1: BEGIN   pushc TEMPERATURE
2:         sense          // measure the temperature
3:         pushcl 200     // push 200 onto stack
4:         clt            // set condition=1 if temperature > 200
5:         rjumpc FIRE    // jump to FIRE if condition=1
6:         pushcl 4800
7:         sleep          // sleep for 10 minutes
8:         rjump BEGIN
9: FIRE    pushn fir      // push string "fir"
10:        loc            // push current location
11:        pushc 2        // stack has fire alert tuple
12:        pushloc 0 0
13:        rout           // rout fire alert tuple on node at (0,0)
14:        halt
```

Figure 13. The FIREDETECTOR agent

FIREDETECTOR agent whose sole responsibility is to detect fire and 2) a FIRETRACKER agent that actually creates the dynamic border around the fire. We use two types of agents to minimize resource utilization. FIREDETECTOR agents consume few resources and are spread throughout the network during idle periods. Its code is shown in Figure 13. The initial bootstrapping code for cloning the agent throughout the network is omitted. In lines 1-8, the agent takes a temperature reading every 10 seconds. It assumes there is a fire if the sensor returns a value greater than 200. When fire is detected, lines 9-14 notifies the closest FIRETRACKER agent via an `rout`, which springs to life cloning itself and forming a perimeter around the flames, as shown in Figure 2. The FIRETRACKER agent is more complex and its code is not shown in its entirety due to lack of space, but is available at [2]. Details of our experience with developing the FIRETRACKER agent is presented in [13].

We now qualitatively compare the flexibility of Agilla and Maté. Comparing the two middleware systems is difficult because of Maté's relative lack of flexibility. For example, it might be possible to compare Agilla's FIREDETECTOR agents with a similar program written in Maté since they both need to be installed throughout the network. But once fire is detected, instead of notifying a nearby FIRETRACKER agent, a program written in Maté will either have to include the tracking function with the detection code, or the base station will have to be notified so it can re-program the entire network. Both are less efficient as they entail distributing code throughout the entire network. It is also less flexible since only one application is enabled to run on the network at a time.

6 Conclusion

Agilla promises to accelerate the rate at which our society adopts WSNs and harnesses the benefits that these networks offer. It simplifies the way WSN applications are developed and the way WSNs are perceived and used. Instead of working with rigid fixed-location code written in a

complex language, developers can write mobile agents in a higher-level language, which can migrate across nodes performing application-specific tasks. Instead of deploying a propriety network with an application pre-installed, an Agilla network is deployed without any program initially, but can be continuously reprogrammed by injecting new agents. By using tuple spaces, each agent remains autonomous allowing multiple users with different applications to simultaneously share the same network. The sensor network becomes a general-purpose computing grid with a heightened degree of context awareness. This paper takes the first step of supporting a mobile agent-based programming paradigm in WSNs. Our MICA2 implementation demonstrates the feasibility of using mobile agents and a tuple space-based coordination model in WSNs and serves as a foundation for rapidly building flexible WSN applications.

Acknowledgment

This research was supported by the Office of Naval Research under MURI research contract N00014-02-1-0715 and by the the NSF under ITR contract CCR-0325529. 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 also thank the reviewers for their valuable feedback.

References

[1] http://www.xbow.com.

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

[3] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[4] M. Baldi and G. P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In R. Kemmerer, editor, *Proceedings of the 20th International Conference on Software Engineering*, pages 146–155. IEEE Computer Society Press, 1998.

[5] J. Baumann, H. K. Rothermel, M. Strasser, and W. Theilmann. Mole: A mobile agent system. *Softw. Pract. Exper.*, 32(6):575–603, 2002.

[6] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys*, 2003.

[7] N. Bulusu, J. Heidemann, and D. Estrin. Gps-less low cost outdoor localization for very small devices. Technical Report 00-729, University of Southern California, April 2000.

[8] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, 1477:237–252, 1998.

[9] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *Internet Computing*, 4(4):26–35, 2000.

[10] G. Cugola and G. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, 2001.

[11] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *IEEE Computer*, 37(8):41–49, 2004.

[12] C.-L. Fok, G.-C. Roman, and G. Hackmann. A Lightweight Coordination Middleware for Mobile Computing. In R. DeNicola, G. Ferrari, and G. Meredith, editors, *Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004)*, number 2949 in Lecture Notes in Computer Science, pages 135–151. Springer-Verlag, 2004.

[13] C.-L. Fok, G.-C. Roman, and C. Lu. Mobile agent middleware for sensor networks: An application case study. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN'05)*, 2005.

[14] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[15] R. Gray. Agent Tcl. *Dr. Dobb's Journal of Software Tools*, 22(3):18–71, 1997.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[17] D. Johansen, R. van Renesse, and F. B. Schneider. An introduction to the TACOMA distributed system—version 1.0. Technical Report 95-23, University of Tromsø, Tromsø, Norway, June 1995.

[18] C. Julien and G.-C. Roman. Egocentric Context-Aware Programming in Ad hoc Mobile Environments. In *Pro. of the 10^{th} Int. Symp. on the Foundations of Software Engineering*, pages 21–30, Nov. 2002.

[19] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[20] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[21] P. Maes, R. H. Guttman, and A. G. Moukas. Agents that buy and sell. *Communications of the ACM*, 42(3):81–91, 1999.

[22] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21^{st} International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[23] P.E.Clements, T. Papaioannou, and J. Edwards. Aglets: Enabling the virtual enterprise. In *Proc. of the Int. Conf. on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement*, 1997.

[24] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In R. Popescu-Zeletin and K. Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, Apr. 1997. Springer Verlag.

[25] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. of the ACM SenSys*, 2003.