Report Number: WUCSE-2006-16

2006-01-01

Agilla: A Mobile Agent Middleware for Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu

Agilla is a mobile agent middleware for sensor networks. Mobile agents are special processes that can migrate across sensors. They increase network flexibility by enabling active in-network reprogramming. Neighbor lists and tuple spaces are used for agent coordination. Agilla was originally implemented on Mica2 motes, but has been ported to other platforms. Its Mica2 implementation consumes 41.6KB of code and 3.59KB of data memory. Agents can move five hops in less than 1.1s with over 92% success. Agilla was used to develop multiple applications related to fire detection and tracking, cargo container monitoring, and robot navigation.
... Read complete abstract on page 2.

Agilla: A Mobile Agent Middleware for Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu

Complete Abstract:

Agilla is a mobile agent middleware for sensor networks. Mobile agents are special processes that can migrate across sensors. They increase network flexibility by enabling active in-network reprogramming. Neighbor lists and tuple spaces are used for agent coordination. Agilla was originally implemented on Mica2 motes, but has been ported to other platforms. Its Mica2 implementation consumes 41.6KB of code and 3.59KB of data memory. Agents can move five hops in less than 1.1s with over 92% success. Agilla was used to develop multiple applications related to fire detection and tracking, cargo container monitoring, and robot navigation.

2006-16

Agilla: A Mobile Agent Middleware for Sensor Networks

Authors: Chien-Liang Fok, Gruia-Catalin Roman, Chenyang Lu,

Corresponding Author: liangfok@wustl.edu

Web Page: http://mobilab.wustl.edu/projects/agilla/index.html

Abstract: Agilla is a mobile agent middleware for sensor networks. Mobile agents are special processes that can migrate across sensors. They increase network flexibility by enabling active in-network reprogramming. Neighbor lists and tuple spaces are used for agent coordination. Agilla was originally implemented on Mica2 motes, but has been ported to other platforms. Its Mica2 implementation consumes 41.6KB of code and 3.59KB of data memory. Agents can move five hops in less than 1.1s with over 92% success. Agilla was used to develop multiple applications related to fire detection and tracking, cargo container monitoring, and robot navigation.

Type of Report: Other

Agilla: A Mobile Agent Middleware for Sensor Networks

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

Agilla is a mobile agent middleware for sensor networks. Mobile agents are special processes that can migrate across sensors. They increase network flexibility by enabling active in-network reprogramming. Neighbor lists and tuple spaces are used for agent coordination. Agilla was originally implemented on Mica2 motes, but has been ported to other platforms. Its Mica2 implementation consumes 41.6KB of code and 3.59KB of data memory. Agents can move five hops in less than 1.1s with over 92% success. Agilla was used to develop multiple applications related to fire detection and tracking, cargo container monitoring, and robot navigation.

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

General Terms: Agent, Design, Reliability, Algorithms

Additional Key Words and Phrases: Mobile Agent, Wireless Sensor Network, Middleware

1. INTRODUCTION

Wireless sensor networks (WSNs) contain a multitude of devices that integrate sensors, processors, memories, and network interfaces. Since they are embedded, each device is typically small, run on batteries, and communicate over low-power links. WSNs are often ad hoc; once deployed, they are expected to autonomously form a network without infrastructure support. Depending on the application, the network may form routing trees for delivering data to base stations, or a multi-hop mesh for delivering data amongst themselves. There are many applications for WSNs.

While many WSNs have been successfully deployed [Culler et al. 2004], their utility is limited by inflexible software. Most WSN devices are programmed prior to deployment and, once deployed, can only be marginally tweaked using pre-defined parameters. Unfortunately, identifying the parameters prior to deployment is often impossible, and providing parameters that drastically change network behavior may be too expensive to implement. In addition, many applications are custom-tailored to maximize efficiency. This limits code reuse and, more importantly, reduces the

Authors' address: Department of Computer Science and Engineering, Washington University, Saint Louis, MO 63130.
Authors' emails: {fok, roman, lu}@cse.wustl.edu

possibility that an application can perform tasks it was not originally intended to do.

WSN software must be flexible because it must react to unexpected changes within the environment. Developing software that addresses every scenario that may occur is impossible. There are many situations where the application must be replaced, especially as improvements in batteries and other power sources enable longer deployment intervals. For example, suppose a WSN is initially deployed for habitat monitoring when, during a drought, fire fighters want to reprogram the network to detect fires. When a fire is detected, the fire fighters may want to reprogram the network once more with a search and rescue application. Integrating and installing all three applications at once is not flexible or scalable.

Several systems enable users to wirelessly reprogram WSNs [Hui and Culler 2004; Levis and Culler 2002; Boulis et al. 2003; Liu and Martonosi 2003; Kang et al. 2004; Han et al. 2005; Dunkels et al. 2004]. These systems, however, either flash the instruction memory, flood the entire network, or exploit simple forms of code mobility. Re-flashing the instruction memory requires the entire image to be transferred. This is inefficient and has high latency. Flooding a network with new code is undesirable because oftentimes a user is only interested in a portion of the area covered by a WSN. Installing software on devices in irrelevant locals wastes resources and reduces a network's utility by preventing relevant applications from running. The ability to install new code in a WSN increases a network's flexibility, but code mobility without state mobility (e.g., the program counter, heap, and stack) may decrease efficiency and complicate the development effort since the software will have to restart upon arrival on each device.

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 mobile agents that can migrate across nodes. Mobile agents are dynamic, localized, and intelligent. Each agent is a virtual machine with dedicated instructions and data memory. An agent can execute special instructions that allow it to interact with the environment and move or clone from one node to another while maintaining its execution state. Multiple agents can coexist on a node. The middleware maintains a neighbor list on each device that agents use to discover neighboring devices that they may migrate to. Localized tuple spaces [Gelernter 1985] facilitate context discovery and inter-agent communication while ensuring each agent remains autonomous. Tuple spaces offer a shared memory model where the datum is a `tuple` that is accessed via pattern matching using `templates`. This allows one agent to insert a tuple containing a sensor reading and another to later retrieve it without the two knowing each other or being collocated, 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 the focus from parallel distributed algorithms to the sequential behavior of an individual agent. 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 repeat-

edly migrating to the node that best detects it. Finally, by allowing in-network reprogramming and multiple users, Agilla transforms WSNs into general-purpose computing platforms that are usable by a larger community.

This paper makes four contributions. First, it explores the benefits of using mobile agents, neighbor lists, 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, neighbor lists, and tuple spaces in existing WSNs through middleware implementation. 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 development in 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 necessary to cope with limited resources and an unreliable network. Section 4 presents the experimental results on Agilla's performance in terms of both micro and macro-benchmarks. Section 5 contains several case studies that illustrate how Agilla simplifies programming and increases network flexibility. Section 6 presents recent extensions made to Agilla based on lessons learned from these case studies. Section 7 discusses related work. The paper ends with conclusions in Section 8.

2. MODEL

This section presents a motivating example followed by Agilla's model.

2.1 Motivating Example

In the remote arid forests of central Arizona, lighting ignites a fire that spreads with the prevailing winds. The remoteness of the region would allow the fire to rage out of control. Fortunately, the USDA Forest Service recognized this area as highly incendiary and pre-deployed a WSN for detecting fire. As the fire grows, nearby sensors detect it and spawn tracking agents that swarm around the fire collecting real-time information about the exact location of the flames. The tracking agents form a dynamic perimeter jumping away as the fire approaches, 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 that the heat can be felt from miles away.

Upon arrival, the fire fighters' first priority is to evacuate the area. They inject search-and-rescue agents that spread and coordinate with the tracking agents to scour the region 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 control its movements preventing it from approaching populated areas where property can be damaged and people injured.

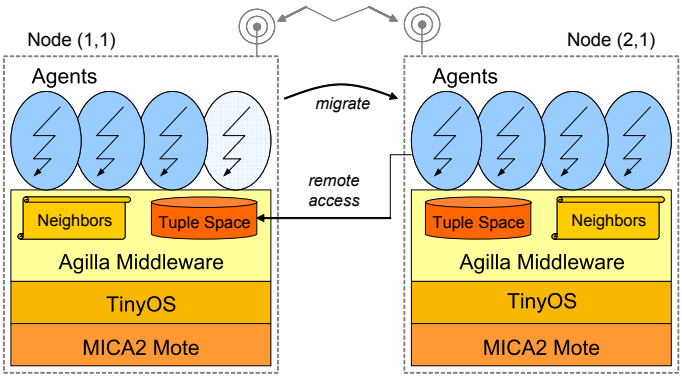

Fig. 1. The Agilla model

Once the fire has died, the tracking agents also die leaving only small fire detection agents. As new nodes are deployed to replace those that were destroyed, the detection agents migrate onto them filling any voids in the fire detection coverage. The minuscule resource consumed by the detection agents allow other applications to run, which biologists exploit by injecting habitat monitoring agents for learning about coyotes.

2.2 The Agilla Model

The motivating example presented in Section 2.1 illustrates that WSNs must be flexible because of changing user requirements and the need to adapt to a highly dynamic environment. In the example, several applications must execute within the same WSN. Most of them were not deployed until after the fire exists and the user requirements are known. All applications must adapt to contextual changes like the fire's location. Achieving this level of flexibility is difficult. Agilla is shown to simplify flexible application development by providing mobile agents as the basic unit of execution. Its model is now presented.

Agilla's 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 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 mechanism that allows them to communicate. Agilla provides this through localized tuple spaces owned by nodes and shared by the agents. 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 may contain wild cards that match by type. To remove a tuple from a tuple space, an agent must 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. If there are multiple tuples that match a template, one is chosen non-deterministically.

Tuple spaces provide a high level of decoupling that ensures agent autonomy. They also provide a convenient way for agents to discover their context. For example, since each node may have different sensors, Agilla inserts special tuples indicating what type of sensors are available, e.g., if a node has a thermometer, Agilla would insert a "temperature tuple" into its tuple space. Other context information stored in the tuple space includes 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. These tradeoffs are discussed further in Section 3.

Agilla tuple spaces provide operations `out`, `in`, `rd`, `inp` (probing `in`), and `rdp` (probing `rd`). 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, respectively. If a match does not exist, the executing agent blocks until one does. `inp` and `rdp` are the same as `in` and `rd` except that they do not block.

Like many other tuple space-based systems [Fok et al. 2004; Julien and Roman 2002; Murphy et al. 2001; Cabri et al. 1998], Agilla allows agents to register *reactions*. Reactions provide interrupt semantics and consist of a template and a *call-back function*, which is a block of code that is executed when the reaction *fires*. A reaction fires when a tuple matching the reaction's template appears in the local tuple space. When this occurs, a copy of the tuple is pushed onto the stack, and the agent's program counter is changed to point to the first instruction of the reaction's call-back function. Reactions allow an agent to tell Agilla that it is interested in tuples that match a particular template. When a 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. An agent carries its reactions across migrations.

Agilla reactions are necessarily weak due to the limited resources within a WSN. For example, the reactions are strictly local; an agent can only react to tuples in the local tuple space. This differs from LIME [Murphy et al. 2001] and Limone [Fok et al. 2004], which allow reactions to propagate and operate over multiple hosts. By limiting the reaction to the local tuple space, the host engagement and disengaging protocol is simplified, and the middleware does not require buffers for holding reactions belonging to remote agents. In addition, Agilla's reaction call-back functions are *not* executed atomically as they are in LIME and Limone. They are treated like regular instructions that may include blocking operations and be preempted by another reaction. This eliminates pending reaction executions and the buffers necessary to hold them. Finally, to reduce state, when a reaction is registered, it will only react once to the pre-existing tuples in the tuple space; if there are several matches only one will fire the reaction. It is up to the programmer to check whether

additional matches exist at the end of the reaction call-back function. However, once the reaction is registered, it will react to every new matching tuple placed into the tuple space. This reduces middleware complexity and overhead by eliminating buffers for holding pending reaction executions, and makes stack overflows caused by multiple rapid firings, each pushing a tuple onto the agent's operand stack, less probable.

Tuple spaces allow agents to communicate in a decoupled fashion. For example, suppose there is a fire detection and habitat monitoring agent residing on the same node when a fire is detected. The fire detection agent inserts a fire tuple to indicate fire and activates a tracking agent before dying. Assuming this fire tuple remains in the tuple space, the habitat monitoring agent will eventually react to it and voluntarily die to free additional resources. Notice how the fire detection agent need not know who received the fire tuple, the sending and reception can occur at different times, and reception can occur even if the sender no longer exists. This spatial and temporal decoupling ensures that each agent remains autonomous, simplifying programming.

One alternative to using tuple spaces is message passing. Message passing does not decouple agents since it requires that the sender know the receiver, and both must be present for communication to occur. Furthermore, the receiver must either block waiting for the message to arrive, continuously poll for it, or use an active-messaging system like that of TinyOS [Hill et al. 2000]. The first two options are inefficient. The third option tightly couples the sender and receiver since they have to agree on active message numbers. Message passing is less flexible than tuple spaces because it introduces dependencies among agents.

Agilla agents need to coordinate across nodes. For example, in the motivating example, the tracking agents residing on different hosts need to coordinate to ensure the perimeter is not breached. Agilla allows agents to remotely coordinate by providing *remote tuple space operations*. They include `rout` (remote `out`), `rinp` (remote probing `in`), `rrdp` (remote probing `rd`), `routg` (remote group `out`), and `rrdpg` (remote group probing `rd`). The first three are analogous to `out`, `inp`, and `rdp` except they take an additional location parameter that specifies on which node to perform the operation. `rrdpg` searches all nodes within a one-hop range for matching tuples, while `routg` inserts a tuple into every tuple space within a one-hop range. Only non-blocking operations are provided to prevent an agent from blocking forever due to message loss or network topology changes. In the example above, the tracking agents would periodically perform `rrdp` or `rrdpg` to ensure neighboring tracking agents remain alive.

Note that Agilla does *not* support federated tuple spaces that span multiple nodes à la LIME [Murphy et al. 2001] because doing so requires employing transactions to ensure tuple space consistency and operation atomicity. Transactions limit scalability and rely on assumptions about network reliability that are unrealistic for WSNs [Carbunar et al. 2004]. Instead, Agilla supports *local* tuple spaces where each node maintains a distinct and separate tuple space. Most remote tuple space instructions rely on unicast communication with the specific node hosting the tuple space. Hence, a unicast remote tuple space operation entails the transmission of only two messages, a request and a reply, and is scalable to networks of any size.

Operations that involve multiple nodes use broadcast and restrict flooding to a single hop. Since wireless is a broadcast medium, this does not impose additional overhead and is, thus, also scalable.

Agilla assumes each node knows its physical location. This is reasonable since sensor data without its originating location is often meaningless. Nodes may acquire their location through GPS or any number of localization techniques [Hightower and Borriello 2001]. Our prototype integrates Agilla with the Cricket indoor localization system [Priyantha et al. 2000]. Location-awareness allows Agilla to use geographic routing to support agent interactions that span multiple hops.

Since many WSN applications are localized [Qi et al. 2002; Intanagonwiwat et al. 2000], Agilla performs one-hop neighbor discovery using beacons. The one-hop neighbor information is stored in a neighbor list that is updated by Agilla. Agents can access this list using instructions `numnbr` and `getnbr`. `numnbr` returns the size of the neighbor list, while `getnbr` fetches a specific neighbor's address from the list.

An Agilla network is deployed without any application installed. Agents implementing application behavior can later be injected, effectively reprogramming the network. An agent's life cycle begins when it is either injected into the network, or cloned from another agent already in the network. An agent contains its own instructions, 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`. The first letter specifies whether the operation is *weak* or *strong*. In a weak operation, only the code is transferred. The program counter, heap, stack, and reactions are reset and the agent resumes running from the beginning. In a strong operation, everything is transferred and the agent resumes execution where it left off. Naturally, strong migrations simplify programming by reducing a distributed application into a linear program. However, they require more state to be transferred and thus impose higher overhead. Through geographic routing, an agent can move or clone itself to any node regardless of the number of hops away. 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 reclaim its resources and use them for other agents. An agent dies by executing `halt`.

Another reason why nodes should be location-aware is because 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` 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. This error is defined at compile-time and is a function of the network density. It is chosen such that when an agent migrates to a particular location, there is at least one node within ϵ of it. By using location as addresses, Agilla primitives can be easily generalized to enable operations on a region. For example,

```
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
```

Fig. 2. The **FireTracker** agent

a fire detection node may need to clone itself onto *all* nodes in a geographic area, or it may wish clone itself to *at least one* node in a region.

To solidify Agilla's model, Figure 2 shows a portion of the FireTracker agent mentioned in the motivating example. Recall that FireTracker agents swarm around the fire forming a dynamic perimeter, a complex resource-consuming process. To minimize overhead, the application uses lightweight FireDetector agents during idle periods, and spawns heavier-weight FireTracker agents only when needed. Figure 2 demonstrates how a FireTracker agent is notified. When a FireTracker agent is injected, it registers a reaction sensitive to tuples containing a location and the string "fir" and waits for the reaction to fire. This is done by lines 1-6. When a FireDetector agent detects fire, it performs a `rout` using a tuple containing the string "fir" and its location on the node hosting the FireTracker agent, which reacts to the tuple by executing the code beginning at line 7. Notice that on line 8, the agent clones itself onto the node that detected the fire. Once there, it will continue to spread forming a dynamic perimeter around the fire.

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 the middleware followed by that of the agent. It ends with a discussion of the agent's instruction set.

3.1 Implementation Platform

Agilla was initially implemented and tested on Mica2 motes [Crossbow Technology 2005a]. These motes have an 8-bit 7.38MHz Atmel ATmega128L 8-bit microprocessor connected to a Chipcon CC1000 radio transceiver. The radio communicates at up to 38Kbps over a range of 100m, though the actual amounts vary substantially based on the environment [Zhao and Govindan 2003]. They have 128KB of instruction memory and 4KB of data memory. Mica2 motes are representative of a typical device used in WSNs. Agilla has since been ported to the MicaZ [Crossbow Technology 2005b], Tyndall 25mm [Tyndall National Institute 2005], and Tmote Sky [Polastre et al. 2005] motes. 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 PC. It has a MIB510 interface board that forms a bridge between the WSN and the Internet. The laptop runs a Java application

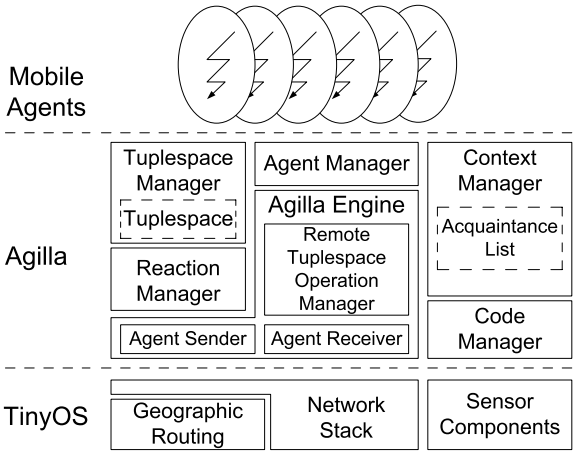

Fig. 3. Agilla's middleware architecture

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 an operating system called TinyOS [Hill et al. 2000]. TinyOS applications are divided into components arranged in a hierarchy. Dynamic memory is not provided; all variables must be statically declared. While this simplifies compile-time analysis, it also makes the meager 4KB of data memory more precious. As pointed out in [Levis and Culler 2002], TinyOS has a high learning curve. This 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 instruction memory must be re-flashed with a new image. Having a middleware that provides higher programming abstractions that hide these complexities allows programmers to quickly implement, test, and deploy their applications.

3.2 Agilla Architecture

Agilla's architecture, shown in Figure 3, is divided into three layers, the highest containing the agents that are discussed further in Section 3.3. The middle layer contains the core middleware components, while the bottom is TinyOS. The middleware consists of various managers including an agent manager, context manager, code manager, tuple space manager, and reaction manager. These managers are orchestrated by an Agilla engine. In principle, each manager may be thought of as a separate process that communicates via message passing. In reality, they are implemented as TinyOS components that use TinyOS's event and task model.

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. An agent's memory consists of its execution state and its code. The execution state consists of a program counter, heap, stack, and condition code. The code memory is divided into fixed sized blocks as described below. The agent manager allocates the minimum number of blocks that will hold

Type	Size (Bytes)	Description
Location	3	An (x,y) location
Agent ID	3	An Agent ID
String	3	A three character string
Reading	5	A sensor ID and its reading
Type	5	A data or sensor type
Value	3	A 16-bit signed value

Fig. 4. Agilla data types.

an agent's instructions. If there are insufficient blocks, the migration is aborted and the agent is resumed running at its original location with the condition code set to indicate failure. During cloning operations, the agent manager is responsible for determining whether an agent is the original or the clone, and setting the condition codes appropriately. The agent manager 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 three agents. This is easily changed at compile-time and is primarily limited by processor speed and memory availability.

Context Manager. The context manager determines the location of the host and its neighbors. 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 their neighbors' locations to adjust the perimeter. The context manager uses beacons to discover neighbors and stores the neighbor locations in an acquaintance list that is accessible via instructions `numnbrs` (count number of neighbors), `getnbr` (select a neighbor), and `randnbr` (select random neighbor). Dedicated instructions are provided because they are frequently used. The alternative would be to store the data in the tuple space and have the agent perform tuple space operations to access it. However, given the frequency of use, this would significantly increases an agent's code size.

Code Manager. Since TinyOS does not provide dynamic memory allocation, Agilla implements a dynamic memory allocator. When an agent initiates a migration, it specifies the amount of memory it requires, and the code manager allocates the minimum number of 22 byte blocks necessary to store the code. 22 byte blocks is a compromise between internal fragmentation and forward pointer overhead. It ensures that a block can fit within a TinyOS message. When agents are running, the code manager retrieves the next instruction to execute. When an agent migrates, it packages the agent's code into messages. By default, the code manager is allocated 330 bytes (15 blocks). With a few exceptions, instructions are one byte meaning an agent can have up to 330 instructions. The number of blocks and the block size is easily configurable at compile-time. Their maximum values depend on memory availability and TinyOS's message size, respectively.

Tuple Space Manager. The tuple space manager implements the non-blocking tuple space operations (e.g., `out`, `inp` and `rdp`) and manages the contents of the local tuple space. Blocking operations are implemented within the agent and are described in Section 3.4. The tuple space manager dynamically allocates memory for each tuple. By default, the tuple space is allocated 100 bytes and a tuple may contain up to 18 bytes worth of fields, whose types and sizes are shown in Figure 4.

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. 5. Messages used during migration

Currently, the tuple space is stored entirely in RAM. In the future, it may be expanded into external flash RAM. If 100 bytes is insufficient, it can be increased at compile-time, but the maximum number of agents per node may need to be decreased to ensure Agilla fits within the 4KB of RAM. A tuple is limited to 18 bytes so it can fit within a TinyOS message.

To prevent internal fragmentation and the need for forward pointers, the 100-bytes are allocated linearly. When a tuple is removed, all tuples behind it are shifted forward. While this results in more memory swapping, it is simple and enables a powerful but lightweight implementation of an enhanced reaction manager that is described in Section 6. A more in-depth investigation of efficient tuple space implementations, possibly using hash functions and external flash memory, is left as future work.

Reaction Manager. The reaction manager stores each agent's reactions in a registry. Whenever a reaction is registered, the reaction manager checks the tuple space for a match. If a match is found, the reaction manager notifies the agent manager, which pushes the tuple onto the agent's operand stack, and updates the agent's program counter to execute the reaction's code. Likewise, whenever a tuple is inserted into the tuple space, the reaction manager checks all of the reactions in the reaction registry and fires those whose templates match the tuple.

During a migration, the reaction manager packages the reactions registered by the agent into messages so they can be transferred. When an agent arrives, the reaction manager restores the agent's reactions. By default the reaction registry is allocated 130 bytes, allowing it to remember up to 5 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 8. Naturally, if an agent executes a long-running instruction like `sleep`, `sense`, or `wait`, the engine immediately switches agent context. A large body of work exists on scheduling policies that provide real-time guarantees or agent priority levels [Stallings 2001]. These advanced scheduling policies may be incorporated into the Agilla engine.

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. Minimally, 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, all of which must be transferred. If a single message is lost,

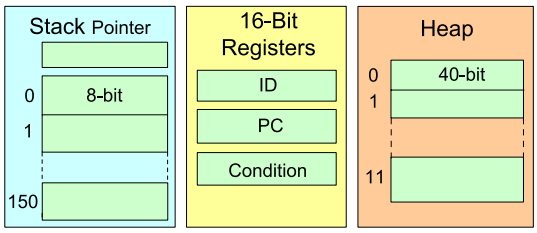

Fig. 6. The mobile agent architecture

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 rendered this unreliable. If a one-hop acknowledgement is not received within 290ms, the message is retransmitted. This repeats up to four times. If the operation stalls for over 1s, the receiver aborts. These values provide reasonable performance and reliability on the Mica2 platform, as will be shown in Section 4, but they can be easily customized at compile-time to match other platforms and environments. 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 is preferable. Consider the motivating example; it is better to have duplicate warnings that there is a fire rather than no warnings at all. If the migration succeeds, the agent's condition code is set to 1. If the operation is a strong clone, the agents often need to distinguish between the original and the clone, possibly to establish a parent-child relationship. To facilitate this, Agilla sets the parent's condition to 1, and the clone's condition to be 2. Finally, for all migration operations, it is possible that the destination node does not have enough resources to accept an agent (e.g., it may not have enough free code blocks). To distinguish between a failure due to message loss and insufficient resources, the condition code is set to 0 in the former, and 3 in the latter.

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. When the destination receives it, it performs the operation on its local tuple space and sends back the result. Unlike agent migration operations, end-to-end communication is used for remote tuple space operations and acknowledgements are not used. This is because they are usually done on nearby nodes, a request can fit in one message, and the operational semantics are not violated 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. Again, these values are easily configured at compile-time.

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 instructions to

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

Fig. 7. Noteworthy Agilla instructions

be small. Most Agilla instructions are a single byte. A few consume 3 bytes for pushing 16-bit variables onto the stack. Instructions within Agilla's extended ISA consume 2 bytes each, but these are specialized instructions that are infrequently used as will be described in Section 6. The heap is a random-access storage area that allows an agent to store 12 variables. It is accessed by instructions `getvar` and `setvar`. The heap size is customizable at compile time.

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 to each agent 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. 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 below the tuple that caused the reaction to fire. 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 [Levis and Culler 2002]. 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 [Fok 2005]. Agilla's ISA can be divided into four categories: general purpose, tuple space, and migration instructions.

General purpose instructions. Agilla's general purpose instructions are nearly

identical to those 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 behaviors 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 that 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.

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 try the `inp` or `rdp` operations. If the probe fails, the agent's context is stored in a wait queue. When a tuple is inserted, the agents in this queue are notified and can re-check 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 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. Note that since `rrdpg` may find multiple matches, its tuples are not stored on the stack because it may cause a stack overflow. Instead, the *locations* of the neighbors that possess a match are stored on the heap. 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 presents micro-benchmarks that determine the latency, reliability, and overhead of Agilla operations, followed by macro-benchmarks that evaluate Agilla using a fire detection and tracking application.

4.1 Micro-Benchmarks

The following micro-benchmarks evaluate the latency, reliability, and overhead of individual Agilla instructions. They examine the remote tuple space and agent migration operations, followed by the overhead of Agilla's local instructions that do not involve the network. The experiments are performed on a 25-mote Mica2 network arranged in a 5x5 grid as shown in Figure 8. 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 (0,0). To simulate multi-hop routing, TinyOS's network stack is modified to filter all messages except those from neighbors based on the grid topology. For geographic routing, a simple best-effort greedy forwarding algorithm is used. We tried integrating CLDP [Kim and Govindan 2005], a real geographic

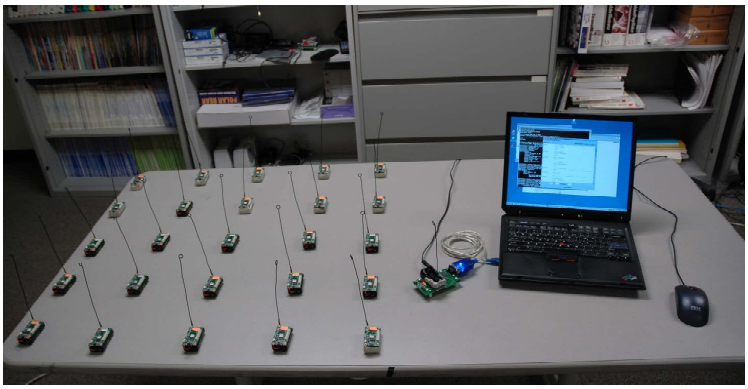

Fig. 8. A 5x5 Mica2 mote test bed

```
// 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
```

Fig. 9. The agents that test `smove` (top) and `rout` (bottom)

routing implementation, but failed due to insufficient memory.

The wireless links in WSNs are notoriously unreliable. This is problematic since agents migrate using multiple messages and the operation will fail if just one message is lost. Agilla uses two techniques to increase network reliability. First, it uses a simple protocol involving acknowledgements, timers, and retransmits. Second, whenever the operation involves multiple messages, all of them are transmitted and acknowledged one hop at a time to prevent having to re-send a message multiple hops when it is lost. Implementing these techniques adds overhead and, since the semantics of the remote tuple space operations are not violated if they fail, only the migration operations employ them.

To test reliability and overhead, the agents shown in Figure 9 are injected into node (1,1). 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 each successful execution, and the number of failures are recorded (`smove` latencies are halved to account for the double migration). The results, shown in Figures 10 and 11, indicate that both operations perform well across short distances. However, as the distance increases, the probability of a

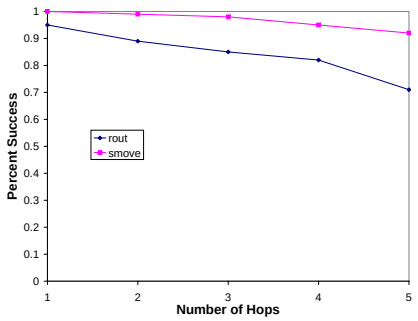

Fig. 10. The reliability of `smove` vs. `rout`

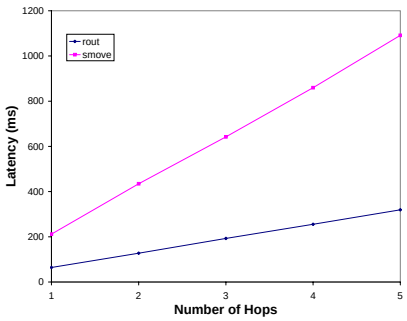

Fig. 11. The latency of `smove` vs. `rout`

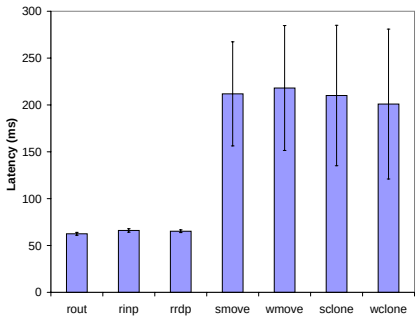

Fig. 12. The latency of remote operations.

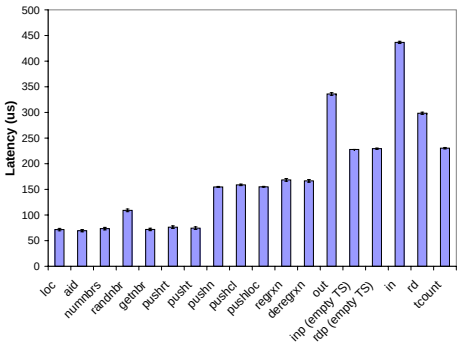

Fig. 13. The latency of local operations.

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. 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 instructions, we measured the one-hop execution time of all these instructions by timing each 100 times and finding the average. The results, shown in Figure 12, indicate that `rout` and `smove` are representative, and that the reliable agent migration instructions have significantly higher overhead than the unreliable remote tuple space operations. Note that migration operations have higher variance. This makes sense since the reliable protocol employs retransmit timers at each hop, which may result in higher latency. The results also suggest that the quickest an agent can reliably migrate is once every 0.3s. Assuming the radio range is around 50m, this means an agent can migrate across a network at 600km/h (373mph), which is sufficient for tracking many interesting events like fire.

We also benchmarked 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 13, indicate that there are three

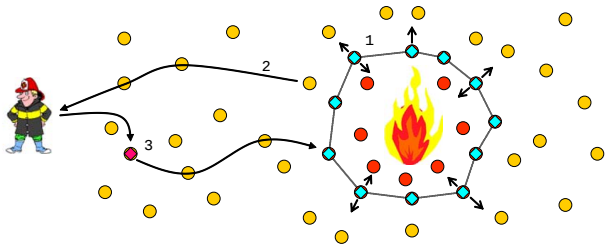

Fig. 14. An overview of the fire detection and tracking application. Initially, when a fire breaks out, detection agents send a message to a base station (1), which injects a tracker agent into the network (2). This agent migrates to the fire and repeatedly clones itself to form a perimeter (3). The perimeter is continuously adjusted based on the fire's behavior.

general classes of local operations. The first class has the least execution time and includes the `loc`, `aid`, `numnbrs`, and various `push` instructions. These instructions simply push a value onto the stack; they do not perform any calculation and take about 75μs. The second class of instructions have higher latency 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 successful blocking tuple space operations take slightly longer than the non-blocking ones. This makes sense since blocking operations actually consist of performing the non-blocking equivalent operation, a check to see whether a result was found, and blocking if none was found. Also note that `in` takes longer than `rd`, which makes sense since it requires modifying the tuple space.

Agilla can perform reliable 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 reliable operations is justified by their resilience to message loss across multiple hops. Local operations take between 60 and 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 against which to be compared. However, the latency of simpler Agilla instructions like `loc` and `aid` that execute within 100μs is comparable to corresponding Maté operations.

4.2 Macro-benchmarks: Fire Detection and Tracking

While the previous section evaluated individual Agilla instructions, this section evaluates Agilla's overall model in terms of its ability to support dynamic applications such as the one shown in Figure 14, fire detection and tracking. The fire detection and tracking application illustrates the need for a flexible middleware. A WSN is deployed in a region susceptible to wild fires. When a wild fire breaks out, its initial position and movements are unpredictable. Agilla allows the network to continuously reprogram itself using mobile agents based on the fire's behavior. In

```
1:  BEGIN   pushc temperature
2:          sense              // take a temperature reading
3:          pushcl 200
4:          cgt
5:          rjumpc FIRE        // jump to FIRE if temperature > 200
6:  SLEEP   pushc 8
7:          sleep              // sleep for 1 second
8:          rjump BEGIN        // repeat
9:  FIRE    loc
10:         pushn fir
11:         pushc 2            // tuple <"fir", location> is on the stack
12:         pushloc 1 1        // assume base station is at (1,1)
13:         rout               // send tuple <"fir", location> to base station
14:         halt               // die
```

Fig. 15. A fire detector agent

this case, light-weight fire detection agents sense the fire and notify the base station. The base station in turn injects a fire tracker agent that moves to the engulfed region and repeatedly clones itself to form a perimeter. The tracker agents continuously adjust their numbers and location to maintain a perimeter as the fire changes shape. This adaptive behavior is difficult to provide without mobile agent technology. The remainder of this section discusses how Agilla can be used to implement the detection and tracker agents, and the performance of the tracker agents.

After the WSN is deployed, enough fire detection agents are injected to achieve sensing coverage. These agents determine when a fire breaks out, and notify the base station when one does. Agilla does not provide a special instruction for detecting fire. However, it does provide tuples that describe the available sensors, and an instruction for accessing them. A basic detector agent that senses fire based on temperature is shown in Figure 15. It takes a temperature reading every second (lines 1-8), and inserts a tuple containing its location and the string "fir" into the base station's tuple space if the temperature reading is above 200 (lines 9-13). After it notifies the base station, it dies freeing its resources (line 14).

In the experiments presented later in this section, fire is modeled by inserting the string "fir" into the tuple spaces of the nodes that are on fire. The tracker agents search for these tuples using remote tuple space operations (e.g., `rrdp` and `rrdpg`) to detect fire. Two types of fire agents are used: *static* and *dynamic*. A static fire agent does not move. Its code is shown in Figure 16. Lines 1-3 insert the fire tuple, while lines 4-8 blink the red LED. By blinking the red LED, we can visually determine the state of the network. The static fire agent is used to create fires of different shapes. It serves as a baseline on how quickly the tracker agent can form a perimeter around a 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 on Agilla's website (`http://mobilab.wustl.edu/projects/agilla/index.html`). The rate at which it spreads can be set by controlling how long it resides on a node before attempting to clone itself onto other nodes.

The fire tracker agent forms a perimeter around the fire. It dies if its node catches on fire. This is done by registering a reaction that kills the agent when a fire tuple

```
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. 16. 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. 17. The reaction registered by the fire tracker

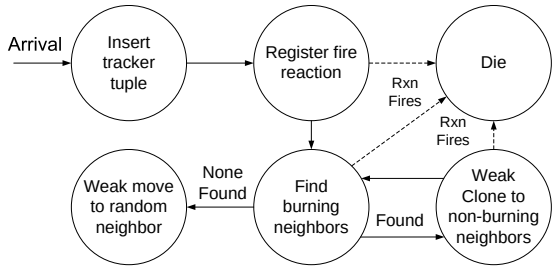

Fig. 18. The life cycle of a fire tracker agent

is inserted into the local tuple space. The code that registers this reaction is shown in Figure 17. Lines 1-2 push 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 off all LEDs (lines 6-7), removes its tracker tuple (lines 8-10), and then dies. The tracker agent inserts a tracker tuple which is used by other tracker agents to determine the integrity of the perimeter. If the fire breaches the perimeter, the tracker agents next to the breach will detect the lack of a neighbor, and re-form the perimeter by cloning themselves onto other neighbors. A persistent breach of the perimeter is considered a failure.

The life cycle of a fire tracker agent is shown in Figure 18. It works by repeatedly checking whether any of its neighbors are on fire. If none are, it performs a weak

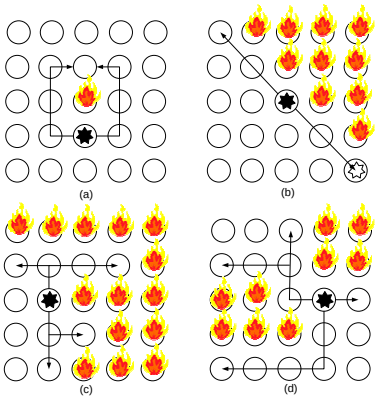

Fig. 19. The static fire tests

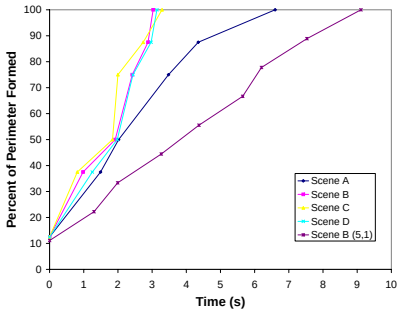

Fig. 20. The results of the static fire tests.

move to a random neighbor and repeats the process. If a neighbor is on fire, it enters a tracking mode where it lights up its green LED and executes 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 does not have 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 near the fire allows the tracker agent to adjust the perimeter as the fire spreads. The fire tracker agent was implemented in 101 bytes of code.

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, the node's (x, y) location can be calculated from 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 than normal scenarios due to an increased likelihood of wireless collisions.

For the static fire tests, the WSN is initialized 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. Four different fires were used, as shown in Figure 19. 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 (5,1) 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.

The results of the tests are shown in Figure 20. Notice that in most cases the perimeter is formed within 3 seconds. Scene A took longer because it 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 20, clearly show how the initial point of fire tracking

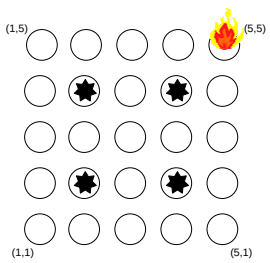

Fig. 21. The dynamic fire tests

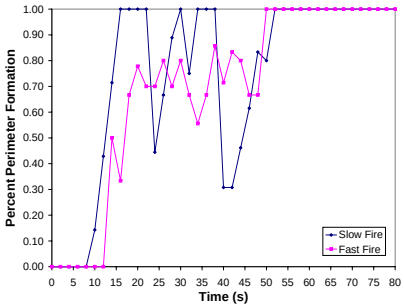

Fig. 22. Results of the dynamic fire tests.

significantly impacts performance.

To evaluate the detector's ability to maintain a perimeter around a spreading fire, we inject four fire tracking agents into the network at the positions marked with a black star in Figure 21, and then inject a dynamic fire agent into node (5,5). We run two tests: one with a slow fire agent that clones every 7 seconds, another with a fast one that clones every 5 seconds. The results of the tests are shown in Figure 22. 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.

The macro-benchmarks show that Agilla can be used to implement sophisticated applications in a dynamic environment. In this case, Agilla agents are used to model, detect, and track fire. Experience developing the mobile agents and experiments evaluating their performance shows that Agilla's model simplifies application development and provides adequate performance for this application.

5. USABILITY CASE STUDIES

In addition to fire detection and tracking, Agilla has been used in several other applications including cargo tracking and robot navigation. These projects were collaborative in nature and are only outlined here. They are presented as case studies that further demonstrate Agilla's usage.

5.1 Monitoring Cargo Containers

In the post-9/11 world, the United States has been keenly aware of terrorists threats and has made large investments in improving our nation's security. One area that remains vulnerable, however, is in the maritime shipping industry and our nation's 361 ports. In 2005, over 10 million cargo containers entered the United States, 95% of which were *not* inspection by the US Customs and Border Patrol (CBP). In the near-term, the number of containers entering this country is forecasted to increase by 11% per year. This presents an obvious and growing security threat since many of the containers originate from regions with lax security and where terrorists are

known to operate. Unfortunately, inspecting every container as it passes through a US port is not possible due to cost.

The primary strategy of the Department of Homeland Security (DHS) is to inspect the containers at the foreign ports before they are loaded onto US-bound ships. However, this requires ensuring that the containers are not tampered with en-route. One way to accomplish this is through the use of container security devices (CSDs).

Existing CSDs are relatively primitive; they consist of a physical cable or seal that must be damaged to gain access through the door. They are easily circumvented and do not detect non-door intrusions, e.g., torching a hole through a side wall. For this reason, there is considerable interest in using an electronic CSD. In November 2004, General Electric began field testing their CommerceGuard™ CSD that can detect unauthorized entry but does not provide real-time event notification. Another CSD, the SkyBitz Mobile Terminal, provides real-time data over a satellite link to a Service Operation Center (SOC). The SOC, however, is a single point of failure and limits scalability. Finally, MachineTalker, Inc., is developing a CSD that forms a wireless ad hoc network among the containers. Their devices communicate over a local gossip protocol where when one device detects a breach, it notifies its neighbors, which in turn notify their neighbors. Eventually, the base station hears about the breach and notifies the authorities. By using an ad hoc network, this system is able to adapt to the dynamic cargo shipping environment where containers are constantly moved.

A flexible wireless ad hoc network infrastructure is only half the solution. In order to be useful, the ad hoc network must have equally flexible software. Many different users are involved in container shipping, e.g., the exporter, importer, carrier, shipper, customer, DHS, and CBP. Each of them will want to execute their own software on the network. The minimal computational resources within a CSD prevents installing everyone's program. In addition, a user's demands will change. For example, the CBP may need to run different programs based on the current security threat level. Also, the CSDs are often installed on a container for years at a time over which the requirements will likely change and new algorithms or policies may be developed that require the devices to be reprogrammed. Agilla provides the necessary flexibility that this application demands. Imagine a ship arriving at a US port and US CBP (software) agents jumping onto the ship bouncing from one container to another looking for anomalies. Other agents are then injected to analyze the electronic manifest lists for suspect cargo. Those containers are flagged for manual inspection, while the remaining containers are expedited through the customs process, increasing efficiency.

Our group collaborated with corporate partners and deployed an 12-node Agilla network on a mock cargo container test bed as shown in Figure 23. Each 40-foot container is represented by a box and is augmented with a Mica2 Mote. The Mica2 mote contains a daughter board for sensing light, temperature, and vibration, and a speaker for emitting an audible alert. After the network is deployed, mobile agents are used to load the manifests, arm the containers, and query the manifests and security events. To load the manifests, the base station creates a mobile agent with the manifest and programs it to migrate to the destination where it inserts

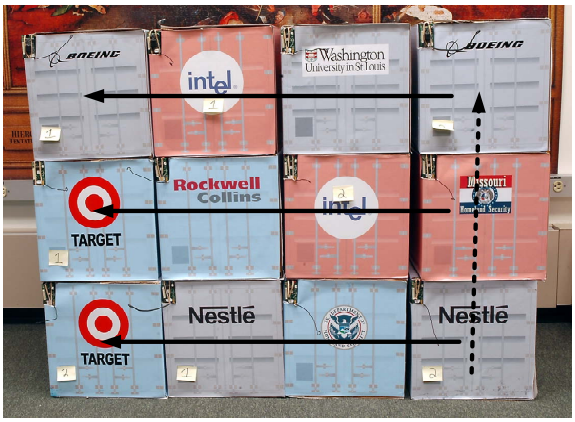

Fig. 23. The cargo container test bed. The arrows show a hybrid approach to visiting each container: dotted is clone, solid is move

a tuple containing the manifest into the local tuple space. For this case study, a light-weight security mechanism is used where each field in the tuple is encrypted at the base station. The underlying communication link is assumed to be secured using lower-layer technologies like TinySec [Karlof et al. 2004].

For the operations that require visiting every container (e.g., search), a balance must be met between network utilization and latency. If a single agent is used to sequentially visit every node, the latency will be high. However, if the agents are programmed to repeatedly clone, the network may be congested, resulting in unreliable results. Through experimentation, we discovered that cloning the agent along one axis, and having the clones move in the same direction along another axis, as shown in Figure 23, resulted in the most reliable results with the lowest latency. Note that since the agents need to remember whether they are moving or cloning, and in which direction they are traveling, strong migration operations are used. Once an agent arrives at a container, it performs some local computations and if it needs to report back to the base station, it does so using the remote out (`rout`) tuple space operation.

A significant advantage that electronic CSDs have over RFIDs is their ability to perform local computations. In this case study, special security agents are injected into the network that "arm" a container by monitoring the sensors and recording anomalies. These agents monitor the accelerometer to detect jarring, and the light sensor to detect unauthorized intrusions (it is assumed that the CSD is located inside the container and that light enters when the container is breached). When excessive acceleration or light occurs, the security agent saves an alert tuple in the local tuple space, sends one to the base station, and flashes the mote's LEDs and emits an audible alert. Since the alert tuple is saved in the local tuple space, it remains even if the security agent dies. Later, a CBP agent may search for these alert tuples, and report back which containers should undergo additional inspection.

The cargo shipping application has been implemented and demonstrated at SenSys 2005 [Hackmann et al. 2005]. It establishes the need for a flexible software

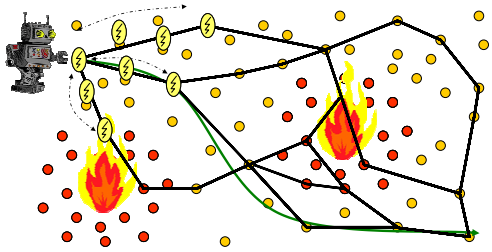

Fig. 24. The robot navigation problem. A roadmap graph is overlayed on the WSN and mobile agents are used to query the temperature along the edges.

infrastructure and illustrates how Agilla may be used to meet this need. The case study uses a variety of mobile agents that serve different users. Their collective functionality could theoretically be aggregated into a single static application, provided there is enough memory, but such a solution may not be updated as easily when new algorithms are developed and unforseen needs may not be as easily accommodated.

5.2 Robot Navigation

Imagine a robot traveling through an area partially engulfed in flames as shown in Figure 24. The robot's goal is to travel across the area while avoiding the fire. The fire is dynamic and unpredictable, preventing the use of a pre-planned route. The robot has on-board sensors that can sense the fire, but have limited range. To help the robot learn more about the fire's location, a WSN running Agilla is deployed that the robot can use. This case study focuses on how the robot can exploit mobile agents and WSNs to help it make better navigation decisions.

A key challenge with developing a robot navigation protocol is efficiency. At each point along a robot's route, the robot can theoretically move in any direction. However, considering every possibility quickly makes the problem intractable [Hwang and Ahuja 1992]. To reduce the problem's complexity, many motion-planning algorithms use a roadmap [Bayazit et al. 2002], which is a pre-computed set of feasible paths that the robot can take. It is shown as an overlay graph in Figure 24. By limiting the problem space to the roadmap, the problem remains tractable; a solution can be generated by running Dijkstra's shortest path algorithm on the roadmap. Unfortunately, certain edges on the roadmap may no longer be safe due to changes in the wildfire's position. Assuming fire can be detected by its temperature, the maximum temperature along a roadmap's edge is indicative of how safe it is to traverse the edge. Since the robot's on-board thermometers have limited range, the robot can make better navigation decisions if it could query the WSN to determine the maximum temperature along an edge in the roadmap.

The robot queries the maximum temperature of an edge using mobile agents. It injects a network exploration agent that repeatedly clones itself onto nodes surrounding the path. These agents form a tree that allows temperature readings to be delivered back to the robot. Since the fire is dynamic, the network exploration agents must not travel too far away. If they do, the temperature readings will have changed by the time the robot arrives. Thus, the robot defines a query area that

bounds the network exploration agents. To limit network overhead, the parent nodes filter data coming from nodes further down the edge. For example, since the robot is only interested in the maximum temperature along a path, each agent can filter all temperature readings less than its current temperature. Also, since the temperature readings of two sensors close to each other are approximately the same, the exploration agents do not clone themselves onto immediate neighbors, but spread themselves out along an edge. This process is repeated for each outgoing roadmap edge. Once the robot receives the maximum temperature along these edges, it can weigh the edges based on their temperature and length, and run Dijkstra's shortest path to find a safe and efficient route.

The robot navigation case study was implemented and evaluated using a test bed consisting of 17 Mica2 motes and an ActiveMedia Pioneer-3 DX robot [Bhattacharya et al. 2005]. The evaluation showed that the use of mobile agents and WSNs increases a robot's success rate by up to 77% relative to traditional protocols that relied only on sensors on-board the robot.

6. RECENT EXTENSIONS

The experience with developing applications using Agilla has led to extensions that improve upon Agilla's basic model. They include an extended instruction set and lightweight asynchronous reactions.

Extended instruction set. While developing applications on top of Agilla, it became clear that Agilla needed to accommodate new instructions for handling complex tasks that would otherwise incur excessive overhead if implemented in Agilla's interpreted byte code. For example, the fire detection agent presented in Section 4.2 used a relatively simple algorithm for detecting fire based only on the temperature reading. Implementing a more sophisticated algorithm that uses multiple sensors or complex signal processing algorithms in Agilla byte code will be inefficient if not impossible. Application-specific instructions may significantly increase an agent's efficiency by changing the virtual-native code boundary [Levis et al. 2005].

Recall that Agilla instructions are implemented as TinyOS tasks. An instruction's byte code is mapped to the task that implements it using TinyOS's parameterized interface mechanism. Specifically, the Agilla Engine uses interface `ByteCodeI` that is parameterized by an 8-bit value. This 8-bit value corresponds to the byte code, and the component that is wired to the interface contains the implementing task. Since the parameter is an 8-bit value, up to 256 instructions can be wired to the Agilla Engine. While this is enough to hold the commonly used instructions, it does not offer enough space for adding application-specific instructions. To address this, the Agilla Engine was modified to provide 12 additional `ByteCodeI` interfaces for adding up to 3072 application-specific instructions. The instructions wired to these interfaces make up Agilla's extended ISA. The Agilla assembler includes a configuration file that specifies the extended ISA. Extended instructions are assembled into two instructions: the first switches the Agilla Engine into one of 12 extended ISA modes, while the second is the actual instruction. After executing the extended instruction, the Agilla Engine switches back into basic mode. While extended instructions are double the size of most basic instructions, they are used

less frequently, and perform application-specific functions more efficiently than a long sequence of interpreted Agilla byte code could.

Lightweight asynchronous reactions. Recall that Agilla's original reaction mechanism evaluated in the macro-benchmarks was not atomic, and that when a reaction was registered, it would only fire once on the existing tuples even if there were multiple matches. The purpose of this was to simplify the middleware's implementation and minimize memory usage. However, it complicated the agent's code since it would have to search for additional matches at the end of each reaction call-back function, and the lack of atomicity made it difficult to achieve reliable behavior. This became especially troublesome in the cargo tracking and robot navigation scenarios where numerous agents were trying to coordinate to cover a region of the network. Furthermore, if enough matching tuples were inserted after the reaction was registered, a stack overflow may occur as the reactions preempt each other. To address these problems, a new reaction mechanism with lightweight asynchronous semantics was introduced.

The new Agilla reactions are atomic but are asynchronous in terms of when a matching tuple is inserted, and when the reaction fires. Agilla's reactions are similar to TinyOS tasks in that they are executed sequentially, i.e., a reaction cannot preempt another reaction. They exhibit "*eventually*" semantics in which a reaction will react to *all* matching tuples so long as they remain in the tuple space. If there are multiple tuples that match a reaction when it is registered, or if matching tuples are inserted afterwards, the reaction may not fire immediately, but rather is only guaranteed to fire if the tuples remain in the tuple space. Thus, it is possible for a tuple to be inserted and removed so fast that an agent never reacts despite having registered a reaction sensitive to it. However, reactions are guaranteed to react to *all* matching tuples that do remain in the tuple space regardless of *when* they were inserted, which simplifies agent code. By providing these semantics, Agilla need not implement the heavyweight transaction mechanisms that are necessary to provide stronger semantics.

The semantics of Agilla's new reaction mechanism were carefully chosen to ensure a lightweight implementation. The implementation required a re-design of the reaction manager to run when the Agilla engine is idle. To ensure that a reaction reacts to all matches exactly once, the reaction manager must remember for each reaction which tuples have been checked. Since the tuple space's memory is allocated linearly, this can be efficiently implemented using a single pointer for each reaction that divides the tuple space into a region containing tuples that have been considered, and those that have not. When a matching tuple is found, the agent's current program counter (PC) and condition code is saved onto the stack and the PC is changed to point to the first instruction of the reaction's function. The agent enters a "reaction" mode in which it cannot execute a blocking operation or be interrupted by another reaction. This ensures that reactions are atomic. Finally, to leave the reaction mode, the agent executes instruction `endrxn`, which restores the agent's PC and condition code.

7. RELATED WORK

Agilla is a general-purpose middleware that simplifies application development while increasing a WSN's flexibility. There are many projects related to Agilla, both in terms of exploiting mobile agent technology, and providing a key requisite for flexibility: in-network reprogramming.

Mobile agents have been used for many years on other networks and their benefits are well established [Lange and Oshima 1999]. Some systems for the Internet include Agent Tcl [Gray 1997], Ara [Peine and Stolpmann 1997], Java Aglets [P.E.Clements et al. 1997], Mole [Baumann et al. 2002], Sumatra [Acharya et al. 1997], TACOMA [Johansen et al. 1995], PeerWare [Cugola and Picco 2001], and MARS [Cabri et al. 2000]. They have been successfully used in data mining [Lange and Oshima 1999], e-commerce [Maes et al. 1999], and network management applications [Baldi and Picco 1998]. Mobile agents have also been used in wireless ad hoc networks. Systems that enable this include Lime [Murphy et al. 2001], EgoSpaces [Julien and Roman 2002], and Limone [Fok et al. 2004]. All of these systems differ in the communication and migration primitives provided, however they adhere to the general mobile agent principle of autonomous execution units capable of migrating across physical nodes.

The mobile agent system closest to Agilla is concurrent work done by [Szumel et al. 2005]. Like Agilla, their system is built atop TinyOS and Maté [Levis and Culler 2002], runs on Mica2 motes, provides a neighbor list, and agents communicate using per-host shared memory spaces. However, there are several key differences. First, unlike Agilla, their system provides both unreliable and reliable migration. This allows the programmer to decide whether the added overhead of reliable migration is worth it. Second, agents always resume from the beginning after migrating, meaning strong migration is not provided. Finally, they do not provide remote communication primitives at the middleware level, but expose the underlying network interface. This is a lower level of abstraction than that provided by tuple spaces. They have evaluated their system in event tracking, localized data collection, and global data collection applications, and showed that it performs linearly with the number of nodes.

Much work has been done regarding how mobile agents can be exploited in a WSN. Most of these studies are theoretical and evaluated using simulations or on resource-rich devices. For example, in [Wu et al. 2004], the authors developed algorithms that calculate efficient routes for mobile agents that perform data fusion and showed through simulations that these algorithms are better than existing heuristics. [Qi et al. 2001] developed algorithms that allow agents to perform target classification efficiently. [Qi et al. 2003] ran simulations to determine when mobile agents perform better than traditional client/server solutions. [Tseng et al. 2004] use mobile agents in a WSN to track mobile entities as they move through a sensor field. Experiments on a WiFi network and simulation results show that their algorithm works, but could be improved. [Tynan et al. 2005] developed a methodology for debugging mobile agent applications for WSNs. Their methodology involves three stages of debugging that attempts to fix as many bugs as possible while running on a central base station before it is deployed on a distributed WSN. Finally, [Wooldridge and Jennings 1995] investigates how strong agents that are capable of

deliberation through, for example, the Belief-Desire-Intention paradigm [Rao and Georgeff 1995], can increase a WSN's energy efficiency [O'Hare et al. 2005].

Many systems enable in-network reprogramming. They can be divided into those that reprogram a node's native code, and those that reprogram virtual code. Deluge [Hui and Culler 2004] and MOAP [Stathopoulos et al. 2003] are two systems that work by re-flashing native code. They are designed to transfer large program binaries and are inappropriate for frequent reprogramming. SOS [Han et al. 2005] provides a micro-kernel that supports dynamically linked modules. This allows networks to be partially reprogrammed. SOS modules, however, are not mobile agents because they cannot control where they are installed, or carry execution state as they are propagated. A middleware that supports reprogramming native code, but does not transfer the entire program image, is Impala [Liu and Martonosi 2003]. Impala applications are divided into seven modules that can be independently updated. However, the implementation was done on relatively powerful PDA-class devices. Other native-code reprogramming systems limit overhead by only sending the changes as determined by the diff [Reijers and Langendoen 2003] and rsync [Jeong 2005] algorithms.

Systems that use virtual code to reprogram a network include Maté [Levis and Culler 2002], SensorWare [Boulis et al. 2003], and Smart Messages [Kang et al. 2004]. In Maté, 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 instructions within them. Maté does not allow a user to control where an application is installed. This limits the network to run one 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 powerful iPAQ 3670 platform. Smart Messages support strong migration, but unlike Agilla, it only supports a single thread of execution per node and is implemented on iPAQs. Like Agilla, it enables local communication through shared memory. However, it does not allow inter-node communication, which is achieved through migration.

8. CONCLUSION

Agilla simplifies WSN application development while increasing the network's flexibility. It does this by offering a mobile agent paradigm and enabling users to inject application-specific agents into the network. Once injected, these agents discover their context and coordinate using neighbor lists and localized tuple spaces, and are capable of autonomous self-directed propagation through clone and move primitives. Since mobile agents can be continuously injected, the network is flexible. Micro-benchmarks performed on a Mica2 test bed show that Agilla operations exhibit performance comparable to those of other VM-based middleware. Macro-benchmarks on a fire detection and tracking application demonstrate that Agilla's computational model enables complex application development. Usability case studies further demonstrate Agilla's usefulness, while also offering insights on how Agilla may be improved. These improvements include adding an extended instruction set for implementing application-specific instructions, and lightweight

asynchronous reactions with stronger atomicity that further simplifies agent implementations. Much work has been done on how mobile agents may be used in WSNs. Agilla serves as a foundation on which to evaluate this work in a real WSN, and rapidly build exciting new, more flexible, applications.

Acknowledgements

This research is supported by the Office of Naval Research under MURI research contract N00014-02-1-0715 and by the the NSF under contract CNS-0520220. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors. We would like to thank Boeing for collaborating on the cargo tracking application, and Sangeeta Bhattacharya, Nuzhet Atay, Gazihan Alankus, and Burchan Bayazit for collaborating on the robot navigation application.

This paper is a substantially revised and extended version of earlier work that appeared in "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications" in the *Proceedings of the 24th International Conference on Distributed Computing Systms*, 653–662, June 2005, and "Mobile Agent Middleware for Sensor Networks: An Application Case Study" in the *4th International Conference on Information Processing in Sensor Networks*, 382–387, April 2005.

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: Heidelberg, Germany, 111–130.

Baldi, M. and Picco, G. P. 1998. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proceedings of the 20th International Conference on Software Engineering*, R. Kemmerer, Ed. IEEE Computer Society Press, 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 *Proc. 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. 2005. Roadmap query for sensor network assisted navigation in dynamic environments. Tech. Rep. WUCSE-05-41, Washington University in St.Louis.

Boulis, A., Han, C.-C., and Srivastava, M. 2003. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys*. USENIX, 187–200.

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

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

Carbunar, B., Valente, M. T., and Vitek, J. 2004. Coordination and mobility in CoreLime. *Mathematical Structures in Computer Science 14,* 3, 397–419.

Crossbow Technology. 2005a. MICA2 wireless measurement system. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

Crossbow Technology. 2005b. MICAz wireless measurement system. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

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

Culler, D., Estrin, D., and Srivastava, M. 2004. Overview of sensor networks. *IEEE Computer 37,* 8, 41–49.

Dunkels, A., Grönvall, B., and Voigt, T. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*.

Fok, C.-L. 2005. The Agilla ISA. http://mobilab.wustl.edu/projects/agilla/isa.html.

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

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

Gray, R. 1997. Agent Tcl. *Dr. Dobb's Journal of Software 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 the 3rd Annual Conference on Embedded Networked Sensor Systems (SenSys'05)*. ACM, 303.

Han, C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. B. 2005. A dynamic operating system for sensor nodes. In *The Third International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 163–176.

Hightower, J. and Borriello, G. 2001. Location systems for ubiquitous computing. *IEEE Computer 34,* 8 (Aug.), 57–66.

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

Hui, J. 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*. ACM Press, 81–94.

Hwang, Y. K. and Ahuja, N. 1992. Gross motion planning – a survey. *ACM Comput. Surv. 24,* 3, 219–291.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of MobiCom 2000*. 56–67.

Jeong, J. 2005. Incremental network programming for wireless sensors. M.S. thesis, EECS 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. June.

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

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. *Special Issue on Mobile and Pervasive Computing, The Computer Journal 47*, 475–494.

Karlof, C., Sastry, N., and Wagner, D. 2004. Tinysec: A link layer security architecture for wireless sensor networks. In *Second ACM Conference on Embedded Networked Sensor Systems (SensSys 2004)*.

Kim, Y.-J. and Govindan, R. 2005. Geographic routing made practical. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. USENIX.

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

Levis, P. and Culler, D. 2002. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, NY, USA, 85–95.

Levis, P., Gay, D., and Culler, D. 2005. Active sensor networks. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*. USENIX.

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

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

Murphy, A. L., Picco, G. P., and Roman, G.-C. 2001. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*. 524–533.

O'Hare, G. M. P., Marsh, D., Ruzzelli, A., and Tynan, R. 2005. Agents for wireless sensor network power management. In *Proceedings of International Workshop on Wireless and Sensor Networks (WSNET-05)*. IEEE Computer Society, Oslo, Norway, 413–418.

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

Peine, H. and Stolpmann, T. 1997. The architecture of the Ara platform for mobile agents. In First International Workshop on Mobile Agents MA'97, R. Popescu-Zeletin and K. Rothermel, Eds. *Lecture Notes in Computer Science 1219*, 50–61.

Polastre, J., Szewczyk, R., and Culler, D. 2005. Telos: Enabling ultra-lower power wireless research. In *The Fourth International Symposium on Information Processing in Sensor Networks SPOTS Track (IPSN'05)*. ACM and IEEE, ACM Press, Los Angeles, CA, 364–369.

Priyantha, N., Chakraborty, A., and Balakrishnan, H. 2000. The cricket location-support system. In *Mobile Computing and Networking*. 32–43.

Qi, H., Iyengar, S. S., and Chakrabarty, K. 2001. Multiresolution data integration using mobile agents in distributed sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics – Part C 31,* 3 (Aug.), 383–391.

Qi, H., Kuruganti, P. T., and Xu, Y. 2002. The development of localized algorithms in wireless sensor networks. *Sensors 2,* 7, 286–293.

Qi, H., Xu, Y., and Wang, X. 2003. Mobile-agent-based collaborative signal and information processing in sensor networks. In *Proceedings of the IEEE*. Vol. 91. IEEE, 1172–1183.

Rao, A. S. and Georgeff, M. P. 1995. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*. San Francisco.

Reijers, N. and Langendoen, K. 2003. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. ACM Press, San Diego, CA, USA, 60–67.

Stallings, W. 2001. *Operating Systems Fourth Edition*, 4 ed. Pretence Hall, New Jersey.

Stathopoulos, T., Heidemann, J., and Estrin, D. 2003. A remote code update mechanism for wireless sensor networks. Tech. Rep. CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing.

Szumel, L., LeBrun, J., and Owens, J. D. 2005. Towards a mobile agent framework for sensor networks. In *Second IEEE Workshop on Embedded Networked Sensors*. IEEE, Sydney, Australia, 79–87.

Tseng, Y.-C., Kuo, S.-P., Lee, W.-W., and Huang, C.-F. 2004. Location tracking in a wireless sensor network by mobile agents and its data fusion strategies. *The Computer Journal 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 proceeding of SEKE'05, the 17th International Conference on Software Engineering and Knowledge Engineering,Taiwan, IJSEKE press.*.

Tyndall National Institute. 2005. The 25mm cube module. http://www.tyndall.ie/research/mai-group/25cube_mai.html.

Wooldridge, M. and Jennings, N. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review 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 Transactions on Knowledge and Data Engineering 6,* 16 (June), 740–753.

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