Middleware Mechanisms for Agent Mobility in Wireless Sensor and Actuator Networks

Nikos Tziritas[1,2,3], Giorgis Georgakoudis[1,2], Spyros Lalis[1,2], Tomasz Paczesny[4], Jarosław Domaszewicz[4], Petros Lampsas[1,5], and Thanasis Loukopoulos[1,5]

[1] Center for Research and Technology Thessaly,
Technology Park of Thessaly 1st Industrial Area, 385 00 Volos, Greece
[2] Department of Computer and Communication Engineering,
University of Thessaly,
Glavani 37, 38221 Volos, Greece
`{nitzirit,ggeorgak,lalis}@inf.uth.gr`
[3] Shenzhen Institutes of Advanced Technology,
Chinese Academy of Sciences, Shenzhen, 518067, China
`nikolaos@siat.ac.cn`
[4] Institute of Telecommunications,
Warsaw University of Technology,
Nowowiejska 15/19, 00-665, Warsaw, Poland
`{t.paczesny,domaszew}@tele.pw.edu.pl`
[5] Department of Informatics and Computer Technology,
Technological Educational Institute of Lamia,
3rd km. Old Ntl. Road Athens, 35100 Lamia, Greece
`{plam,luke}@teilam.gr`

Abstract. This paper describes middleware-level support for agent mobility, targeted at hierarchically structured wireless sensor and actuator network applications. Agent mobility enables a dynamic deployment and adaptation of the application on top of the wireless network at runtime, while allowing the middleware to optimize the placement of agents, e.g., to reduce wireless network traffic, transparently to the application programmer. The paper presents the design of the mechanisms and protocols employed to instantiate agents on nodes and to move agents between nodes. It also gives an evaluation of a middleware prototype running on Imote2 nodes that communicate over ZigBee. The results show that our implementation is reasonably efficient and fast enough to support the envisioned functionality on top of a commodity multi-hop wireless technology. Our work is to a large extent platform-neutral, thus it can inform the design of other systems that adopt a hierarchical structuring of mobile components.

Keywords: wireless sensor networks, middleware, mobile agents, embedded systems, performance evaluation, Imote2, ZigBee.

1 Introduction

In the POBICOS project [11] we have produced a platform aimed to simplify the development and deployment of monitoring and control applications for the home and office

environment, which exploit regular objects with embedded sensing, actuating and wireless communication capabilities. Objects do not have any application-specific code pre-installed and are agnostic about the applications that might run on them. Each application is injected into the network (referred to as object community) using a special device (the application pill), which stores the code of the application and serves as its controller [6]. To start the application, the user simply pushes a button on the pill, letting the middleware deploy and execute the application on the object community.

POBICOS applications are programmed as a set of cooperating components, called agents. Agents are mobile in the sense that they can be instantiated on remote objects and can be migrated between objects, at runtime. Agent mobility is central to achieving non-trivial functionality. Firstly, it enables a flexible deployment of the application code in the object community, by placing individual agents directly on the objects that provide the required (computing, sensing, actuating) resources. Secondly, it allows the programmer to dynamically control the type of agents that execute in the object community, depending on the application's internal state. This in turn can reduce the amount of code that needs to be kept in the main memory of embedded nodes at any point in time, especially in the presence of several concurrently running applications. Thirdly, the middleware can migrate agents between objects in order to perform certain optimizations, e.g., to reduce the traffic over the wireless network. Unlike in many other embedded agent systems, agent mobility is transparent for the programmer who does not have to discover (suitable) objects or to deal with the placement of agents on objects in an explicit fashion.

This paper describes the protocols and mechanisms that were developed to support agent creation and agent migration in POBICOS. It also provides an experimental evaluation based on a prototype implementation of the middleware on Imote2 nodes that communicate over ZigBee. The results provide valuable insight into the overhead and performance of the agent mobility operations on top of a popular multi-hop wireless technology, showing that they are reasonably efficient and fast enough to support the envisioned functionality. Notably, this work is to a large extent orthogonal to the POBICOS platform: as explained in the next section, the basic underlying assumption is that agents are arranged as a tree according to their parent-child relationship. Hence the presented middleware support and performance trends can inform the design of other systems which adopt a hierarchical structuring of mobile components.

The rest of the paper is structured as follows. Sec. 2 provides an overview of the application model. Sec. 3 describes the implementation of the agent mobility protocols and mechanisms in our middleware. Sec. 4 analyzes their performance, while Sec. 5 puts the costs and benefits of agent mobility in context of an indicative application scenario. Sec. 6 discusses related work. Finally, Sec. 7 concludes the paper.

2   Application Model

The POBICOS application model evolved from that of the ROVERS system [3]. An application is designed as a collection of cooperating agents, with each agent being dedicated to a specific, perhaps very simple, task. In the spirit of hierarchical control

systems [9], agents are organized in a tree. Leaf agents interact with the physical environment by acquiring information or effecting change through the sensors and actuators embedded into the objects of the community. The rest of the agents in the application tree implement higher-level aggregation, processing and control tasks, using only general-purpose computing resources (CPU and memory). Agents communicate via message passing. Being consistent with the hierarchical approach, an agent can exchange messages only with its parent and children.

Fig. 1a shows the structure of a simple application that turns off lights when there is no user activity (of course, applications can be more complex). The root agent (R) employs a user presence inference agent (I), which relies on multiple user activity agents (A) to detect user activity (and to infer inactivity). The root also employs multiple light agents (L) to switch off lights when the user presence inference agent (I) reports user absence, and a notification agent (N) to inform the user about this action.

(a)   (b)

Fig. 1. (a) Tree structure of a simple light control application; (b) A concrete deployment of the light control application on top of an indicative object community

The agent tree is formed, at runtime, in a top-down fashion. The root is automatically created by the middleware on the application pill when the application is started. All other agents are created under the control of the application according to the desired tree structure. The placement of agents in the community is performed by the middleware, with no direct involvement of the programmer, based on the objects that are available. In the deployment shown in Fig. 1b, user activity agents are created on the motion detector, refrigerator and water tap, because these objects can serve as user activity sensors. Light agents are created on all light sources, while the user notification agent is created on the radio, which can issue messages to the user. The middleware can move non-leaf agents between objects in a transparent way. For instance, the user presence inference agent could be migrated from the application pill on the motion detector, to communicate locally with the respective user activity agent. Migration is supported only for non-leaf agents because they are object- and location-neutral by design; leaf agents remain on the nodes where they are created.

It is worth noting that parts of the agent tree can be instantiated and destroyed spontaneously, long after the application has been deployed. As an example, the light control application could create light agents only when the decision is taken to turn

the lights off, and then remove them once they perform their task (as opposed to creating them "statically" at startup). Of course, it is up to the programmer to decide if such dynamic changes in the application tree are meaningful.

3   Implementation of Agent Mobility

The middleware components involved in the implementation of agent mobility are shown in Fig. 2. The core functionality is provided by the Agent Manager, which invokes the Agent Runtime to check resource availability, as well as to initialize, run, suspend and resume agents. Agent binaries are downloaded via the Code Transport, employing a stop-and-wait protocol and a cache to avoid fetching the same code repeatedly over the network. The Network Abstraction offers a generic datagram interface, used by both the Agent Manager and Code Transport.
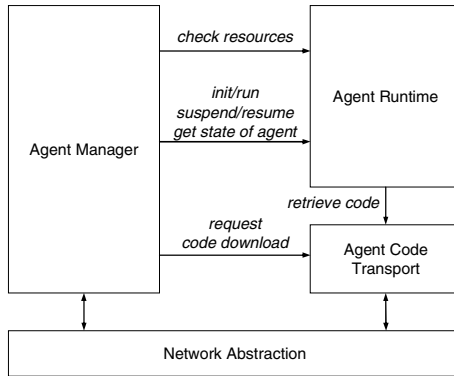
Fig. 2. Key middleware components and interactions for supporting agent mobility

Our prototype is developed for TinyOS v2.1 running on Imote2 nodes [8] at 104MHz. Wireless communication is done via an external ZigBee modem from the Z430-RF2480 demo kit of Texas Instruments [14]. The Network Abstraction component breaks datagrams into ZigBee packets and implements its own software-based acknowledgement and retransmission scheme (relying on ZigBee for packet routing). The middleware is portable, assuming support for TinyOS; obviously, the Network Abstraction must be adapted to the underlying networking technology. Thanks to a system component that provides transparent access to external memories (e.g., Flash), the minimal RAM requirements are below 8KB, making it possible to target more resource-constrained devices (even though access to certain middleware data structures would be slower, we believe that the middleware would still work acceptably).

3.1   Micro-agent Code, Execution and State

Nodes provide a platform-neutral runtime on top of which agents execute. The VM is based on the 8-bit AVR architecture [1]. Agents are written in C and the respective

binaries are generated via the standard AVR-GCC tool-chain. The binaries are then processed using a special tool to bind into the POBICOS-specific primitives and reduce their size [12].

Agent execution is purely event-driven (agents do not have threads of their own). The Runtime puts events issued to agents in a FIFO queue, and executes the respective handlers in a non-preemptive fashion. Agents are migrated only between handler executions, when the stack is empty and the VM CPU is not being used, which greatly simplifies the respective suspend-resume process. Also, agents do not use dynamic memory, so the runtime state that needs to be transferred over the network when a migration takes place is just the agent's static data.

The Agent Manager maintains additional data for each local agent, namely entries for its children and any pending creation requests. In case of a migration, this information must also be sent to the destination along with the agent's state.

3.2   Creation of Leaf and Non-leaf Agents

Agent creation requests issued by the application are processed in an asynchronous fashion. The process for creating a leaf agent is as follows (see Fig. 3a).

To find nodes that can serve as hosts, a probe is broadcast (1) carrying information about the agent's type and resource requirements. When a probe arrives to a node, a resource check is performed to see whether it is able host such an agent (2). A reply is generated (3) only if this check succeeds. Replies are collected (4) and sorted based on how well nodes match the agent's requirements (the details of this matching are beyond the scope of this paper). Candidates are then approached one at a time.

A node is asked to create an agent by sending it a creation request (5). On receipt of such a request, to avoid races, the node repeats the same checks as for a probe (6). Then, it downloads the agent code from the application pill (7), creates a new instance (8), recording the sender of the request as the agent's parent, and sends back a reply with the agent's identifier (9). If the reply is positive, a new child entry is added to the parent (10). Else, if the reply is negative or no reply arrives within a timeout period, the next candidate is considered.

The creation of non-leaf agents works in a similar way. However, host discovery is performed only if the local host cannot host the agent, and candidates are contacted in the arrival order of their replies. The rationale is that since non-leaf agents are object-neutral it is reasonable to place them, at least initially, close to their parent.

3.3   Migration of Non-leaf Agents

The algorithm for deciding about the migration of a non-leaf agent in order to reduce the wireless traffic is described in [15]. The idea is to locally record the agent's messaging activity with its parent and children, and to move the agent towards the center of gravity, i.e., over the link that accounts for more than half of the total load. We have implemented the k-hop variant of the algorithm, which assumes knowledge about the routing structure within a k-hop radius and can pick migration destinations in this range. In ZigBee tree networks, where node addresses reflect the routing topology [10], this information can be gained without any extra communication.
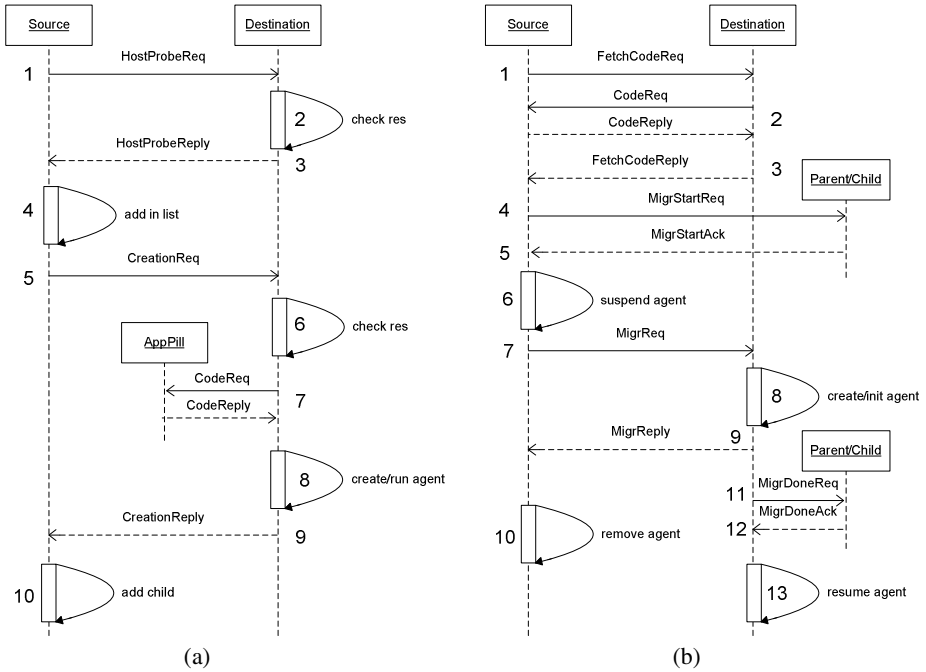
Fig. 3. Simplified (a) agent creation and (b) agent migration message sequence diagrams

Once a migration decision is taken, the process is as follows (see Fig. 3b). First, the destination node is asked to download the code (1-3). The code is fetched from the node that initiates the migration (which, obviously, has the binary). Then, the hosts of the agent's parent and children are notified (4) to buffer messages addressed to it, but also to prohibit concurrent migrations (parents have precedence over their children). When all acknowledgements arrive (5), the agent is suspended (6) and a migration request with the agent's state information is sent to the destination (7). The destination creates a new instance, initializing it with the received state (8), and sends a confirmation to the agent's old host (9), which removes the obsolete instance (10). It then informs the agent's parent and children (11) to update the agent's contact address and resume message transmission towards it. Finally, when the parent confirms the address change (12), the agent is resumed on the new host (13).

Note that the agent binary is "pre-fetched", before contacting the agent's parent and children or suspending the agent. Consequently, the latency of code transfer does not affect the execution of the application. As it will be shown in the next section, this greatly reduces the period during which the application may become unresponsive due to agent migrations performed by the middleware in the background.

4   Performance Measurements

This section presents measurements on the performance of agent creation and migration. The cost of the respective protocols is reported as the number of bytes

exchanged through the Network Abstraction component of the middleware, as well as through the (lower-level) ZigBee modem interface; the difference is mainly due to datagram fragmentation. Unless stated otherwise, the network topology is a 4-node chain, with the ZigBee coordinator at the one end acting as the source and other nodes as the destinations of the mobility operations (we report results for up to 3 hops because the network was very unreliable for longer routing paths).

4.1   Agent Creation Overhead

In a first set of experiments, we measure the overhead for creating a leaf agent that requires a special resource, e.g., a user activity sensor. The results for non-leaf agents are similar. The local creation delay for such an agent is 1ms.

Table 1. Cost of agent creation protocol, at the network abstraction layer and ZigBee

agent size [B]	code transport cost [B]		signaling cost [B]		total protocol cost [B]	
	Net Abstr.	ZigBee	Net Abstr.	ZigBee	Net Abstr.	ZigBee
300	352	484	71	107	423	591
600	684	912	71	107	755	1019
900	1032	1392	71	107	1033	1499
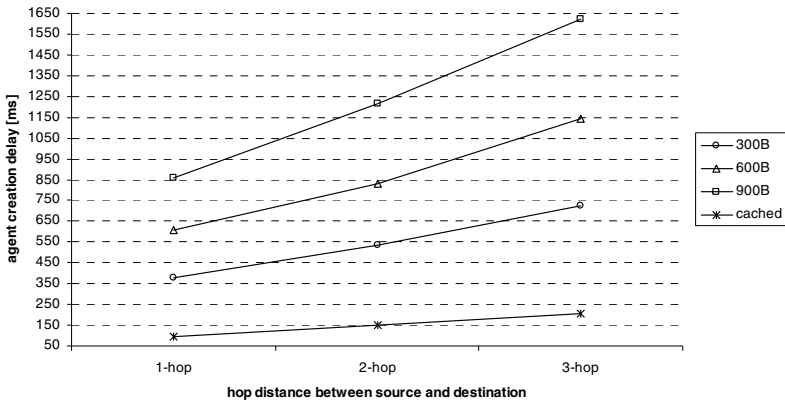
Fig. 4. Agent creation delay as a function of hop distance for different agent sizes

Table 1 analyzes the protocol cost for different agent sizes. The signaling overhead is constant and relatively low, corresponding to one host probe and one agent creation request-reply interaction. Clearly, the dominating component is the code transfer cost, which grows as expected to the agent size. The relative protocol overhead drops as code size increases, but the conversion of datagrams to ZigBee packets costs 35-40%.

Fig. 4 plots the creation time as a function of the hop distance between the source and the destination. The delay rises to the agent size, yet with an economy of scale: creating a 600B agent requires 80% of the time needed to create two 300B agents, and

a 900B agent is created in 75% of the time it takes to create three 300B agents. If the agent binary is already cached at the destination, only the signaling cost is incurred, as per Table 1. Hence the respective delay, shown in Fig. 4, is much smaller vs. when code needs to be transferred over the network, yielding an average speedup of 3.7x, 5.8x and 8.4x for a 300B, 600B and 900B agent.

In all cases, the routing overhead is non-negligible. Nevertheless, creating an agent on a remote node directly (as in our middleware) seems a better choice than letting an agent clone itself in a hop-by-hop fashion (as done in other systems). Based on our results, direct creation over 2 and 3 hops is roughly 1.4x and 1.6x faster vs. cloning the agent along these paths.

4.2 Agent Migration Overhead

In a second set of experiments, we measure the migration overhead for a non-leaf agent that is co-located with its parent and has one child on a remote node to which it migrates directly. The runtime state is set to 256B. The delay for performing a corresponding agent suspend-create-init-resume cycle locally is about 2ms.

Table 2. Cost of agent migration protocol, at the network abstraction layer and ZigBee

agent size + state [B]	code transport cost [B]		signaling cost [B]		total protocol cost [B]	
	Net Abstr.	ZigBee	Net Abstr.	ZigBee	Net Abstr.	ZigBee
300+256	352	484	387	543	739	1027
600+256	684	912	387	543	1071	1455
900+256	1032	1392	387	543	1419	1935
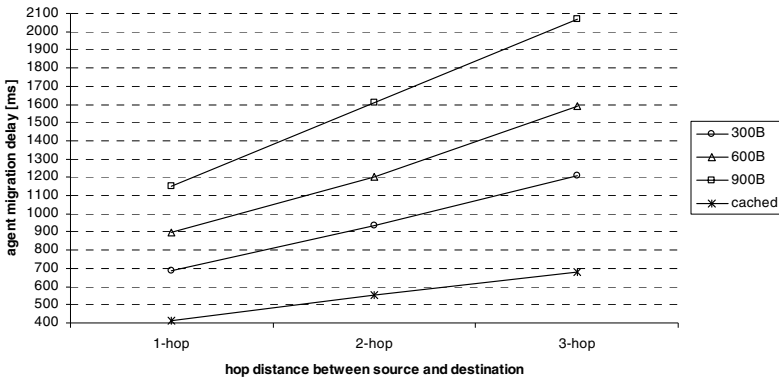
Fig. 5. Agent migration delay as a function of hop distance for different agent sizes

The breakdown of the protocol cost is listed in Table 2. The numbers reported for the code transfer are naturally the same as for agent creation. The signaling cost is much higher though, because it includes the synchronization with the agent's parent and child, as well as the transfer of the agent's state. As a result, the code transfer overhead is less dominant compared to agent creation.

Fig. 5 plots the agent migration time as a function of the hop distance. The trends are similar to the ones observed for agent creation with the respective delays being longer due to the higher signaling overhead. Again, the delay rises with code size, but at a greater economy of scale compared to agent creation, due to the expensive signaling. Namely, the migration of a 600B and 900B agent takes 65% and 57% of the time required to perform two and three migrations of a 300B agent, respectively. For the same reason, the speedup achieved by caching vs. when code transfer occurs is less impressive: 1.7x, 2.2x and 2.9x for a 300B, 600B and 900B agent.

Notably, a direct migration over 2 hops is roughly 1.4x faster vs. two 1-hop migrations, and a direct migration over 3 hops is 1.7x faster than three 1-hop migrations; or 1.5x and 1.8x faster, respectively, when the binary is cached at the destination. This speaks in favor of performing a single long-distance migration vs. several 1-hop ones, as supported by our implementation (the range is set at compile time).

The synchronization with the agent's children also affects the migration delay. To get a feeling of this overhead, we measured the time required to migrate a 600B agent with 256B runtime state while varying the number of its children. In this case, a 5-node star topology is used, with the center node hosting the agent and all children being hosted on different nodes. The recorded delay is 843ms, 874ms, 945ms and 974ms for 1, 2, 3 and 4 children, rising due to the extra signaling required for each additional child. The slight non-linearity from 2 to 3 children is due to the increase in the child information that needs to be transferred along with the agent's runtime state, which happens to exceed the datagram payload limit, requiring an additional datagram to be sent over the network.

4.3   Summary

Our results show that agent creation is fast enough to support not only the gradual formation of the agent tree when the application is deployed but also a quick adaptation of the tree structure at runtime. Furthermore, since creation is practically instantaneous when the binary is cached at the destination, the repeated instantiation (and removal) of agents is a perfectly affordable option for the programmer.

Agent migration is reasonably quick too. Most importantly, since the agent remains fully operational while its code is being fetched by the destination node, the application is blocked only during the signaling and state transfer phase. The latter requires well under 1 second in our experiments (see the values reported for caching), which is quite acceptable given that the home automation applications we wish to support using our middleware have rather soft real-time requirements.

Finally, the 1-hop throughput achieved by the agent mobility operations (calculated as the number of bytes exchanged through the network abstraction layer in order to perform agent creation/migration divided by the time required to complete this operation), is 12-14Kbps. This is close to the 15Kbps throughput of the Network Abstraction component for reliable datagrams.

5   Application Scenario

In this section we put the benefit and cost of agent mobility in the context of a concrete application scenario. The application structure and logic is kept simple in order to easily follow its operation (the network setup was constrained by the number of nodes at our disposal, as well as the difficulties we encountered in setting up a working network with more than 3 hops). Still, we believe that the results are indicative of the potential gains for more complex applications and larger scale settings.

5.1   Application, Network Topology and Test Scenario

The test application is a subset of the light control application discussed in Sec. 2, namely the part used to infer user absence based on the user activity sensors found in a home. Fig. 6a shows the corresponding tree structure.
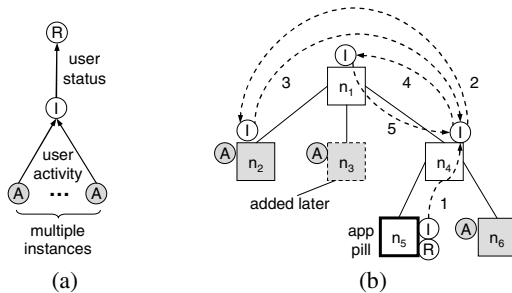
Fig. 6. Experiment setup: (a) application tree; (b) nodes, routing topology, and agent placement at different phases of the test scenario.

When a sensing agent does not detect activity, it sends to the inference agent a 1-byte report every 5 seconds. As long as user activity is being detected, the reporting frequency rises to 1 report per 2 seconds. Based on the reports received, the inference agent sends a 1-byte status report to the root every 10 seconds. The size of the root, inference, and user activity sensing agent is 50B, 240B and 24B, respectively.

Fig. 6b shows the network used to deploy and run the application. Unlike n_1, n_4 and n_5, nodes n_2, n_3 and n_6 represent objects which can act as user activity sensors, and therefore can host a user activity sensing agent. The application is launched from n_5 where the root remains fixed. The inference agent can be placed on any node.

The test scenario is as follows:

0. The application is deployed in the network of Fig. 6b without n_3 (added later). The root and the inference agent are created on n_5, while user activity sensing agents are created on n_2 and n_6.
1. Since the message traffic with its children is greater than the message traffic with its parent, the inference agent is migrated on n_4.

2. The agent on n_2 detects user activity and starts reporting at a higher frequency. This increase in message traffic drives the middleware to move the inference agent on n_2.
3. User activity stops, and the sensing agent on n_2 reverts to the normal reporting frequency. Consequently, the inference agent is migrated back on n_4.
4. Node n_3 (which can act as a user activity sensor) is added to the network, leading to the creation of a sensing agent on it. Due to the reporting activity of its new child, the traffic for the inference agent via node n_1 becomes greater than the traffic with n_5 and n_6, so the inference agent is migrated on n_1.
5. Finally, n_3 is removed, the respective user activity sensing agent disappears, and the inference agent is moved back on n_4.

The dashed lines in Fig. 6b denote the migrations that lead to the different placements of the inference agent for each stage.

5.2   Results

Table 3 lists the protocol cost for each migration of the inference agent, as well as the reduction achieved in the wireless network traffic by the resulting placement (after the migration) vs. the old placement (before the migration). These numbers are reported for the ZigBee modem interface, adjusted to take into account the routing cost for each packet as per the topology in Fig. 6b (ZigBee performs routing transparently). The amortization time for each migration, i.e., the time that must elapse in order for the traffic reduction achieved by the new placement to outweigh the cost of the migration that lead to this placement, is also given in Table 3.

Table 3. Cost of migration, wireless traffic reduction achieved by the resulting placement, and the time (of stable operation) required to amortize each migration of the inference agent.

scenario stages	migration of inference agent	migration cost [B]	traffic reduction [B/min]	relative traffic reduction	amortization time [min]
1	$n_5 \rightarrow n_4$	873	559	30%	1.5
2	$n_4 \rightarrow n_2$	1495	522	22%	2.7
3	$n_2 \rightarrow n_4$	769	486	27%	1.6
4	$n_4 \rightarrow n_1$	1007	174	8%	5.8
5	$n_1 \rightarrow n_4$	511	270	17%	1.9

It can be seen that the migration of the inference agent leads to considerable savings in network traffic, also at a cost that can be recovered in a rather short amount of time. More specifically, the first, the third and the last migration can be amortized in less than 2 minutes, while the second and the fourth migration requires slightly less than 3 and 6 minutes, respectively. Note that when the inference agent returns to a node where it was previously hosted (third and fifth migration), caching reduces the migration cost to 50%, shortening the respective amortization times.

In terms of responsiveness (not shown in Table 3), the delay for creating a user activity agent is about 200ms on average (e.g., the application is deployed in less than half a second). Since user activity agents are created just once on the respective nodes, caching does not apply to this scenario. The average migration delay for the inference agent is 620ms vs. 390ms when the code is cached at the destination. In any case, migration delays are far too insignificant to affect the amortization times or the functionality of the application.

Of course, a migration may turn out to be non-beneficial if the agent tree or the communication pattern between agents changes very fast. In our implementation we use two criteria for identifying and suppressing migrations that are unlikely to be beneficial. Namely, a migration is not performed unless (i) it reduces the amount of network traffic above a threshold and (ii) it can be amortized within a certain amount of time, assuming stable operation. These checks can be done based on information that is locally available. The gains in network traffic that will be achieved after a migration takes place are computed based on the agent's message traffic (the same information is used by the algorithm to decide for a migration), while the migration cost can be estimated using an analytical formula. Both checks are disabled in the experiment. Depending on the thresholds, they would simply lead to fewer migrations.

6   Related Work

Code mobility is supported in many platforms targeted at wireless sensor networks. In the following, we briefly discuss work that is most closely related to ours and give an indicative performance comparison.

Agilla [4] follows a mobile agent approach like POBICOS. However, the application code is written in low-level VM instructions, and the programmer must provide the agent's host discovery and migration logic. Agilla agents communicate indirectly by adding, reading and removing tuples on nodes. Smart Messages [5] (SMs) are mobile code units written in Java, executed using an adapted version of Sun's Java KVM. SMs resemble Agilla agents in that they communicate via the local tag spaces of nodes, and carry their own host discovery and migration code. Also, in both Agilla and SMs, to create an agent/SM instance on a remote node, it must be created locally and then be cloned to the desired destination, typically, in a hop-by-hop fashion. SensorWare [2] allows TCL-based scripts to be injected in a network. Like in Agilla and SMs, the programmer is responsible for providing the logic for cloning/migrating a script, but scripts communicate via message passing. The addressing scheme of SensorWare is very flexible, allowing for attribute-based descriptions combined with the invocation of (default or custom) routing protocols.

MagnetOS [7] statically partitions Java applications, and then places them on different nodes at runtime. Communication transparency is achieved via RPCs. Unless the programmer specifies a placement, the MagnetOS runtime is free to move components between nodes to reduce the network traffic. In DFuse [13], applications are built using so-called fusion points or channels, structured in a hierarchy. Each fusion point takes input from one or more producers and generates output towards one or

more consumers. The initial placement of fusion points, computed off-line, is evaluated at regular intervals to minimize communication and energy consumption, relocating fusion points accordingly.

Agilla is very lightweight, running on MICA2 nodes. All other systems are prototyped on PDA devices, while the reported experiments in MagnetOS were done using laptops. POBICOS seems to be in the middle ground. In fact, given its modest RAM needs and the fact that it is based on TinyOS, the POBICOS middleware could be ported on more constrained platforms than the Imote2. The POBICOS VM can also be implemented efficiently on AVR-compatible microcontrollers, which are a popular choice for low-end devices.

The differences in the programming abstractions, platform CPUs (Atmel 8-bit microcontroller in Agilla, XScale or StrongARM in other systems, except MagnetOS) but most notably the wireless technologies used (WLAN in all systems but Agilla, ZigBee in POBICOS), make a direct performance comparison hard and possibly unfair. Still, to give an idea of where our prototype stands, we pick a few cases where a comparison does not seem entirely out of order. In terms of local operations, creating a POBICOS agent takes about 1ms vs. 2ms for spawning a SensorWare script, or 2.6ms for the creation of a Smart Message using a single 1KB Java class. The suspend-create-init-resume cycle for a POBICOS agent with 256B of state takes 2ms, which is the time needed for serializing and de-serializing a Smart Message with a 53B stack frame and 2KB of state. In terms of remote 1-hop operations that do not involve (significant) code transfer, the creation of a cached POBICOS agent requires 95ms vs. 200ms for weakly cloning a null Agilla agent, 35ms for creating an empty DFuse channel (over WLAN) and 10ms for spawning a 60B script in SensorWare (over WLAN). The migration of a cached POBICOS agent with 256B of state that is co-located with its parent and has one child on a remote node requires 410ms vs. 225ms for a null Agilla agent (in which case no communication endpoints need to be redirected), 200ms for the relocation of an empty DFuse channel with one producer and consumer (over WLAN) and 12ms for the migration of a cached Smart Message with 200B bytes of state (no redirection of communication endpoints, over WLAN).

Overall, given the non-triviality of the underlying protocols and the moderate throughput of our communication subsystem (e.g., compared to WLAN), the performance of our agent mobility operations is quite satisfactory.

7   Conclusion

We have described the implementation of agent mobility in hierarchically structured applications, and have presented a performance evaluation of our middleware prototype on Imote2 nodes that communicate over ZigBee. The results show that agent creation is fast enough to deploy and adapt the application tree structure at runtime. Furthermore, agent migration can reduce the wireless network traffic significantly, even for relatively short periods of heavy inter-agent communication. While a faster communication subsystem would reduce agent mobility delays, the current

performance is already sufficient for a wide range of monitoring and control applications in the home domain.

As future work we plan to implement our own routing on top the native Imote2 radio to experiment with various cross-layer optimizations. We also wish to investigate techniques that will allow the middleware to take smarter agent placement and migration decisions.

References

1. Atmel Corporation: 8-bit AVR Instruction Set, rev. 0856H–AVR–07/09 (2009)
2. Boulis, A., Han, C.-C., Shea, R., Srivastava, M.B.: SensorWare: Programming Sensor Networks beyond Code Update and Querying. Pervasive and Mobile Computing Journal 3(4), 386–412 (2007)
3. Domaszewicz, J., Roj, M., Pruszkowski, A., Golanski, M., Kacperski, K.: ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks. In: Intl. Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 615–620 (2006)
4. Fok, C.L., Roman, G.C., Lu, C.: Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In: 25th Intl. Conference on Distributed Computing Systems (ICDCS), pp. 653–662 (2005)
5. Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., Iftode, L.: Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems. The Computer Journal 47(4), 475–494 (2004)
6. Lalis, S., Domaszewicz, J., Pruszkowski, A., Paczesny, T., Ala-Louko, M., Taumberger, M., Georgakoudis, G., Lekkas, K.: Tangible Applications for Regular Objects: An End-User Model for Pervasive Computing at Home. In: 4th Intl. Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM), pp. 385–390 (2010)
7. Liu, H., Roeder, T., Walsh, K., Barr, R., Sirer, E.G.: Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In: 3rd Intl. Conference on Mobile Systems, Applications and Services (MOBISYS), pp. 149–162 (2005)
8. Memsic, Imote2 node datasheet,
   http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134%3Aimote2
9. Mesarovic, M.D.: Multilevel Systems and Concepts in Process Control. Proceedings of the IEEE 58(1), 111–125 (1970)
10. Pan, M.-S., Fang, H.-W., Liu, Y.-C., Tseng, Y.-C.: Address Assignment and Routing Schemes for Zigbee-based Long-thin Wireless Sensor Networks. In: 67th IEEE Intl. Conference on Vehicular Technology (VTC), pp. 173–177 (2008)
11. POBICOS website, http://www.ict-pobicos.eu

12. Pruszkowski, A., Paczesny, T., Domaszewicz, J.: From C to VM-targeted Executables: Techniques for Heterogeneous Sensor/Actuator Networks. In: 8th IEEE Workshop on Intelligent Solutions in Embedded Systems (WISES), pp. 61–66 (2010)
13. Ramachandran, U., Kumar, R., Wolenetz, M., Cooper, B., Agarwalla, B., Shin, J., Hutto, P., Paul, A.: Dynamic Data Fusion for Future Sensor Networks. ACM Transactions on Sensor Networks 2(3), 404–443 (2006)
14. Texas Instruments: Z-Accell Demonstration Kit,
    `http://focus.ti.com/docs/toolsw/folders/print/`
    `ez430-rf2480.html`
15. Tziritas, N., Loukopoulos, T., Lalis, S., Lampsas, P.: On Deploying Tree Structured Agent Applications in Networked Embedded Systems. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 490–502. Springer, Heidelberg (2010)