Mobile Agents for Integration of Internet of Things and Wireless Sensor Networks

Teemu Leppänen, Meirong Liu, Erkki Harjula, Archana Ramalingam, Jani Ylioja, Pauli Närhi, Jukka Riekki, Timo Ojala
Department of Computer Science and Engineering,
University of Oulu, Finland
{teemu.leppanen, meirong.liu, erkki.harjula, archana.ramalingam, jani.ylioja, pauli.narhi, jukka.riekki, timo.ojala}@ee.oulu.fi

***Abstract*—We demonstrate interoperable mobile agents to integrate Internet of Things and wireless sensor networks with resource-constrained low-power embedded networked devices. We introduce adaptable composition for the mobile agent, complying with the Representational State Transfer principles, which are then used for agent migration, controlling the agent and exposing the data, system resources, tasks and services, to the Web. We gather requirements for the system and heterogeneous networked devices and present an application programming interface to enable mobile agents in these systems. The agents are demonstrated in a real-world prototype with smartphones and embedded networked devices, where we utilize a proxy component to expose system resources to the Web for human-machine interactions. Spanning over disparate networks and protocols, the proxy translates messages including the agent composition, between HTTP and Constrained Application Protocol. Lastly, we suggest an evaluation method for the agent communication and migration costs, considering the different types of system resources and utilization.**

Keywords*— *Mobile Agent, Interoperability, Collaboration, Internet of Things, Wireless Sensor Networks, Constrained Application Protocol

I. Introduction

The Internet of Things (IoT) refers to globally connected and interactive network of physical and virtual devices, featuring the integration of disparate technologies and distributed intelligence [1]. The IoT systems require scalability beyond millions of devices, where centralized solutions easily reach their bounds. To achieve global connectivity, standardized protocols and interfaces are necessity to address device heterogeneity and to enable universal access. Resources provided by the IoT devices need to be globally identified, addressable and discoverable. Information about services, their functionality and interfaces needs to be discoverable. The services should be loosely-coupled and support mobility, but at the same time maintain quality of service constrains. Moreover, wireless sensor networks (WSN) with limited resource capabilities, can provide application-specific services as a single entity or collaborate as a part of IoT [2, 3]. In IoT, WSN can be utilized to collect contextual and environmental information [4] and to monitor phenomena and interactions. WSN introduce their own requirements, such as lightweight in-network services and low operating system and communication overhead, where energy consumption is the main concern. Therefore, in the collaborative WSNs, the key issue is how to design collaboration modes for resource-constrained WSN nodes that could optimize the resource utilization [2]. These collaborative systems cannot be instantiated and configured once before deployment, as the devices, services, applications and system configurations are in continuous transition. In the runtime deployment of components and composition of services, it is needed to consider software adaptation and evolution to cope with environment and requirement changes. The major issues here include interoperability between different standards, protocols, data formats, resource types, heterogeneous hardware, software components, database systems and finally human operators [5].

In this context, agent-based systems provide de-centralization and flexibility in the system configuration, abstracting heterogeneous subsystem for integration, cooperative multi-agent systems and high-level abstractions of system resources [2, 5-9]. Agents act autonomously, possess self-properties and allow the direct manipulation of the hosting device. Mobile agents, i.e. autonomous programs that transmit their execution state from device to device in networked systems where the execution of the program then continues, provide robustness, adaptation and evolution. Communication costs are reduced when distributed data processing is moved close to the data source, software components are deployed dynamically and tasks are executed asynchronously [2, 10].

However, this complexity of interactions cannot be anticipated by humans, which greatly complicates the system design and eventually requires the use of metadata and ontologies [5]. The role of humans should be minimal in system management, but also in its behavioral control and coordination [5]. Nonetheless, humans play important roles, chancing them depending on the context; as users, as resource managers, as service providers and as system administrators. Therefore a “smart interface” is required for humans to access various services and applications [5]. For human-machine interactions, abstracting IoT system and WSN as Web Services has benefits: various visualization services for simplified data search, retrieval and aggregation, access to contextual information, uniform interfaces for resource access and linked stateful resources [3, 5].

This work was conducted in the MAMMotH Project, funded by Tekes.

This paper presents interoperable mobile agents to provide collaboration and interoperability in IoT and programmable WSN, facilitating both heterogeneous IoT devices and low-power resource-constrained WSN nodes. This topic has received little attention so far. Adaptable composition is suggested for the mobile agent, where the composition is based on Representational State Transfer (REST) principles, additionally used for agent migration, controlling the agent and for system resource access. Moreover, we present an application programming interface (API) and system architecture components to enable interoperability. A proxy component allows connectivity between disparate networks and translates the messages, including the agent composition, between HTTP and Constrained Application Protocol (CoAP) [11]. The proxy additionally enables Web service access to the system resources, enabling human-machine interactions.

The rest of the paper is organized as follows. Section II describes the requirements for interoperable mobile agents in IoT and WSN. In Section III, is presented the mobile agent composition. In Section IV, we present the API for utilizing mobile agents in heterogeneous systems and system design considerations. A real-world prototype enabling these mobile agents is demonstrated in Section V and evaluated in Section VI. Section VII gives the related work and, in Section VIII, we discuss this approach and the future work.

II. Requirements for Mobile Agents

We gather the requirements for interoperable mobile agents in heterogeneous resource-constrained embedded networked devices. We also consider the REST principles in the agent composition, migration and control.

Scalable system configuration. In IoT, the systems are scalable beyond current networked systems. It is assumed a vast number of devices and tasks running in parallel, consuming the dynamically available system resources in competitive manner. Therefore, distributed architecture is necessity, services should be loosely-coupled and computational load distributed among the devices. Gateways and proxies are introduced to allow access to abstracted resources and heterogeneous subsystems far away, spanning over networks, protocols and communication interfaces.

Abstracted system resources. System resources, hosted by the devices, are the main abstraction in REST, consisting of the resource URL, its state and various representations. Here, the computational task is abstracted as the agent composition, whose state is the representation of the intermediate task result. The agent includes the actual functionality, i.e. the computation code, to create in-time representations of the task state.

Abstracted heterogeneous devices. The system devices expose the available resources based on their dynamic capabilities. These local and global resources include data, hosted resources and particular device capabilities. The device is utilized through basic, standardized, communication primitives with unified interfaces, where the primitives should be interface and protocol independent.

Standard interfaces. Standard, unified and simple interfaces are required to address device heterogeneity, resource abstraction and for universal access. To enable access from the Web, HTTP interface or a proxy component is required. To simplify the system implementation, the agent transfer and agent messaging protocols should be the same, based on the basic communication primitives.

Dynamic binding of system resources. The devices are simultaneously servers for their local resources and clients for the resources hosted by other devices. The agents should, as abstract compositions, allow dynamic binding to resources and dynamic mapping of the task into any system configuration for the lifetime of the task. In distributed systems, with heterogeneous devices the support for different types of resource bindings is required, even simultaneously for varying resource types. The agent composition should, in general, be exposed to the system by the devices and be modifiable and adaptable. Runtime lookups and loose coupling to the resources is facilitated by stateless communication.

Dynamic agent deployment. As the IoT systems are in continuous transition, runtime injections of agents into the system are common and the agent life-cycle is application-dependent. Therefore, the agent composition needs to adapt and be robust.

Lightweight mobile agent. The agent must be lightweight in composition, serializable, transferable as a whole or as sequential parts and executable in embedded devices with limited processing power, memory, communication capabilities and battery lifetime. For the most resource-constrained devices binary message formats are necessity.

Shared task state. The devices maintain their resource and capability states, the agents maintain their task states, which both are then collaboratively utilized by other agents. As the task state is not tightly-coupled into a physical device, the agent provides some robustness and the task state is cacheable.

III. Mobile Agent

We present the agent composition model, facilitating dynamic system configuration based on available resources and modifying the composition in runtime. The agent composition is based on the mobile code execution unit description in [12]: code segment, data space and execution state. To comply with the REST principles and Web Services paradigm, we refer to these as code, resource and state segments. In the following, we elaborate the composition and present the detailed use of the mobile agents in heterogeneous device platforms. The composition is illustrated in Table I. In addition to the given three segments, the data structure includes an unique name for the agent to allow dynamic lookups in runtime. The segments may also include metadata, such as the last time the agent state was updated or the keys for system and resource access.

Code segment. The computation task code is stored into and accessed from the code segment, which can be static but it is not required. The code can be presented in any programming language: high-level macroprogramming language, scripting language, precompiled intermediate code such as bytecode or even as machine language instructions. The code segment facilitates including the code, with known type parameter, in several different programming languages, being intended for multiple heterogeneous devices or execution environments simultaneously, regardless of the devices operating systems or hardware. Additionally, to minimize message size, this segment can include a reference to the location of the code for on-demand retrieval. This addresses the device heterogeneity.

Resource segment. The resource segment contains the local, remote and static resources for the task execution. The local resources refer to the resources offered by the hosting device and remote are external to the hosting device. The static resources are remote and static during the lifetime of the agent. These resources are then mapped to the code in the code segment as variables. The resource references are presented as URLs, with device address and resource name, to comply with Web and REST principles. The remote resources should be addressed using global identifiers, as they should be shareable and globally accessible from anywhere in the system. Additionally, access control can be applied in context of the current task or application. The metadata can contain the access interface and content-type of the resource. The hosting device then needs to bind to the local resources and retrieve the remote and static resource representations when it receives the agent. In case of static resource, the representation is requested only once and moved into the state segment as a constant.

The resource access allows binding-by-identifier, binding-by-value and binding-by-type [12] applied to the system resources. When binding-by-identifier, the agent requests resource representation from particular device or other agent. When binding-by-value, the agent requests the representation from any device hosting the resource through a name server. When binding-by-type, it requests the representation of any system resource of the specific type. Re-binding to the remote resources is required whenever the agent migrates from a device to another. The resource segment therefore presents dynamic and partial view of the system resources utilized by the agent.

Here, we have two options for storing the resource segment contents as list. First, the list does not exist outside the agent. Secondly, the list can be exposed as a system resource. Then, devices can request the participating device addresses from the hosting device with a resource query or utilize it as "home location" for the resources. This additionally allows network devices, proxies or gateways to modify the resource segment outside the agent composition to adapt to runtime changes in the system configuration. The agent may utilize the resource segment in multiple migration policies freely.

State segment. The state segment maintains the current state of the agent, i.e. the intermediate or final result of the computational task. In the hosting device, the agent is registered into the name server while migrating in the system, allowing resource queries for the agent state. Other local data, such as program counter, variables and static resources are moved to this segment, as unattached resources from the resource segment once the representations are known. These resource representations are later mapped into the task code as variables. This segment allows all data types or data structures to be used, depending solely on how the task code handles these variable types. Even different and multiple types of the same data can be used for different programming languages.

Agent mobility. The local resource segment describes how the agent migrates in the system. By utilizing the local resource URLs, the agent migrates from the device hosting the particular resource to another device. Moreover, we introduced two migration policies, thus the agent mobility is resolved by the local resource segment and the given migration policy. If the policy is "service", then the agent migrates to each device the local resource segment in turn, until specifically deleted. If the policy is "task", then the agent migrates to each device only once. With the "service" policy, the local resource segment is considered a ring buffer, where the next resource URL is in the top and URLs shuffle from top to bottom. With the "task" policy, the URLs are deleted from the segment once visited.

When migrating, the agent is cloned in the destination device, where the state is then updated and the hosting device registers the agent into the RD. After successful registration, an acknowledgement is sent to the previous host, which then deletes the agent from its memory. This is implemented to make sure that the previous state of the agent is always available until state update has finished.

A. *Mapping Agent to the CoAP Message Structure*

As CoAP is binary protocol, the agent composition is significantly smaller in size than the composition in human-readable scripting languages and data representations used in the Web, such as JSON. The mapping of the agent composition into the CoAP message structure is presented in Table II. This mapping extends our previous work [13], where we did not consider mobile agents. For the IPv6 addresses, we assumed globally known prefix (8 or 12 bytes) for the network, which

TABLE I. THE MOBILE AGENT COMPOSITION

Segment	Elements	
Name	{ Agent name, i.e. unique resource name }	
Code	Code	{ Task code }
		{ Programming language }
	Reference	{URL}
		{ Programming language }
Resource	Local	{ Resource list as URLs }
	Remote	{ Resource list as URLs }
	Static	{ Resource list as URLs }
State	{ State variable list }	
	{ Local variable list }	
	{ Metadata: API-Key for access control}	

TABLE II. MAPPING THE AGENT COMPOSITION TO COAP MESSAGE STRUCTURE

Agent Composition	CoAP Option	No.	Size (bytes)
Mobile Agent	Content-type: Task / Service	1	1
Name	Uri-Path (not new)	1	k
Local resource segment	Next Address	[0..n]	n * 8 (IPv6)
		[0..m]	m * 4 (IPv4)
Metadata: Access control	API-Key	1	n
Task code	Code	[0..1]	n
Task code reference	Code Reference	[0..1]	k
Remote resource segment[a]	Remote Resource	[0..n]	n * (8 + j)
Static resource segment[a,b]	Static Resource	[0..n]	n * (8 + j)
State segment	Payload	1	n

a. The address length is 8 bytes and j is the length of the particular resource name

b. Static resources are not moved into the state segment in the CoAP message has been omitted from the CoAP option address field. We utilize the CoAP option Content-type to identify the message as mobile agent and its migration policy. Additionally, we introduce a number of new CoAP message options to describe the agent composition. If the task code is not included in the message, the reference contains the resource name for remote retrieval, where the programming language type has been omitted as it is known by the requesting devices. The code, resource and state segments can contain any number of items. The state segment is stored as the message payload, being the actual transferable content, which cannot be omitted. The length of the API-Key for access control is application-specific and the security considerations are outside of the scope of this paper. Considering the CoAP message size, large messages can be transmitted with CoAP Block transfer -mechanism, however this would increase the agent migration latencies.

IV. Application Programming Interface

The API facilitates both inter-device and inter-agent communications, where the implementation is specific to programming language, operating system and the device platform. We follow the agent execution management interface outlined in [10], but we consider the basic methods of HTTP and CoAP protocols for Web-connectivity. Lastly, we outline system design considerations for the REST-based approach.

The API methods provide the following functionality: basic communication primitives, agent composition serialization, agent control, resource binding and task execution methods. See Figure 1. The communication methods relate to the HTTP and CoAP methods [11], with additionally content negotiation. Therefore, the agent transfer and agent messaging protocols are same, realizing the requirement for a uniform interface. The communication primitives, resource binding and serialization methods are used by the hosting devices. Mapping, task execution and control methods are utilized by the execution environment (EE) in the hosting devices.

Post. This method is used to transmit the agent to the next participating device based on the resource segment addresses, through the communication API of the device.

Get. This method is used to respond to local resource queries and to request remote and static resource representations. With the remote resources, it may be needed to first perform resource lookup into the name server.

Delete. This method deletes the task from the hosting device, effectively removing the agent from the system.

Register / Unregister. This method registers the device, its resources and capabilities into a name server, or removes the entries from the name server. Whenever the device is hosting an agent, its URL, i.e. the name with the device address, is registered into the name server. This allows dynamic lookups for the mobile agent and its state. The address of the name server should be globally known by the system devices.

Marshal / Unmarshal. These methods handle the serialization or de-serialization of the data structure in device memory into the agent messages, to be utilized by the device communication API. This internal data structure is used to store

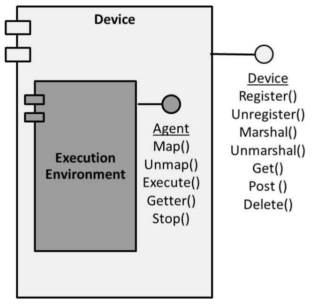

Figure 1. The interfaces to access resources and run agents.

the binding of the remote and local resources before and after the task code execution.

Map / Unmap. This method will map the internal data structure of local and remote resources to variables into the device memory utilized to construct a runnable code object. Unmapping will retrieve the updated data from the device memory after the code execution and update the internal data structure accordingly.

Execute. This method will execute the runnable task code object.

Getter. This method is used to retrieve the intermediate state of the task from the hosted agent, to answer agent state requests from other system devices.

Stop. This method can be called within the EE to stop the task execution and immediately transfer the agent. This allows the agent to control its' own execution from the task code.

A. System Design Considerations

As a part of IoT, WSN middleware should facilitate general and non-specific design solutions for applications. When connecting WSN to IoT, three approaches are discussed in [4]: a single gateway in between networks, dual-mode WSN nodes with several network interfaces in mesh topology and Internet access in one-hop through Wi-Fi access points.

A number of interaction models for WSN programming and resource access in general have been presented in literature: client / server, publish / subscribe, code migration, task offloading, MapReduce, cyber foraging, virtual machines and macroprogramming languages. When some of these are not feasible for the most resource-constrained WSN nodes, they can be considered use cases for interoperable mobile agents. With the flexible structure of the resource segment, devices and intermediates can modify the agent composition. When the agent migrates, the state and resource requests from the other agents and devices facilitate the client / server -paradigm. The publish / subscribe -paradigm can be achieved by CoAP Observe –mechanism within and from the WSN. MapReduce can be implemented by cloning the agent, or by broadcasting the resource requests to the system devices. Partitioning the code into smaller computational units can be considered before sending the task to the devices. Macroprogramming languages are fairly supported as the high-level code abstractions and primitives can be introduced to the system as on-demand task code, exposed as global system resources and accessed from a repository. Then, these primitive's representations can be considered methods or links in the code.

With the REST-based approach, we are able to expose the computational task itself and its intermediate or final result as system resources. The agent composition, or parts of it, can be exposed also, which other devices or agents can then utilize in their tasks or compositions through the references. This way, we provide code re-usability and modularization, furthermore allowing partitioning the computational task into smaller units, to distribute the computational load into the available system configuration. Moreover, the dynamic modification capability assists in adapting to system configuration changes in runtime and network or device failures. The mapping between RESTful Web Services API to the resources in WSN follows easily, as CoAP is as well based on REST principles and provides URIs for the resources and content negotiation. This enables direct access to the WSN resources from the Web and client / server communication to the in-network services in the WSN, ultimately facilitating human-machine interactions.

B. Controlling the Mobile Agent in System Devices

Next, we present examples of the REST API use for mobile agent creation and control, and for resource access based on HTTP requests to the proxy.

1) GET http://proxy_addr/resource

This request will first locate the hosting device for the resource identified by the *resource* from the resource directory, and then query its state from the hosting device. Error is returned if no such resource exists in the system. The parameter *resource* can present an agent name, in which case the hosting device returns the agent state.

2) POST http://proxy_addr/resource/agent_name

Here, we inject an agent into the system. This request is used when the proxy exposes application-specific resources, which require mobile agent creation to be utilized. If no agent composition is provided in the message body, the agent will make a resource lookup, based on the *agent_name* into the code repository and *resource* to the name server. If the resources are found, the agent can compose the agent automatically from these resources. If the *resource* element is omitted, both lookups are based on the *agent_name*.

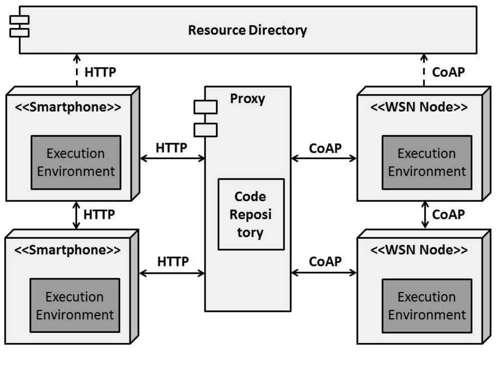

Figure 2. Prototype system architecture.

If the agent composition is included, and message translation is required, the existing resource addresses are inserted into the CoAP message in given order, but HTTP addresses last. The state segment is inserted into the message payload, whereas static resources as are inserted as message options. In case of code references, proxy may retrieve the code from the repository or leave the reference into the message. Otherwise the code is added as it is.

3) DELETE http://proxy_addr/agent_name

This method will first locate the hosting device for the agent identified as *agent_name*, and then requests the agent deletion from the device and from the resource directory.

4) POST http://repo_addr/ agent_name?type={platform}

This request will inject the particular code to the repository, for the given platform identifier, for lookups. The code itself is included in the message body.

5) Additional query parameters

We introduce additional query parameter: *device={list of devices}*, with the list of the unique identifiers of the devices located from the resource directory, to directly manipulate agents and resources in these particular devices. The proxy will then only send GET and POST requests to these devices only. This also applies to the resource segment in the agent composition.

We also included a number of HTTP headers for content negotiation and controlling the access. Currently, we support Content-Type, Authorization and API-Key headers. Authorization allows access to the application-specific proxies in general and the API-Key allows access to the particular system resource or device.

V. REAL-WORLD SYSTEM PROTOTYPE

We have implemented our first system prototype with the mobile agents. We utilized Android 4.0 smartphones, Samsung Galaxy S III, as general-purpose IoT devices communicating with HTTP over Wi-Fi in 2.4 GHz band. On the other end, we have a real-world IP-based WSN atop 6LoWPAN in 868 MHz band [14]. The WSN node platforms are ATmega 1284P and 2560 8-bit microcontroller-based embedded devices, running in 18 MHz with 8 or 16 Kb of RAM. The nodes communicate by CoAP.

We have described the system architecture, in Figure 2, in our earlier work [13-14], which is generally based on the framework in IETF CoRE Working Group [11] and on continuations as the abstraction of the mobile agent. We utilized the resource directory (RD) in [15] as a name server to store the system resource descriptions. The Java-based proxy component abstracts and exposes the WSN as a RESTful Web service conforming with the presented API, translates the HTTP requests to corresponding CoAP messages and also the agent composition between message formats. A code repository is bundled with the proxy to host the task codes for the WSN nodes. The code in the repository is accessed through the RD as any other system resource. The agents migrate in the IoT devices as JSON data structure and in the WSN as CoAP messages.

We implemented an EE for the Android 4.0 in Java, supporting scripting languages JavaScript and Python as the task code. The SL4A Scripting Layer for Android, with Rhino (JavaScript) for Android and Python for Android, is used to run the scripts. The application listens to incoming HTTP requests based on the presented API, which will then invoke the system services: the communication API to handle the message for access both the local and remote resources and the particular engine to execute the script code. The application size for the Android EE is 5.02 megabytes.

For the WSN nodes, we implemented the EE in C language. When the agent message is received, the state segment is copied into a shared memory in RAM. The local and remote resources are accessed by the EE through the communication API and the retrieved values stored also into the shared memory. The shared memory is accessed in the code through 16-bit pointers as variables. Then, a lookup is performed to find if the agent code already exists in a slot the program memory; if it exists, the code is executed by a function call, otherwise the code is first flashed into a particular slot in the memory. In case of code reference, the code is first retrieved from the repository. The agent code in WSN is in the precompiled IntelHEX binary format. For example, the binary footprint size is 20 bytes for memcpy() function, which can be used for copying a memory chunk from the shared memory into the agent composition. The ATmega architecture allows self-programming of the program memory (Flash) without resetting the device by a function in the boot loader section, a crucial feature for the implementation of this work. A C language header file defines the pointers to sections in shared memory in RAM for each segment in the agent composition and the address of the function stop() in the EE program memory. The binary size of the EE is 28864 bytes, including the memory section for the task codes, consuming 3850 bytes of RAM. The ATmega data sheet gives the maximum write time to the program memory (Flash) being 4 ms.

VI. Evaluation

We conducted the small-scale evaluation of the agent operation latencies in the real-world prototype in Figure 2. We conducted 100 experiments with injecting mobile agents into the system with both migration policies. The agents updated state segment with local resource data in each migrated device. As the task code, we utilized the memcpy() function in the WSN nodes, and implemented a Python script for Android. The remote resource segment contained one resource to evaluate access latencies. The static resource segment was omitted. The CoAP message size for the agent composition was 62 bytes (with 4 byte IPv6 addresses) and the size of JSON data structure of 468 bytes.

We measured the latencies in communication, resource access between the system devices, agent migration and for the computational overhead in task execution, see Table III. The migration latencies have been indirectly calculated from the proxy measurements, because the other clocks were not synchronized. We did not consider the agent creation latencies, as in the prototype the agents were created by the proxy, but the agent creation latencies are dominated by the number of resource lookups and should linearly increase as in [17]. The RD access latencies are considered the same as proxy latencies as they were deployed into the same computer, but differing from the device to device resource access latencies. During the evaluation, we experienced considerably varying network conditions in the Wi-Fi, therefore the smartphone to smartphone results are only indicative. In the WSN, the average ping message round-trip time for single-hop distances was 213 ms.

TABLE III. LATENCIES IN THE MOBILE AGENT OPERATION

Operation	Median	Message size
Resource Access	***T_r (ms)***	***bytes***
Smartphone - Smartphone	1419	117
Smartphone - Proxy - WSN node	880	117 -> 8
Proxy - WSN Node	599	8
WSN Node - WSN Node	798	8
Computational Overhead	***T_k (ms)***	***bytes***
Smartphone	315 (20)	586
WSN Node	330	66
Agent Migration	***T_m (ms)***	***bytes***
Smartphone - Smartphone	2409	586
Smartphone - Proxy - WSN node	1856	586 -> 66
Proxy - WSN Node -Proxy	1299	66
WSN Node – WSN Node	1302	66

Resource access. We measured the latencies accessing resources from smartphone to another smartphone, smartphone to WSN node though the proxy, from proxy to the WSN node and from WSN node to another. The third measurement gives the latency for any Internet device to access the WSN resources. This included the request processing time in the hosting device. The message translation times in proxy were omitted, but the HTTP and CoAP GET request message sizes were 117 with additional headers and 8 bytes.

Computational overhead. Here we measured the computational latency of executing the agent task code. The platform-specific latencies include time for system service invocation, marshaling and mapping the composition, running the code and recomposing of the message. In the WSN nodes, no system services needed to be invoked. The execution overhead in WSN, measured approximately as 330 ms, is generally the same for any agent or message handling. For the Android EE, the overhead for resources access was 20 ms and script execution time was 315 ms. For the resource access in the Android EE, it is not needed to invoke any extra system services. We did not consider in the prototype large agent compositions or messages with large payload.

Agent migration. This includes the overhead of agent registration into the RD by the hosting device, sending the agent message to the next device and waiting for acknowledgement, after which the agent is deleted from the memory. This does not include the computational overhead or resource access latencies. The HTTP POST request header size was 118 bytes with additional headers and 4 bytes for CoAP header. Comparing this result to the previous work in [7, 16], the prototype demonstrates 50-70% faster agent migration times over single-hop distances, but without guaranteed reliability and with smaller message size and payload.

A. *Conclusive Real-world Evaluation*

We propose a generic evaluation method for conclusive real-world evaluations in mobile agent-based systems. Otherwise, conclusive evaluation would be difficult to conduct, as the task configuration, device deployment, the required resources and their locations are largely application-specific [17], here additionally spanning over disparate networks. These equations are simplified as varying communication latencies, changing network conditions, device failures and resource availability are not considered.

In equation (1), we calculate the cost *C* of communication and task execution in particular execution environment *k*, taking into account the different resource utilization types in the agent data structure:

$$C_k = (r+1)T_r + T_k + T_m, \qquad (1)$$

where *r* is the number of remote resources, T_r is the response time for remote and static resource requests, T_k is the computational overhead and T_m is the agent migration time. T_r is added once for agent registration. The local resource response time is considered negligible.

The equation (2) gives the total migration costs C_{Total}, for a particular agent-based service. This includes the response time for static resource requests, where *s* is the number of static resources, and including the number of disparate networks *d*. The agent migration time between networks is given as $T_{m,d}$ for $C_{m,d}$. We include the time of message translation in the proxy as T_p. The number of devices running each execution environment is *n*:

$$C_{Total} = sT_r + \sum_{d}(T_p + C_{m,d}) + \sum_{d}\sum_{n-1}(C_n). \qquad (2)$$

Considering the evaluation agent in the prototype system in Figure 2 and our evaluation results, we can estimate the total service migration cost (*d*=2, *n*=2, *s*=0, *r*=1) with the above formulas, see Table IV. The overhead is proportional to the number of migrated devices and remote resource accesses. With these network conditions, the remote resource access times, 4434 ms, contribute 28% of the total cost. Therefore the remote resources, imaginably hosted in devices over disparate networks, should in the system design be considered as static or local resources as much as possible. Resource caching in the proxy, with indirectly calculated total access time 1100 ms, would significantly reduce (75%) the overhead. However, for improving service response times the agent states are always available for resource queries in this approach.

TABLE IV. AGENT MIGRATION TIMES IN PROTOTYPE SYSTEM

Evaluation Agent	**C_n (ms)**	**T_p (ms)**	**$T_{m,d}$ (ms)**	**$C_{m,d}$ (ms)**	**C_{Total} (ms)**
Wi-Fi	4423	1	1856	5009	16044
WSN	2830	1		3782	

VII. RELATED WORK

We consider the previous work with agent-based architectures for IoT. In [5], the authors envision agent-based IoT system architecture, where an agent represents each resource, monitoring and coordinating the resource use through specific roles. The tasks are written in a rule-based language, where the agents provide system configuration for the tasks and react to configuration changes. Semantic queries used for resource discovery. In [6], a multi-layered agent-based architecture for smart objects is presented, where a coordinator controls the agents in hosting devices. System heterogeneity is abstracted by layers and the coordinator solely communicates directly with the other smart objects or system devices. In [7], agents are used as gateways to access heterogeneous devices and communication protocols in IoT, enabling interoperability, context-awareness and one-to-many communication with centralized group management. In [8] is presented a multi-agent platform for embedded systems based on the Java Virtual Machine. The device platforms have static system agents providing interfaces to the system services and dynamic service agents running the smart home applications. In Agilla [9], mobile agents wait for specific events and when the event fires, agent migrates to the source node to run event-based task.

Next, we consider the previous work of programming WSN with mobile agents. In [16], Java-based mobile agent framework for SunSPOT is implemented, where agents are modeled as multi-plane event-based state machines. System components offer services for communication, agent control and notably for timing agent operations. The agent state transitions, i.e. computations, occur in response to events and if a condition holds, then new events can be emitted asynchronously. Evaluation of the framework with a real-time application for wireless body sensor networks is also provided. In [18], the authors present reference system architecture with coordinator nodes connected to the Internet, based on the work in [16]. Coordinators enable registered applications to receive events from the sensor nodes, provide means for register new sensors and services. For the sensor nodes, coordinators provide the abstractions of the system resources and a set of data processing tasks. The coordinator is very similar to the proxy functionality presented in this paper. In [19], platform-independent framework for mobile agent-based applications in collaborative wireless body sensor networks is presented. An agent is installed in the system devices to run the tasks as recipes. Task execution is controlled through API with specific messaging protocol, providing system adaptation. The publish / subscribe -paradigm is used for data dissemination.

Flexeo [20] provides layered architecture to connect WSN to IoT with Sensors and Actuators layer, Coordination layer and Supervision layer. The WSN architecture is multi-hop with sink nodes connected to the upper layer. The lowest layer provides data aggregation. The Coordination layer, based on OSGi, abstracts the WSN by REST API, allowing queries for device type or identifier. The intelligence is provided as domain-based rule sets triggering actions, where OSGi bundles are used for programming. The layer also handles connections to disparate networks. The Supervision layer provides centralized GUI for global view of the system.

In comparison, we present a novel, language- and platform independent, agent composition for heterogeneous systems. Our approach is based on open standards for communication over disparate networks and for collaboration support without specific interaction protocols or middleware. The system architecture is flat and is not restricted to specific interaction model. Centralized system configuration or management is not facilitated and we do not apply any specific system or task configuration into the devices, but we expose the agent composition into the system. We facilitate dynamic interlinked many-to-many communications despite the roles of the agent or devices. Furthermore, we utilize REST principles for agent migration, agent control and exposing system resources to the Web, realizing the same protocol for agent transfer and messaging. Lastly, although Java software components are modular, portable and provide object-oriented features for programming, virtual machine-based solutions may be too heavy for the most resource-constrained embedded devices. We omitted the discussion of the integration of WSN to the Web, as we follow IETF CoRE WG's work.

VIII. Discussion and Future Work

This work demonstrated the integration of IoT and WSN with mobile agents. The expected benefits include: agent-based adaptable service composition is facilitated, the computational load is distributed, locality can be exploited in communication and system resources are exposed to the Web for human-machine interaction.

In a real-world prototype system, we utilized a proxy component as a gateway to connect IP-based multi-hop WSN nodes to Internet, whereas generic IoT devices are directly connected through Wi-Fi. However, we only considered IP protocol stack here. In the agent migration we utilized straightforward migration policies and did not consider, for example, device or network failures, varying network conditions, requiring more sophisticated and reliable agent migration methods. Caching would also significantly reduce the experienced latencies. The formulas presented in Section V are very generic, but nevertheless can offer assistance in application-specific system design and service response time estimations. Additional system-specific parameters should be introduced to real-world evaluations. Currently, data streaming is not supported and the real-time capabilities are unknown. The security and privacy issues were omitted in this work. The future work includes deployments in real-world environment and evaluation with novel mobile agent-based applications.

References

[1] L. Atzori, A. Iera and G. Morabito, "The internet of things: a survey," Computer Networks, vol. 54, no. 15, October 2010, pp. 2787-2805.

[2] W. Li, J. Bao and W. Shen, "Collaborative wireless sensor networks: a survey," In: IEEE International Conference on Systems, Man, and Cybernetics, pp.2614-2619, Anchorage, AL, USA, October 9-12, 2011.

[3] M. Gomes, H. Paulino, A.Baptista and F. Araújo, "Dynamic interaction models for web enabled wireless sensor networks," In: 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 823-830, Madrid, Spain, July 10-13, 2012.

[4] C. Delphine, A. Reinhardt, P. Mogre and Ralf Steinmetz, "Wireless sensor networks and the Internet of Things: Selected challenges," In: Proceedings of the 8th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze, pp. 31-34, Hamburg-Harburg, Germany, August 13-14, 2009.

[5] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin and V. Terziyan, "Smart Semantic Middleware for the Internet of Things," In: Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization, pp. 169-178, Funchal, Portugal, May 11-15, 2008.

[6] G. Fortino, A. Guerrieri, and W. Russo, "Agent-oriented smart objects development," In: 16th IEEE International Conference on Computer Supported Cooperative Work in Design, pp. 907-912, Wuhan, China, May 23-25, 2012.

[7] I. Ayala, M. Amor, and L. Fuentes, "An agent platform for self-configuring agents in the Internet of Things," In: 3rd International Workshop on Infrastructures and Tools for Multiagent Systems, Valencia, Spain, June 5, 2012.

[8] E. Kazanavicius, and L. Ostaseviciute, "Agent-based framework for embedded systems development in smart environments," In: 15th International Conference on Information and Software Technologies, pp. 194-200, Kaunas, Lithuania, April 23-24, 2009.

[9] C. Fok, G. Roman and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," ACM Transactions on Autonomous and Adaptive Systems, vol. 4, no. 3, 16, 2009.

[10] I. Satoh, "Mobile agents," In: Nakashima et al (eds.) Handbook of Ambient Intelligence and Smart Environments, Springer, 2010, pp. 771-791.

[11] Constrained RESTful Environments. (Accessed: 3rd August 2013). https://datatracker.ietf.org/wg/core/charter/.

[12] A. Fuggetta, G. Picco and G. Vigna, "Understanding code mobility," IEEE Transactions on Software Engineering, vol. 24, no. 5, May 1998, pp. 342-361.

[13] T. Leppänen, P. Närhi, J. Ylioja, J. Riekki, Y. Tobe and T. Ojala, "On using continuations in wireless sensor networks," In: 9th International Conference on Networked Sensing Systems, pp. 1-2, Antwerp, Belgium, June 11-14, 2012.

[14] T. Leppänen, J. Ylioja, P. Närhi, T. Räty, T. Ojala and J. Riekki, "Holistic energy consumption monitoring in buildings with IP-based wireless sensor networks," In: 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys2012), pp. 195-196, Toronto, Canada, November 6, 2012.

[15] M. Liu, T. Leppänen, E. Harjula, Z. Ou, A. Ramalingam, M. Ylianttila, and T. Ojala, "Distributed resource directory architecture in machine-tomachine communications," In: IEEE WiMob 2013, Workshop on Internet of Things Communications and Technologies, Lyon, France, October 7-10, 2013. [To appear].

[16] F. Aiello, G. Fortino, R. Gravina and A. Guerrieri, "A Java-based Agent Platform for Programing Wireless Sensor Networks," Computer Journal, vol. 54, no. 3, March 2011, pp.439-454.

[17] S. Malek, N. Medvidovic and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," IEEE Transactions on Software Engineering, vol. 38, no. 1, January- February 2012, pp. 73-100.

[18] F. Aiello, F. Bellifemine, G. Fortino, S. Galzarano and R. Gravina, "An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks," Journal of Engineering Applications of Artificial Intelligence, vol. 24, no. 7, October 2011, pp. 1147-1161.

[19] G. Fortino, A. Guerrieri, F. Bellifemine and R. Giannatonio, "Platform-independent development of collaborative wireless body sensor network applications: SPINE2," In: IEEE International Conference on Systems, Man, and Cybernetics, pp. 3144-3150, San Antonio, TX, USA, October 11-14, 2009.

[20] J. Vazquez, A. Almeida, I. Doamo, X. Laiseca and P. Orduña. "Flexeo: an architecture for integrating Wireless Sensor Networks into the Internet of Things." In: Corchado et al (eds.) Advances in Soft Computing, , vol. 51, pp. 219-228, Springer, 2009.