Mobile Agent Middleware for Sensor Networks: An Application Case Study

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

Abstract— **Agilla is a mobile agent middleware that facilitates the rapid deployment of adaptive applications in wireless sensor networks (WSNs). Agilla allows users to create and inject special programs called mobile agents that coordinate through local tuple spaces, and migrate across the WSN performing application-specific tasks. This fluidity of code and state has the potential to transform a WSN into a shared, general-purpose computing platform capable of running several autonomous applications at a time, allowing us to harness its full potential. We have implemented and evaluated a fire tracking application to determine how well Agilla achieves its goals. Fire is modeled by agents that gradually spread throughout the network, engulfing nodes by inserting fire tuples into their local tuple spaces. Fire tracker agents are then used to form a perimeter around the fire. Using Agilla, we were able to rapidly create and deploy 47 byte fire agents, and 100 byte tracker agents on a WSN consisting of 26 MICA2 motes. Our experiments show that the tracker agents can form an 8-node perimeter around a burning node within 6.5 seconds and that it can adapt to a fire spreading at a rate of 7 seconds per hop. We also present the lessons learned about the adequacy of Agilla's primitives, and regarding the efficiency, reliability, and adaptivity of mobile agents in a WSN.**

I. Introduction

Wireless sensor networks (WSNs) consist of tiny sensors deeply embedded within the environment. Example applications include habitat monitoring, microclimate research, surveillance, medical care, and structural monitoring [1]–[5]. Such applications have lengthy deployment intervals during which the sheer number of nodes, and their exposure to the harsh physical environment, result in a high probability that many of them will be disabled. The long deployment intervals also increase the likelihood that the user requirements will change. Furthermore, WSNs must deal with highly dynamic environments. For instance, while a fire tracking network deployed in a forest may remain dormant most of the time, a wildfire may break out and spread unpredictably, rapidly triggering numerous network activities. Therefore, WSN applications need to be highly flexible and adaptive, which places an additional burden on the application developer.

WSN applications are inherently difficult to develop and deploy. For example, a representative WSN platform is comprised of MICA2 motes and the TinyOS [6] operating system. TinyOS is a minimalist event-based operating system with a high learning curve [7]. It hard-wires software components limiting flexible application development. Thus, once deployed, TinyOS applications can only be marginally tweaked by changing parameters defined prior to deployment. However, the number of nodes and the fact that they are deeply embedded preclude the option of manually collecting each node, reprogramming them, and re-deploying them. To complicate matters, memory and other computational resources are scarce, and radio communication is notoriously unreliable [8].

Middleware promises to enhance the flexibility of WSN applications. Excluding Agilla, other middleware include XNP [9], Deluge [10], Maté [7], SensorWare [11], Impala [12] and Smart Messages [13]. XNP and Deluge both support network reconfiguration by flashing the mote's instruction memory. Deluge improves upon XNP by enabling multi-hop support. Both XNP and Deluge take a long time and consume a lot of energy transferring an entire image over the lossy wireless network and flashing the instruction memory. Maté and Impala both divide an event-based application into modules that are flooded across the network. Maté is a virtual machine, Impala uses native code. More fundamentally, XNP, Deluge, Maté, and Impala do not allow a programmer to control where a program is installed (it is installed on every node) preventing multiple applications from running on a network simultaneously. SensorWare allows users to inject mobile scripts into the network, but the scripts only support weak mobility, execution state is not transferred. Both SensorWare and Impala are implemented on the relatively powerful iPAQ 3670 platform.

To address the limitations of the above middleware solutions, we have developed Agilla [14], a mobile agent middleware for WSNs. Agilla is based on Maté, but unlike Maté which divides an application into capsules that are flooded throughout a network, Agilla allows users to deploy applications by injecting mobile agents into a sensor network. Mobile agents can intelligently move or clone themselves to desired locations in response to changing conditions in the environment. Each node maintains a local tuple space, and different agents can coordinate through local or remote operations on these tuple spaces. We have implemented Agilla on the MICA2 and TinyOS platform. The design and preliminary performance micro-benchmarks of Agilla were reported in [14].

In this paper, we present an in-depth case study of Agilla using a fire tracking application. In this application, mobile agents are deployed to dynamically form and maintain a perimeter around a fire as it spreads through a network comprised of 26 MICA2 motes. The fire itself is modeled using special fire agents that epidemically spread throughout the network. This paper makes three primary contributions. First, it demonstrates how a mobile agent middleware can be used to facilitate the development and deployment of a non-trivial application. Using Agilla, we were able to rapidly create and deploy the entire fire tracking application by injecting 47-byte fire agents and a 100-byte tracker agent. Second, we present a set of application-level performance results that demonstrate the reliability and efficiency of mobile agents and tuple spaces in a highly dynamic application. Finally, we provide new insights into, and lessons about, mobile agent programming techniques for WSNs. To the best of our knowledge, this paper provides the first case study of mobile agents using a real application on a physical WSN test bed.

The remainder of this paper is organized as follows. Section II provides an overview of Agilla. Section III describes the fire tracking application which is used for the case study. Section IV contains the experiments performed to evaluate the fire tracking application's performance. Section V discusses lessons learned while building the fire tracking application. Conclusions appear in Section VI.

0-7803-9202-7/05/$20.00 (C) 2005 IEEE

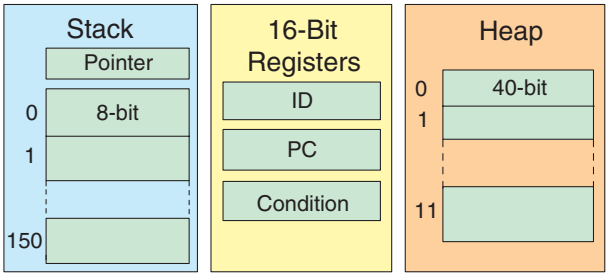

Fig. 1. The mobile agent architecture

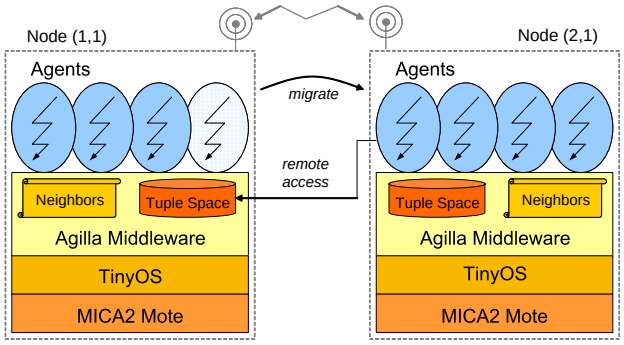

Fig. 2. The Agilla model

II. Agilla Overview

This section provides a brief overview of Agilla. For more details, see [14]. Agilla applications consist of mobile agents that can move and clone themselves performing application-specific tasks. The agents coordinate through tuple spaces and address nodes by their location. An acquaintance list identifying one-hop neighbors is maintained on each node. Agilla provides high-level instructions that allow agents to perform complex tasks.

The mobile agent architecture is shown in Figure 1. 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). A full listing of the instructions are given here [15]. Agilla agents are currently written using an assembly-like language similar to that of Maté. During this case study, we learned that programming in this assembly-like language was very tedious and error prone. To address this, we are currently working on a higher-level declarative language that is described in more detail in Section V.

While running, an agent can move or clone to other nodes using the instructions `smove`, `wmove`, `sclone`, and `wclone`. The first letter indicates whether the operation is *weak* or *strong*. A weak migration transfers only the code, all execution state is reset and the agent resumes running from the beginning when it arrives at the destination. A strong migration transfers everything, meaning an agent resumes execution where it left off. Our experience with the fire tracking application shows that the choice between strong vs. weak migration significantly affects the application overhead, performance, and reliability.

The Agilla model is shown in Figure 2. Each node currently supports up to four agents. Agilla automatically handles the context switching that allows the agents to run concurrently and independently. Instead of addressing destinations by their ID, Agilla addresses them by their geographic location, which is assumed to be known. This is reasonable since most WSN applications must know

```
1 BEGIN   pushn fir
2         pushc 1
3         out
4         halt
```

(a) agent at (1,1)

```
1 BEGIN   pusht string
2         pushc 1
3         pushloc 1 1
4         rrdp
5         rjumpc FOUND
6         halt
7 FOUND   pushc 1
8         putled
9         halt
```

(b) agent at (2,1)

Fig. 3. An example of an agent (a) inserting a tuple into node (1,1)'s tuple space, and (b) remotely reading it

their spatial orientation to make sense of the sensor data they collect. Sensors can get their location through GPS, or any number of other localization schemes [16]–[18]. Addressing nodes by location enables Agilla primitives to be easily extended to operate over geographic regions, and geographic routing to be used for multi-hop interactions.

Agilla provides two components that facilitate inter-agent coordination: a Linda-like tuple space [19], and an acquaintance list. Both are maintained on each node by the middleware. A node's tuple space is shared by local agents and is remotely accessible. Tuple spaces offer a decoupled style of communication where one agent can insert a tuple, and another can later read or remove it using pattern matching via a template. For example, consider the agents shown in Figure 3. Agent *a* in Figure 3 is located on node (1,1). It pushes a tuple containing the string "fir" onto the stack (lines 1-2), and inserts it into the local tuple space (line 3) before halting. When an agent halts, its resources are deallocated, but its tuples remain in the tuple space. Agent *b*, located on node (2,1) reads the tuple inserted by agent *a*. Lines 1-2 push a template that matches a tuple with one field of type string. Line 3 pushes the location (1,1) to indicate the target of the `rrdp` instruction executed by line 4. `rrdp` is a remote probing read instruction that probes a remote tuple space for a matching tuple. If a match is found, it pushes a copy of it onto the stack and sets the condition code to 1. Line 5 branches based on the condition code. The agent halts if no match is found, and turns on the red LED otherwise. Tuple spaces decouple agent communication, allowing each agent to run autonomously. To avoid polling, Agilla augments the tuple space with reactions, which allow an agent to react to the presence of a tuple matching a particular template. This is a technique used by other tuple space-based middleware [20]–[23].

Note that Agilla does *not* support a global tuple space that spans across multiple nodes primarily due to bandwidth and energy constraints. Instead, it supports *local* tuple spaces where each node maintains a distinct and separate tuple space. Special instructions are provided for agents to access tuple spaces located on remote nodes. These instructions rely on location-addressed 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. During this study, however, we learned that sequentially accessing each neighbors' tuple space required lots of code and entailed significant overhead. To address this, we added a `rrdpg` (remote probing group read) instruction that uses multicast to query the tuple spaces of all one hop neighbors. Since `rrdpg` operates over one hop, it does not saturate the bandwidth of the entire network.

Agilla also maintains an acquaintance list on every node. This list contains the location of all one-hop neighbors. Local agents can access it by executing special `getnbr`, `numnbr`, and `randnbr`

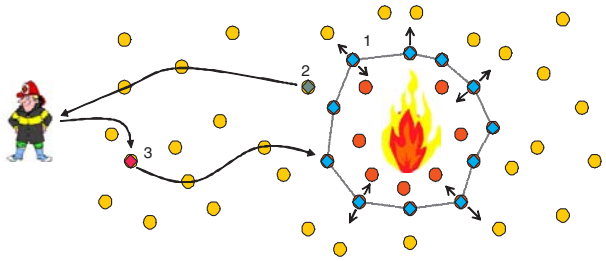

Fig. 4. An overview of the fire tracking application. Tracker agents (1) form a perimeter around the fire and notify a fire fighter (2). The fire fighter injects a guidance agent (3) that directs him to the fire.

```
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. 5. The static fire 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. 6. The reaction registered by the fire tracker

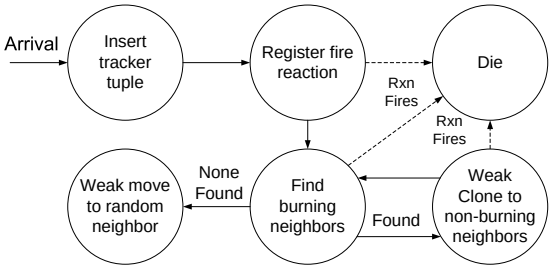

Fig. 7. The life cycle of a fire tracker agent

instructions. While Agilla could use a beacon-based approach or other middleware [24] to maintain this list, we use a grid topology in our prototype and take advantage of it to populate the list.

There are many challenges that Agilla must address. First, WSNs have severely limited resources. For example, the MICA2 motes have a mere 128KB of instruction and 4KB of data memory and run a relatively slow 8MHz Atmel 128 microprocessor. Furthermore, TinyOS does not provide dynamic memory management, meaning all data memory must be statically allocated. Hence, Agilla implements its own dynamic memory manager for agent instructions and the tuple spaces. Second, the small physical size, reliance on batteries, and embedded installation of the motes result in the use of unreliable low-bandwidth (38.4 Kbaud) wireless links. Mobile agents are particularly susceptible to message loss because it interferes with the agent's ability to migrate and clone. To minimize the impact of message loss, agents are divided into tiny packets (less than 41 bytes), are migrated a single hop at a time, and utilize timeouts and retransmits. Since this introduces store-and-forward delay, it is only used while migrating or cloning agents, not for remote tuple space operations. Remote tuple space operations are intentionally non-blocking, preventing an agent from deadlocking due to message loss.

III. The Fire Tracking Application

The fire tracking application is shown in Figure 4. A fire ignites in a region within the sensor network. As the fire spreads, tracker agents swarm around it repeatedly cloning themselves to form a perimeter. Once the perimeter is formed, they notify a fire fighter, who injects a guidance agent leading him along a safe path to the fire. This case study focuses on the tracker agents. The guidance agent and its safe-route discovery algorithm are left as future work. We model the fire by inserting tuples containing the string "fir" into the nodes that are supposed to be burning. The tracker agent can then use remote tuple space operations (e.g., `rrdp` or `rrdpg`) to detect fire.

We use two types of fire agents for modeling fire: *static* and *dynamic*. Static fire agents simply insert a fire tuple into the local tuple space, and then sit in a loop blinking the red LED. By blinking the red LED, we can visually determine the state of the network. The code is shown in Figure 5. Lines 1-3 insert the fire tuple into the local tuple space, while lines 4-8 blink the red LED. The static fire agent is used to create fires of different shapes for the tracking agent to form a perimeter around. We use it to isolate the detection and tracking phase and to evaluate how quickly the tracking agent can form a perimeter upon detecting fire.

The dynamic fire agent models a fire that epidemically spreads throughout the network. It is implemented in a mere 47 bytes of instructions, and is available in [15]. The dynamic fire agent works by inserting a fire tuple upon arrival, and then blinking the red LED a certain number of times. It then clones itself onto a random non-burning neighbor and repeats the blinking. The process of blinking and cloning to a random neighbor is repeated indefinitely, ensuring that every node will eventually be on fire. Since it takes 0.25 seconds to blink the LED, we can control the rate at which the fire spreads by changing the number of times it blinks between clones.

The fire tracker agent discovers and forms a perimeter around the fire. It dies if the node it is on catches on fire. This is done by registering a reaction that kills the agent when a fire tuple is inserted into the local tuple space. The code that registers this reaction is shown in Figure 6. Lines 1-2 pushes a template containing the string "fir" onto the stack. Line 3 pushes the address of the reaction's call-back function onto the stack, and Line 4 registers the reaction. Lines 6-11 define the reaction's call-back function, which is executed when the reaction fires. When the reaction fires, the tracker agent turns of all LEDs (lines 6-7), removes its tracker tuple (lines 8-10), and then halts. The tracker tuple is inserted into the tuple space by the tracker agent when it arrives. This tuple is used by other tracker agents to determine the integrity of the perimeter by checking whether a neighboring tracker agent is still present. If the fire breaches the perimeter, the tracker agents next to the breach must re-form the perimeter as quickly as possible by cloning themselves. Any persistent breach of the perimeter is considered a failure.

The life cycle of a tracker agent is shown in Figure 7. It works by repeatedly checking whether any of its neighbors are on fire. If none are, it performs a weak move to a random neighbor and repeats the process. If a neighbor is on fire, it enters a tracking mode. While in tracking mode, the fire tracker lights up its green LED and repeats

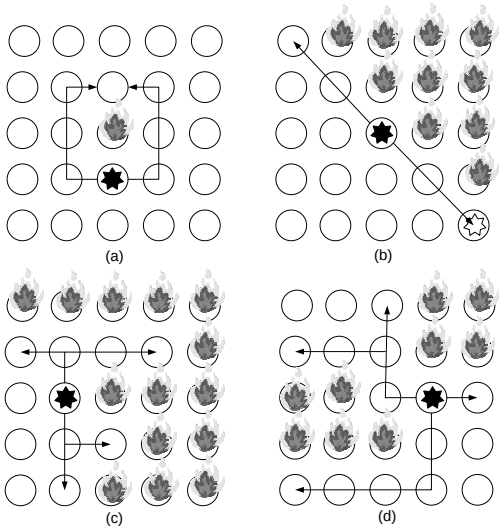

Fig. 8. The static fire tests

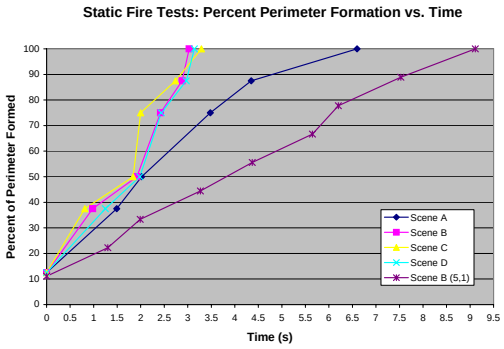

Fig. 9. The rate of perimeter formation around a static fire.

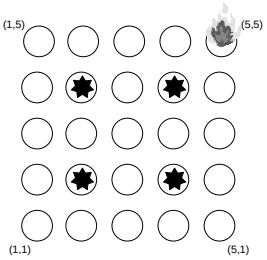

Fig. 10. The initial setting of the dynamic fire tests

the following loop: It first determines the locations of all neighbors that are on fire. Then for each non-burning neighbor that is within a certain distance of the fire and not already running a tracker agent, it performs a weak clone (`wclone`) to it. This process is repeated indefinitely until the fire dies. The periodic checking of neighbors in danger of catching fire allows the tracker agent to adjust the perimeter as the fire spreads.

The fire tracker agent was implemented in 101 bytes of code. It was intentionally designed to only use weak migration instructions to minimize overhead. While implementing the tracker agent, we discovered two useful instructions that reduced the size of the agent by 58%: `rrdpg` and `vicinity`. These instructions store their results in the heap and are discussed further in Section V.

IV. Performance Benchmarks

To evaluate our fire tracking agent, we tested its performance in a WSN consisting of 26 MICA2 motes arranged in a 5 x 5 grid (one mote serves as a separate base station). By arranging the motes in row-major order, we were able to calculate the node's (x, y) location based on its address. To create a multi-hop network in our lab's limited space, we modified the TinyOS network stack to filter out all messages except those from immediate horizontal, vertical, and diagonal neighbors based on the grid topology. Since our network is physically single-hop, our results reflect worse-case scenarios due to an increased likelihood of wireless collisions. The base station is located at $(0, 0)$ and can communicate with any node in the first row.

Two types of tests were run. The first used static fire agents to determine how fast the perimeter can be formed around a static fire of various sizes and shapes. The second set of experiments use the dynamic fire agents to determine how well the tracker agents can adjust the perimeter as the fire spreads. Since this is an application case study, we do not present micro-benchmarks evaluating the latency of each individual instruction. These experiments were conducted previously and are presented in [14]. They show that one-hop remote tuple space operations take around $50ms$, while migration operations take around $250ms$. Local instructions range from $50\mu s$ to $450\mu s$.

For the static fire tests, we initialize our WSN by injecting static fire agents onto certain nodes to form fires of various shapes and sizes. A fire tracker agent is then injected onto a node next to the fire. Note that we do not inject the tracker on a distant node because then it will randomly move around until it finds a neighbor that is on fire. It is not interesting to benchmark the random walk phase because an agent's performance relies entirely on how lucky it is at moving to a node next to the fire. By injecting the tracking agent next to the fire, we isolate the discovery phase from the tracking phase allowing us to focus these experiments on the efficiency of perimeter formation. We ran tests on several different fires, as shown in Figure 8. The node on which we initially injected the detector agent is marked with a black star. The arrows indicate where the detector must clone itself to form the perimeter. Note that in test b, node (1,5) also has a star. This is because our tests revealed that the starting location of the tracker has a significant impact on the efficiency, and will be described later in this section.

To capture the progress of the perimeter formation, we used a SONY DCR-TRV18 digital camcorder to record each run. The camcorder records video at 30fps with enough color resolution to tell which LEDs are lit up. By performing off-line analysis of the video, we are able to correlate the status of the detector agent by looking at when the green LEDs of the perimeter nodes turn on. The videos of our experiments are available at [15]. Recall that the green LED lights up when the detector enters the tracking phase. We recorded the time at which the green LEDs of the perimeter nodes lit up, and plotted the percent perimeter formation over time. The results are shown in Figure 9. Notice that in most cases the perimeter is formed within 3 seconds. The only scenario that took longer is scene *A*. The reason why scene A took longer is because its configuration contains areas that prevent multiple agents from spreading in parallel. For example, when a detector is at node (2,2), it is the only agent that can clone to (2,3). To test this theory, we re-ran scenario *B* with the fire detector initialized at node (5,1) which presents many instances where only one agent can clone to advance the perimeter. The results, shown on Figure 9, clearly show how the initial point of fire tracking has a significant impact on the speed at which the perimeter is formed.

To evaluate the detector's ability to maintain a perimeter around a

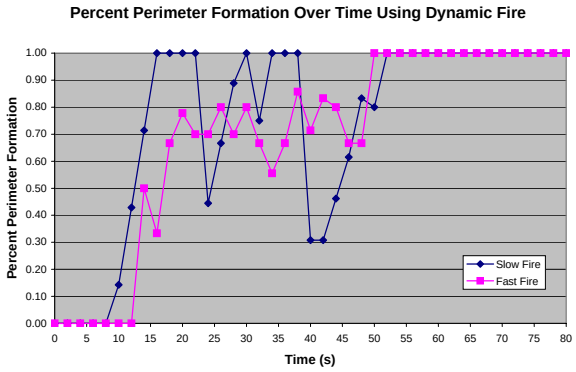

Fig. 11. The rate of perimeter formation around a dynamic fire.

spreading fire, we inject four fire tracking agents into the network at the positions marked with a black star in Figure 10, and then inject a dynamic fire agent into node (5,5). We run two tests: one with a slow fire agent, another with a fast one. Recall that the speed of propagation is controlled by the number of times the agent blinks its red LED between cloning operations. The slow agent blinks 28 times, whereas the fast one blinks 20 times. This translates to a maximum of one cloning every 7 and 5 seconds, respectively.

We used the digital camcorder to record each run of the dynamic fire tests, then sampled every other second to see what percentage of the perimeter is formed. Note that since the fire keeps spreading, the percent formed can vary non-uniformly over time. The results are shown in Figure 11. They show that the fire tracker does a reasonable job maintaining a perimeter around the slow fire, but has difficulty with the fast fire. In the fast experiment, the fire agent spreads so quickly that it cuts off a portion of the network preventing the detector agent from forming a full perimeter. The reason why both converge to 100% is because as the fire spreads, the network eventually becomes saturated with an agent on every node.

V. Lessons Learned

The most important lesson is that care must be taken when writing mobile agents. The choice between using strong and weak migration operations has a significant impact on the complexity and efficiency of the mobile agent code. Initial versions of the tracking agent used strong migrations because of the way we designed the agent. Specifically, the agent code was split into two modules to reflect the two modes of operation, e.g., fire detection and fire tracking. After writing the agent, we ran into problems with the agent size being too big to fit under the severe memory constraints of Mica2 motes. After several rounds of optimizing the two-module agent, we realized that the agent's mode could be inferred based on whether there were any neighboring nodes on fire. This allowed us to merge the two blocks of code reducing the agent's size, and use weak migration operations since an agent no longer needed to remember which mode it was in across migrations.

We also learned that the instruction set can significantly reduce agent code size, both in terms of what they do and how they store their results. Two instructions we added during this case study were `rrdpg` and `vicinity`. `rrdpg` takes a template and determines the address of all neighbors who have a tuple matching this template. Without it, each neighbor would have to be queried individually, a process that takes 33-bytes of code and many more message transmissions. Unlike other instructions, `rrdpg` stores the results on the heap because they are often used numerous times. This is a

```
void main() {
    String f = "fir";
    Tuple t;
    t.addField(f);
    out(t); // insert fire tuple;
    while(true) {
        ledToggle(RED);
        sleep(100); // sleep 100ms
    }
}
```

Fig. 12. The static fire agent written in our higher-level language

tradeoff since it prevents the agent from using the heap for other purposes. We chose this because manually transferring the results from the operand stack to the heap takes 24-bytes of code (1/4 of the current tracker agent). `vicinity` operates on the results of `rrdpg`. It takes a location and determines whether any of the locations returned by the `rrdpg` are within 1.4 non-diagonal grid hops of it. This instruction is useful when determining whether a neighbor should be part of the perimeter. The introduction of this instruction saved an additional 30 bytes of code within the fire tracker agent.

An agent's size is clearly dependent on the instructions available. We were only able to achieve 101 byte tracker agents and 47 byte fire agents by introducing the special instructions mentioned above. This highlights the need for a middleware architecture that allows the user to customize the instruction set. As it stands, Agilla's architecture allows users to easily add and remove instructions off-line, prior to deployment. Once Agilla is compiled and installed on a mote, new instructions cannot be added. As part of our future work, we plan on investigating how new instructions can be added post-deployment. One possibility is to use Deluge or XNP to update Agilla with new instructions on an infrequent basis.

Finally, we learned that programming in an assembly-like language is an extremely tedious and error-prone task. It is clearly not scalable to complex applications requiring large mobile agents. While developing the tracker agent, for example, we repeatedly ran into problems with maximum branch distances and overriding condition codes. Maté, the virtual machine on which Agilla is based, suffered similar difficulties [25]. Since one of Agilla's goals, in addition to increasing application and network flexibility, is to enable rapid application development, we are currently working on a higher level declarative language that will compile into agilla byte code. The severe resource constraints and need for supporting a dynamic instruction set makes this effort non-trivial. Our current prototype is able to compile simple applications that blink the LEDs and perform simple tuple space operations, like the static fire agent shown in Figure 12. Our goal is to be able to program any agent behavior using this language.

VI. Conclusions

Throughout this study, we have demonstrated that Agilla can be used to deploy complex applications in wireless sensor networks. We have also demonstrated how multiple applications can simultaneously share a network (e.g., a fire-simulation application and a tracker application). We presented a case study of how mobile agents can be used to program a WSN for tracking fire. We showed that mobile agents and tuple space-based communication are feasible even in highly restrictive environments, and that these abstractions can be used to increase network flexibility. Through experiments on a 26 node MICA2 network, we demonstrated that 101 byte tracker agents were able to quickly form a perimeter around a static fire, and that the efficiency depends a great deal on the degree of agent parallelism.

We also showed that the fire tracker agents can maintain a perimeter around a dynamic fire as it spreads throughout a network. Many lessons were learned including careful agent design, the importance of choosing an appropriate instruction set, and the need for a higher-level agent programming language. Our experience with developing this application led to the addition of several instructions, enabling Agilla to provide a better foundation for rapidly developing flexible applications for WSNs.

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] D. Culler, D. Estrin, and M. Srivastava, "Overview of sensor networks," *IEEE Computer*, vol. 37, no. 8, pp. 41–49, 2004.

[2] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 13–24.

[3] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton, "Codeblue: An ad hoc sensor network infrastructure for emergency medical care," in *Proceedings of the MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, June 2004.

[4] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh, "An energy-efficient surveillance system using wireless sensor networks," in *Proceedings of the MobiSys 2004*, June 2004.

[5] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet," in *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[6] 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*, 2000, pp. 93–104. [Online]. Available: citeseer.ist.psu.edu/382595.html

[7] 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. [Online]. Available: citeseer.ist.psu.edu/levis02mate.html

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

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

[10] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 81–94.

[11] 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.

[12] T. Liu and M. Martonosi, "Impala: A middleware system for managing autonomic, parallel sensor systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[13] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode, "Smart messages: A distributed computing platform for networks of embedded systems," *To Appear in the Special Issue on Mobile and Pervasive Computing, the Computer Journal*, 2004.

[14] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

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

[16] N. Priyantha, A. Chakraborty, and H. Balakrishnan, "The cricket location-support system," in *Mobile Computing and Networking*, 2000, pp. 32–43. [Online]. Available: citeseer.ist.psu.edu/priyantha00cricket.html

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

[18] D. Moore, J. Leonard, D. Rus, and S. Teller, "Robust distributed network localization with noisy range measurements." in *The Second ACM Conference on Embedded Networked Sensor Systems (Sensys 04)*, November 2004.

[19] D. Gelernter, "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.

[20] C.-L. Fok, G.-C. Roman, and G. Hackmann, "A Lightweight Coordination Middleware for Mobile Computing," in *Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004)*, ser. Lecture Notes in Computer Science, R. DeNicola, G. Ferrari, and G. Meredith, Eds., no. 2949. Springer-Verlag, 2004, pp. 135–151.

[21] 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*, Nov. 2002, pp. 21–30.

[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*, April 2001, pp. 524–533.

[23] G. Cabri, L. Leonardi, and F. Zambonelli, "Reactive tuple spaces for mobile agent coordination," *Lecture Notes in Computer Science*, vol. 1477, pp. 237–252, 1998. [Online]. Available: citeseer.ist.psu.edu/cabri98reactive.html

[24] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004, pp. 99–110.

[25] P. Levis, D. Gay, and D. Culler, "Bridging the gap: Programming sensor networks with application specific virtual machines," University of California, Berkeley, Tech. Rep. UCB//CSD-04-1343, 2004.