Teemu Leppänen

RESOURCE-ORIENTED MOBILE AGENT AND SOFTWARE FRAMEWORK FOR THE INTERNET OF THINGS

UNIVERSITY OF OULU GRADUATE SCHOOL,
UNIVERSITY OF OULU,
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

UNIVERSITY OF OULU P.O. Box 8000 FI-90014 UNIVERSITY OF OULU FINLAND

ISBN 978-952-62-1813-7 (PDF)
ISSN 0355-3213 (Print)
ISSN 1796-2226 (Online)
TEEMU LEPPÄNEN

RESOURCE-ORIENTED MOBILE AGENT AND SOFTWARE FRAMEWORK FOR THE INTERNET OF THINGS

Academic dissertation to be presented with the assent of the Doctoral Training Committee of Technology and Natural Sciences of the University of Oulu for public defence in Auditorium L7, Linnanmaa, on 23 March 2018, at 12 noon

UNIVERSITY OF OULU, OULU 2018
Abstract

The Internet of Things vision proposes a global platform in an unforeseen scale for distributed applications that rely on data provided by interconnected resource-constrained things. In such large-scale systems, centralized control of system operation by a single component through vertical interactions becomes unfeasible. Ideally, decentralized control in the proximity of things enables to take into account the local dynamic resource availability and environmental characteristics that are used to optimize the application execution. To realize decentralization, capabilities for horizontal interactions, that complement the vertical interactions, and for opportunistic participation of things are needed.

This thesis explores mobile agent technology to implement distributed Internet of Things applications that benefit from both vertical and horizontal interactions of the application components. First, a resource-oriented reactive mobile agent architecture and a software framework are presented. The framework facilitates RESTful interactions between agents and other system components and provides a REST-based interface to build opportunistic agent-based applications. Two agent platforms are presented that integrate resource-constrained things into the framework as mobile agent hosts. Second, mobile agent based mobile crowdsensing and edge computing applications are evaluated with large-scale simulations and real-world experiments. The results show that energy consumption is decreased in the participating things, in comparison with the existing approaches, by agent-based in-network data processing and control of the thing operation.

This thesis makes a valuable contribution by enabling mobile agent operations in a heterogeneous set of resource-constrained things. The presented empirical evidence shows how mobile agent technology increases energy efficiency in distributed application execution. The presented mobile agent architecture and software framework potentially accelerate the utilization of mobile agent technology in the Internet of Things.

Keywords: agent architecture, agent platform, energy efficiency, internet of things, mobile agent, multi-agent systems, reactive agent, representational state transfer, software framework
Tiivistelmä

Esineiden Internet liittää resurssirajoitteiset sultautetut laitteet laajamittaisesti Internet-verkkoon, jossa niiden tuottamaa tietoa hyödynnetään hajautettujen järjestelmien soveltukseissa. Esineiden Internetin järjestelmien odotetaan skaalautuvan niin laajoiksi, ettei keskitetty, vertikaaliseen vuorovaikutukseen perustuva järjestelmähallinta ole enää käytännössä vaikuttavaksi. Hajautettu hallinta lähellä tietoa tuottavia laitteita tuo etuja, kun paikallisesti sovelluksen suoritus on otettava huomioon toimintaympäristön tila ja voidaan reagoida dynaamisesti resurssien saatavuuteen. Hajautus Esineiden Internetin sovellusalkioilla edellyttää menetelmiä sekä laitteiden vertikaaliseen ja horisontaaliseen vuorovaikutukseen, että niiden dynaamisen osallistumisen mahdollistamiseksi sovelluksen suorittamisessa.


Työssä esitetty resurssisuuntautunut mobiiliagenttiarkkitehtuuri sekä ohjelmistokehys edesauttavat mobiiliagentti-teknologian hyödyntämistä resurssirajoitteisissa sultautuissa laitteissa. Toisella malleilla saatut tulokset osoittavat mobiiliagentti-arkkitehtuurin hyötyjä hajautettujen sovellusten toteuttamisessa Esineiden Internetin.

Asiasanat: agenttialusta, agenttiarkkitehtuuri, energiatehokkuus, esineiden internet, mobiiliagentti, moniagenttijärjestelmät, ohjelmistokehys, reaktiivinen agentti, REST-arkkitehtuurimalli
Acknowledgements

This study was carried out during the years 2012 to 2017 with the iSpaces research group at the Department of Computer Science and Engineering, University of Oulu, which later evolved to the Center for Ubiquitous Computing with other groups.

First of all, I am grateful to my supervisor, professor Jukka Riekki, for his valuable advice, clarifying discussions and guidance throughout my doctoral studies. I would also like to thank professor Timo Ojala for providing an energetic research environment in the projects and academic activities led by him. I am grateful to professor Kaoru Sezaki, the University of Tokyo, Japan, for accepting me as a research associate to visit his laboratory during my doctoral studies. I am indebted to professor Yoshito Tobe, Aoyama Gakuin University, Japan, for his friendship and co-operation throughout the years.

This research was conducted in the MAMMotH project and the Digile IoT research program. Additionally, to support this research, personal grants from the University of Oulu Graduate School, the Pohjoista voimaa - Ympäristötööli and the Academy of Finland - International Researcher Mobility are gratefully acknowledged. Partly, this research was conducted during a one year research visit to the Sezaki Laboratory, Institute of Industrial Science, the University of Tokyo, Japan.

I would like to acknowledge all my colleagues who have worked with me in the research projects and within the research groups. Dr. Susanna Pirtikangas deserves special thanks for originally inviting me to finish my Master’s degree studies with the iSpaces group. Particularly in the MAMMotH project, I worked towards this thesis with Dr. Meirong Liu, Dr. Erkki Harjula, Mr. Pauli Närhi, Mr. Jani Ylioja and Mrs. Archana Ramalingam. Mr. José Álvarez Lacasia deserves thanks for our inspiring co-operation at the University of Tokyo. I would like also to thank the students at the Sezaki laboratory for a truly memorable year in Japan. The chairman of my follow-up group, Dr. Denzil Ferreira, kindly provided clarifying comments for the manuscript of this thesis. Dr. Pertti Väyrynen deserves thanks for proofreading the manuscript.

I wish to express my sincere thanks to professor Gregory O’Hare, University College Dublin, Ireland, and associate professor Paolo Bellavista, University of Bologna, Italy, for reviewing this thesis and providing perceptive comments to clarify the results.
presented in the thesis. I respectfully thank professor Giancarlo Fortino, University of Calabria, Italy, for serving as the opponent in my doctoral defense.

My academic studies have been a long journey, with more than a decade’s stint in the national information technology industry before returning to the university to work in research. The journey now culminates in this thesis. I wish to thank Sanna, who lived with me the years of distress and joy during which time this thesis was formed. I am deeply thankful to my family, Kerttu, Pentti and Veera and her family, for their encouragement during the years. Particularly insightful have been the discussions about the ups and downs of a life in academia.

Oulu, January 2018

Teemu Leppänen
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>ADS</td>
<td>Agent Directory Service</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AMS</td>
<td>Agent Management System</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BDI</td>
<td>Belief-Desire-Intention</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol</td>
</tr>
<tr>
<td>CoD</td>
<td>Code on Demand</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CoRE</td>
<td>Constrained RESTful Environments</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>C/S</td>
<td>Client-Server</td>
</tr>
<tr>
<td>DF</td>
<td>Directory Facilitator</td>
</tr>
<tr>
<td>DRD</td>
<td>Distributed Resource Directory</td>
</tr>
<tr>
<td>EE</td>
<td>Execution Environment</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTML5</td>
<td>Hypertext Markup Language, version 5</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent Development Framework</td>
</tr>
<tr>
<td>JADE-LEAP</td>
<td>JADE Lightweight Extensible Agent Platform</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-Agent System(s)</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>MEC</td>
<td>Mobile Edge Computing</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>MIME</td>
<td>Multipurpose Internet Mail Extensions</td>
</tr>
<tr>
<td>MTP</td>
<td>Message Transport Protocol</td>
</tr>
<tr>
<td>MTS</td>
<td>Message Transport Service</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>REV</td>
<td>Remote Evaluation</td>
</tr>
<tr>
<td>RFC</td>
<td>Request for Comments</td>
</tr>
<tr>
<td>ROA</td>
<td>Resource-Oriented Architecture</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WoT</td>
<td>Web of Things</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network(s)</td>
</tr>
<tr>
<td>WS-*</td>
<td>Web Service Specifications</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
List of original publications

This thesis is based on the following four original articles published in a peer-reviewed international conference, a research book and journals. The articles are referred to throughout the text by their Roman numerals (I-IV):


Publications I and II propose a resource-oriented mobile agent architecture and software framework based on embedded Web services for the Internet of Things. Publications III and IV address mobile agent based energy efficiency in mobile crowdsensing and edge computing applications atop the framework.
Contents

Abstract ................................. 3
Tiivistelmä ................................ 7
Acknowledgements .......................... 7
Abbreviations ............................ 9
List of original publications .................. 11
Contents ................................ 13
1 Introduction ............................ 15
  1.1 Background and motivation .................... 15
  1.2 Research questions and scope .................. 19
  1.3 Research methodology ....................... 23
  1.4 Contributions of the thesis ................... 24
  1.5 Research history .......................... 26
  1.6 Structure of the thesis ...................... 27
2 Theory and concepts ..................... 29
  2.1 Software agents ............................ 29
    2.1.1 Multi-agent systems ...................... 33
    2.1.2 Mobile agents ........................... 37
    2.1.3 Programming mobile agents ............... 43
  2.2 Internet of Things ........................ 46
    2.2.1 Use cases for software agents .......... 50
    2.2.2 Resource-constrained things ............. 54
  2.3 Resource-oriented architecture ............. 57
    2.3.1 Embedded Web services .................. 61
    2.3.2 Constrained RESTful Environments ....... 63
3 Research contributions .................. 67
  3.1 Resource-oriented mobile agent architecture . 67
  3.2 Mobile agent software framework ............ 74
  3.3 Mobile agent platform ....................... 79
  3.4 Energy efficiency with mobile agents ......... 88
4 Discussion ............................. 101
  4.1 Analysis of the results .................... 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 Limitations and generalization</td>
<td>106</td>
</tr>
<tr>
<td>4.3 Significance of the results and future work</td>
<td>108</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>111</td>
</tr>
<tr>
<td>References</td>
<td>113</td>
</tr>
<tr>
<td>Original publications</td>
<td>129</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background and motivation

“Artificial intelligence is the science of making machines do things that would require intelligence if done by men.”

Marvin Minsky, 1968

Software agents (Nwana 1996, Jennings 2001, Russell & Norvig 2003, Chapter 2) are one of the classical artificial intelligence (AI) paradigms in distributed systems. Software agents are computer programs that possess a degree of autonomy and intelligence, with their individual capabilities, to act and interact on behalf of their owners. While executing their tasks, agents observe their environment, react to the environment state and adapt to changes in the state. At best, agents learn from their own behavior and the environmental changes and operate proactively. As a multi-agent system (MAS), agents in a group are able to share their information and knowledge with each other to facilitate execution of complex tasks, when the capabilities of a single agent are limited, and collaborate towards a common goal (Weiss 2013).

Particularly, one software agent capability, mobility, is a compelling feature in the realization of agent-based distributed computing applications. Software agents having this additional capability are referred to as mobile agents (Cao & Das 2012, Chapter 1). Mobility enables agents to suspend their tasks, relocate themselves to another computer in the system, continue task execution there, and finally return task results to their owners. The benefits of agent mobility in general have been discussed in previous work, e.g. in Chess et al. (1997), Lange & Oshima (1999), Hurson et al. (2010). With in-network data processing, i.e. processing data in the devices producing it, mobile agents reduce the amount of data transmitted over the network. This, in its turn, contributes towards reduced bandwidth requirements, communication latencies and communication energy consumption. Systems become more scalable as reduced network load provides room for more devices to communicate. Mobility increases robustness and fault tolerance in the application execution, as agents migrate in response to changes in the environment, such as device failures, adapt to intermittent network connectivity and operate locally within disconnected network segments. Lastly, distributed applications
can be extended by mobile agents transmitting new application functionality over the network.

The Internet of Things (IoT) is the contemporary platform for large-scale distributed applications. IoT interconnects deployments of things across a wide range of technologies for information sharing and service provisioning to facilitate novel data-intensive applications (Atzori et al. 2010, Miorandi et al. 2012). A thing as a concept refers to a physical or a virtual object, i.e. an abstraction of an object, containing software and hardware components that are contributed as application resources. An example of a thing is a wirelessly connected embedded device equipped with sensors. Another example is a mobile thing, i.e. a modern smartphone equipped with sensors and also functioning as a user interface for human-machine interactions.

The predominant IoT system architecture of today is based on the cloud computing paradigm that provides global connectivity and virtually unlimited resources for both data- and computation-intensive applications (Gubbi et al. 2013, Borgia 2014). This type of architecture is illustrated in Figure 1. The cloud platform facilitates centralized management and configuration of things and other application components, large-scale data analysis, automatic data backups for long-term storage, and information security. Things at the edges of the network collect and upload their data to the cloud platform, either directly or through network infrastructure components. For data collection, things are organized as wireless sensor networks (WSN), where the nodes are either resource-
constrained stationary embedded devices or mobile devices, e.g. smartphones. For
data upload, the real-world wireless environment with lossy networks and intermittent
connectivity is a challenge (Shelby 2010). This is emphasized with battery-operated
things that have low computational and communication resources and are capable
of short-lived interactions only. Moreover, large-scale data upload burdens network
infrastructure components and the backbone network. As the data go all the way to the
cloud for analysis, latencies are introduced to the application execution (Shi & Dustdar
2016). The results may be returned to the same set of things that operate as user devices.
When the IoT systems scale up, this centralized cloud-based model is unlikely to stand
up scalability challenges for unforeseen variety of novel applications and technologies.

Architectural approaches have been proposed to address the challenges of the cloud
based model. Edge computing (Garcia Lopez et al. 2015, Shi & Dustdar 2016), as
illustrated in Figure 2, leverages cloud resources to the resource-rich infrastructure
devices between the cloud and things that participate in data dissemination, storage and
processing. These devices are called edge devices. Application execution is centrally
controlled but occurs collaboratively partly at the cloud and partly at the edge devices at
the close proximity of the users. The services at the cloud, which users invoke, are
transmitted over the network as application-specific self-contained virtual machines
(VM) to the edge devices. To address the challenges introduced by dynamicity in the
IoT systems, opportunistic computing (Conti et al. 2010, Guo et al. 2013) pools mobile
devices of users to share their resources horizontally for local application execution. A well-known example is smartphones sharing their physical sensors for data collection and their network connections for data dissemination. Yet, the interoperability of things and edge devices is a challenge in such a dynamic environment.

Software agents and MAS have been proposed for distributed IoT application execution in such dynamic environments (Uckelmann et al. 2011, Miorandi et al. 2012, Fortino 2016). Sharing the situational understanding from the centralized cloud to the edge and to the things, enables smart behavior in these devices (Gubbi et al. 2013). Software agents that operate on the individual things increase their capabilities towards interoperability and autonomous intelligent behavior, learning and proactivity. This, in its turn, facilitates the development of IoT system architectures, where distributed applications are executed by interacting and collaborating software agents that represent things and other system components. The autonomous behavior of agents in application execution and control reduces the need for human intervention. To this day, software agents specifically in the context of interoperability in IoT are largely unexplored, however.

The reasonable solution today to provide Internet-scale interoperability1 for IoT applications and human-machine interactions alike are the standardized Web technologies (Atzori et al. 2010, Borgia 2014, Want et al. 2015). The Web of Things (WoT) (Guinard et al. 2009) and embedded Web services (Shelby 2010) propose a conceptual framework to integrate things into the Web, where they operate simultaneously as clients and servers for each other. However, resource-constrained things are not capable to run Web servers that rely on service-oriented architectures (SOA), Web Service Specifications (WS-*) and full Hypertext Transfer Protocol (HTTP) specifications (Pautasso et al. 2008, Shelby 2010), but run servers with reduced functionality (Priyantha & Kansal 2008). Therefore, optimized Web protocols are needed for integration. The embedded Web services approach, as described in Shelby (2010), is currently being standardized by the Internet Engineering Task Force (IETF). The underlying idea follows the Representational State Transfer (REST) architectural principles (Fielding 2000) as a lightweight alternative to the SOA. However, the area of REST and embedded Web services with software agents is still an unexplored area.

1Interoperability is defined in Tanenbaum & van Steen (2007, p.8) as the extent by which separate systems and/or components can co-exist and work together by relying on each others’ services as specified by a common standard.
This thesis explores mobile agents for low-power resource-constrained embedded devices in the context of IoT, where embedded Web services provide seamless integration into the Web and facilitate interoperability between mobile agents and other system components. The practical aim is to enable mobile agent based distributed IoT applications and study their energy efficiency in comparison with current centralized approaches that rely on vertical data upload from things to the cloud.

1.2 Research questions and scope

This thesis aims at providing new knowledge regarding the integration of mobile agent technology into the IoT. According to the agent typology by Nwana (1996), the focus is on autonomous and collaborative reactive mobile agents. Agents are designed and programmed to execute tasks as components of distributed applications. Mobile agents may have designated owners, i.e. origin computer or users, but they operate and migrate autonomously in a set of host computers without explicit guidance. Mobile agents, in their task execution, react to stimulus from the environment and interact with system components and other agents as a MAS. Mobile agents share their information with each other, based on how their interactions are designed, and return the task results to the owners. Owners utilize these results for example to provide services.

In IoT, mobile agents are situated in an environment that is resource-constrained, open and dynamic. Due to heterogeneity of IoT system components, interoperability is a key concern. Therefore, embedded Web services (Shelby 2010) are selected as standardized means to integrate mobile agents, resource-constrained things and system services via the Internet. To increase the applicability of mobile agent technology, both stationary and mobile resource-constrained embedded networked devices are considered as mobile agent platforms in this thesis.

Such a system calls for a software framework that provides a set of abstractions and services for agent operations and interactions, and in addition, provides an interface for global system resource access in IoT. As discussed by Wooldridge & Jennings (1998), agent frameworks should be built on existing technologies as much as possible, and in the development of such systems, general distributed systems should not be forgotten, either.

Three fundamental code mobility paradigms have been presented in the literature for distributed systems (Fuggetta et al. 1998), as illustrated in Figure 3, that are today utilized also in the context of IoT. Client-server (C/S) model based on remote procedure
calls (RPC) is shown as a reference. A client can execute a procedure in a remote server transparently by issuing a local RPC that is transmitted over the network. In Remote Evaluation (REV), a client offloads code to be executed on a server. In this case, the client typically has no capabilities to run the program or transferring the required data over the network to the client is too resource-consuming. The results are then returned to the client. Code on Demand (CoD) refers to a client downloading a program from a server to be executed locally in the client. The REV and CoD code mobility paradigms are actively used to realize IoT edge computing. Edge application execution is optimized for low-end IoT devices by sharing computational tasks between participating devices and by offloading computations to the edge (Kumar et al. 2013, Shi & Dustdar 2016). These systems extend their precursor, peer-to-peer (P2P) systems (Rodrigues & Druschel 2010), by introducing stable components into the dynamic assembly of shared resources (Garcia Lopez et al. 2015). The REV and CoD paradigms are largely centrally controlled, which is not optimal for dynamic IoT environments.

Mobile agents, in turn, are programs that are injected from a client to the system and are capable of autonomously migrating between servers to execute their tasks, interacting with the environment and finally bringing the results back to the client. A distinct feature of mobile agents is that they carry their identity and state of their task in migration between servers, which enables to continue task execution in different servers. In addition to executing remote programs as in REV and CoD, mobile agents require support in the server to capture the agent state, transmit it and resume the program execution from the state.

Mobile agents have yet received little attention in IoT applications, in spite of the expected benefits described in the literature, e.g. Hurson et al. (2010). Previous research
on mobile agent technology has confirmed the benefits in reducing communication energy consumption and size of transmitted data in general. The benefits are seen particularly with WSN (Qi et al. 2001, Chen et al. 2006b), where continuous connectivity is challenging in dynamic environments (Akyildiz et al. 2002). When the system size grows, the centralized C/S model does not scale up well due to the increased data flow that demands more bandwidth and increases backbone network load (Qi et al. 2001, Chen et al. 2007, Vinyals et al. 2011). Moreover, mobile agents operate better than C/S interactions on mobile computing systems due to limited device resources and unpredictable mobility (Bellavista et al. 2001, 2002, Spyrou et al. 2004). The data producing IoT devices, i.e. resource-constrained WSN nodes and sensor-equipped mobile phones, could benefit from the distributed mobile agent based application execution.

The integration of mobile agents into these resource-constrained IoT technologies leads to the following research questions:

**RQ1:** How to realize mobile agents in low-power heterogeneous and constrained IoT devices with RESTful embedded Web services?

**RQ2:** How does the proposed solution influence energy efficiency in a descriptive set of IoT applications that rely on low-power IoT devices as data sources?

The scope of the thesis is defined as follows. The presented research provides a mobile agent architecture and a mobile agent software framework for IoT that incorporates resource-constrained things as mobile agent platforms. The mobile agent architecture follows the REST principles. The software framework merges two existing standards. On the one hand, the Foundation for Intelligent Physical Agents (FIPA) provides an abstract architecture for MAS (FIPA 2002a) with services to discover system resources and to interact with other agents. As a part of the FIPA specifications, FIPA (2004) describes a minimum set of services for an agent platform. On the other hand, the standardized IETF Constrained RESTful Environments (CoRE) framework provides interoperability in IoT (Shelby 2010) based on the REST principles. The CoRE framework aims to integrate resource-constrained things that have the roles of clients and servers for each other simultaneously. To enable mobile agent operations on resource-constrained things, this thesis presents a mapping of the mobile agent architecture into an optimized Web protocol, Constrained Application Protocol (CoAP) that is a part of the CoRE framework. Lastly, an application programming interface
(API) based on the resource-oriented architecture (ROA) guidelines (Richardson & Ruby 2007) is proposed that seamlessly integrates these components into the framework.

Energy efficiency in distributed IoT application execution is a crucial concern due to the resource constraints of devices. Once the mobile agent software framework prototype is implemented, two mobile agent based real-world IoT applications are evaluated atop the framework. Mobile crowdsensing (Ganti et al. 2011) and edge computing (Shi & Dustdar 2016) as data-intensive IoT applications are considered. These application scenarios manifest opportunistic utilization of both constrained stationary and mobile IoT devices. For evaluation, these applications are designed and implemented with mobile agents that address energy efficiency by in-network data processing, controlling the device resource use and by optimizing information sharing in a MAS. To provide empirical evidence in these scenarios, energy consumption of mobile agent based application execution is compared to the energy consumption of straightforward data upload to the cloud.

Limitations apply to the scope of the thesis. The thesis focuses on a descriptive set of IoT applications and therefore a large body of research related to the mobile agent application possibilities in distributed systems and networking is not examined. The recent scientific literature about IoT networking, system architectures and applications is increasingly massive, reflecting the importance of IoT as the next generation platform for large-scale distributed applications. However, in this thesis, IoT middleware and networking issues in general, including wireless communications, ad hoc networks and P2P systems, are not examined in detail. IoT application design and programming techniques in general, e.g. cloud computing, mobile computing and opportunistic computing, are not considered except when appropriate in the mobile agent context.

The thesis does not critically examine the underlying Web technologies, i.e. the REST architectural style and ROA. The SOA and WS-* standards are too resource-consuming for resource-constrained things, as discussed in Section 2, thus they are not considered further. Moreover, Web service composition, orchestration and provision with mobile agents are excluded as specific application examples.

Software agents and MAS are actively studied as well. However, in this thesis, software agents in general, their architectures outside reactive agents, nor agent-oriented programming methodologies are not examined. MAS features, such as organization, coordination, communication languages and interaction patterns are not considered, either. Only the relevant FIPA specifications are considered outside the FIPA abstract architecture. Lastly, in-depth examination of mobile agent technology outside reactive
agents in the context of IoT is left for future work. Although mobile code security has been a major concern from the early days on (Carzaniga et al. 2007), security of mobile agents and IoT in general are omitted as large research fields of their own.

1.3 Research methodology

The research in this thesis is based on the constructive research approach. Theoretical studies lead to the development of constructs, i.e. a mobile agent architecture and a software framework, that are then verified through a set of simulations and prototype application implementations in real-world settings. The results are then analyzed to gain insights into the practical relevance of the presented solutions and the possibilities of developing the solutions further.

Publications I and II present the conceptual mobile agent architecture and software framework that are based on the REST principles and ROA guidelines. A real-world prototype of the framework is provided in Publication I. A REST-based API to enable utilization of mobile agents in application development is provided in Publications I and II. Publications III and IV present evaluation of the proposed solution with quantitative criteria, i.e. mobile agent-based application energy consumption is compared with an existing well-known solution. Publication III examines energy efficiency by large-scale simulations in mobile crowdsensing applications, that operate on an opportunistic set of mobile devices. The NetLogo (Wilensky 1999) MAS programming environment was used as the simulation platform. NetLogo particularly provides simulation environment for mobile agents that interact locally over short periods of time (Allan 2010), which corresponds to the characteristics of interactions of resource-constrained devices in the IoT. The simulations are based on scripts written with the Netlogo programming language. Simulation results were then confirmed in a small-scale real-world application. Publication IV examines, through simulations, the mobile agent based MAS for IoT edge computing applications with resource-constrained stationary embedded devices and mobile devices. Simulations are again conducted within the NetLogo programming environment. The simulation parameters are first gathered from real-world experiments and from the literature. The simulation data are analyzed with standard statistical methods.
1.4 Contributions of the thesis

This compilation thesis consists of four scientific articles, where the author is the main author. The relationship between the articles is presented in Figure 4. The contributions of the articles are summarized below and elaborated in Chapter 3.

Publication I presents the resource-oriented mobile agent architecture and the ROA-based software framework as the main contribution. The proposed architecture is designed to be hardware platform-, operating system- and programming language-independent. Mapping of the mobile agent architecture to CoAP message format is described. The article shows how interoperability with mobile agents and system components can be realized with a RESTful API that is implemented with HTTP and CoAP. The author is responsible for the concept and design of the mobile agent architecture, the software framework and agent representation as a CoAP message. The author implemented a CoAP library and a mobile agent platform for the WSN nodes, atop the basic node firmware that was developed by the co-author Mr. Pauli Närhi. The author extended the process migration software for Android smartphones, based on the work by Mrs. Archana Ramalingam in (Ramalingam 2013) under co-supervision by the author, towards a mobile agent platform. The author implemented the framework proxy component software. Dr. Meirong Liu implemented the distributed resource directory (DRD) utilized in the framework. The presented communication latency evaluation results were not used in the thesis, as the node transceiver modules and infrastructure components were prototypes with a number of issues. The manufacturer later stopped the product development.

Publication II extends the framework and architecture presented in Publication I towards the design of mobile agent-based smart objects in IoT. The main contributions of the book chapter are a smart object reference architecture and an API to realize IoT smart objects with mobile agents. Mobile agents can be used to implement and distribute smart object functionality in the IoT system components. The book chapter did not contain any evaluation of the proposed approach. The author is responsible for the conceptual idea of IoT smart objects based on mobile agents, designed the reference architecture, and defined the API.

Publication III presents, as the main contribution, how mobile agents in a MAS can be utilized to realize mobile crowdsensing applications. Mobile agent based crowdsensing campaigns operate within the software framework and API described in Publications I and II. The article also discusses additional use cases for mobile agents in
mobile crowdsensing. Large-scale simulations were conducted to demonstrate energy efficiency in comparison with existing mobile crowdsensing approaches. The results indicate that mobile agent based crowdsensing campaigns can operate with less energy, depending on how the campaigns are designed as a MAS. The results also indicate insignificant communication overhead for agent-based real-time information sharing in a MAS. The evaluation of a small-scale real-world mobile crowdsensing application provided data that verified the simulation results. The author is responsible for the main idea of the article. The author designed the simulation scenarios, conducted the simulations and analyzed the results. The real-world evaluation application was implemented jointly by the author, for the mobile agent Android operating system (OS) platform, and by Mr. José Álvarez Lacasia who contributed the mobile agent task code for in-network data processing. The author collected and analyzed the evaluation data.

**Publication IV** presents as the main contribution a solution for using mobile agents to extend IoT edge computing platforms with the data producing things as computational platforms. Mobile agents as a MAS are used to distribute data processing tasks into an opportunistic set of heterogeneous resource-constrained IoT devices and to autonomously control the device operation. The results show that in-network data processing and agent-based control of devices reduce significantly communication energy consumption. The results were obtained through large-scale simulations that rely
on real-world experimental parameters and on the results obtained in Publication III. The author is responsible for the idea of the article. The author conducted the experiments, with software components developed by Mr. Joonas Kataja under the supervision of the author, and collected the set of real-world parameters for the simulations. The author designed and conducted the simulations and analyzed the results.

1.5 Research history

The author started the work towards this thesis by integration of process migration and Web technologies with embedded systems. This idea was first applied in a real-world WSN application, UBI-AMI v2 for home energy consumption monitoring, based on our previous work in Ojala et al. (2011). This work was done in the RealUBI project and with a personal grant from Pohjoista voimaa - Ympäristötili during the years 2011 and 2012. Particularly, Mr. Pauli Närhi and Mr. Jani Ylioja designed and implemented two different WSN nodes for the project. In summer 2011, the author participated in the 2nd UBI Summer School in Oulu, Finland, where the IETF CoRE framework was introduced. This event had significant effect on this thesis as the framework was then adopted for the UBI-AMI v2 application (Leppänen et al. 2012). However, this WSN was not used in further evaluation in this thesis, but as a software development platform.

During the MAMMotH project from 2012 to early 2014, the mobile agent architecture, the software framework and its components, presented in Publications I and II, were developed by the author and co-workers in the project. In Publication I, the WSN node hardware and firmware developed by Mr. Närhi, was extended by the author into a mobile agent platform. Dr. Meirong Liu implemented the distributed resource directory (DRD) utilized in the framework (Liu et al. 2013a,b), where the author was a co-author. The author and Dr. Erkki Harjula acted as technical supervisors for a Master’s thesis (Ramalingam 2013) in which a process migration software for Android smartphones was developed. Atop this software, the author implemented an Android mobile agent platform that was used in the evaluations conducted in Publications III and IV.

The author visited Sezaki laboratory, Institute of industrial Science, the University of Tokyo, Japan, from October 2012 to September 2013. During this visit, the author implemented a collaborative sensing demo application atop the mobile agent framework (Leppänen et al. 2013b), described a lightweight agent-based IoT architecture with CoAP as the communication protocol (Leppänen & Riekki 2013), defined a distributed architecture for participatory sensing (Leppänen et al. 2013a) and implemented software
components for the mobile agent platform for Android smartphones. The visit resulted in Publication III that presents the evaluation of mobile agent based mobile crowdsensing applications.

The work towards the Publication IV was conducted by the author during the year 2016. The author utilized a mobile agent platform implemented for Arduino embedded boards, adopted from the work in Publication I that was used to collect real-world data for the simulations. This platform was developed under the supervision of the author by Mr. Joonas Kataja with the Digile IoT research program.

1.6 Structure of the thesis

This compilation thesis is structured as follows. This Chapter introduced the focus of the thesis as defined in the research questions and described the author’s contributions based on the research articles. Chapter 2 presents the theoretical concepts that this thesis builds on and the state of the art concerning mobile agents in IoT. The research contributions of the presented work are presented in detail in Chapter 3. Chapter 4 revises the research questions, examines the results and outlines open issues for future work. Lastly, Chapter 5 concludes the research work and results presented for the thesis. The original scientific articles presented for the thesis are attached at the end.
2 Theory and concepts

This Chapter presents the key concepts and technologies in the domains of software agents, Internet of Things (IoT) and resource-oriented architectures (ROA). Mobile agents are based on the concept of software agent and are expected to interact in a multi-agent system (MAS). IoT provides the distributed systems context, where mobile agent operate in the ROA-based software framework.

2.1 Software agents

Russell & Norvig (2003, p.32) define an agent as “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators”. Another definition is given in Nwana (1996): “a component of software and/or hardware which is capable of acting exactly in order to accomplish tasks on behalf of its user”. Agents are categorized into human, hardware and software agents. An example of a human agent is a travel agent, who handles travel arrangements on behalf of a customer. A hardware agent has an embodiment, e.g. a robot operating in a factory. The general benefit of hardware and software agents is that they perform for example difficult, laborous and dangerous tasks on behalf of their owners.

Software agents differ from traditional software programs in relation to autonomy and intelligence (Franklin & Graesser 1997). A program must react to given inputs, execute once when started and wait to be started again. Autonomous agents can decide whether to react to an input or perform an action. An agent can be proactive, i.e. take an initiative. All software agents are programs, but programs need to embrace these capabilities to be software agents (Franklin & Graesser 1997). As a software construct, a software agent is similar to an object in object-oriented programming (Weiss 2013, Chapter 1). An object has a state, performs actions through its methods and communicates by message-passing. An object is not able to control its own behavior, but others invoke public methods of an object.

Weiss (2013, p. 4) gives a definition for a software agent: "a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to achieve its delegated objectives". Wooldridge & Jennings (1995) discuss weak and strong notions for agents and corresponding properties. The weak notion is presented first and the strong notion later in this Section. A weak agent is
autonomous and has control over its state, i.e. operates without direct intervention of human or other software components. A reactive agent reacts to changes in its environment, whereas a proactive agent takes initiative to exhibit goal-directed behavior. A rational agent selects actions that maximize its performance towards its goals, based on its knowledge, and will not act in a way that prevents its goals being achieved (Russell & Norvig 2003, Chapter 2). A stationary agent is bound to a specific computer and interacts through message-passing or by modifying its environment. Mobility allows an agent to autonomously change its logical, i.e. application-specific, or physical location in a networked system (Lange & Oshima 1999). Agents are social, communicate via an agent communication language (ACL) and interact with other agents to share information, cooperate, collaborate, negotiate and compete. Agents should be veracious and benevolent in their operation. An intelligent agent has three capabilities: reactivity, sociality and proactivity (Weiss 2013, Chapter 1). Agents possess different degrees of intelligence and learning capability (Russell & Norvig 2003, Chapter 2). A simple reflex agent acts on its current percepts without any other knowledge. A model-based reflex agent has its own state and internal model of the partially observable world. A goal-based agent plans its actions towards achieving its goals, i.e. desirable situations. A utility-based agent has an utility function, that support rational selection of desirable actions. Learning capability means that an agent acquires knowledge based on feedback of its operations and has a problem generator that enables to explore new actions.

A software agent architecture contains a decision-making system (Russell & Norvig 2003, Chapter 2), as shown in Figure 5a. The agent function, that is realized by an agent program, maps percepts to actions. The agent program runs in a computer that provides its sensors and actuators for the program. Maes (1991) gives another definition for agent architecture: "It specifies how the overall problem can be decomposed into subproblems, i.e. how the construction of the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions and future internal state of the agent."

Weiss (2013) introduces four classes of software agents. The first class is logic-based agents that use deduction in decision-making to select actions based on a symbolic representation of the environment. The second class is reactive agents that perceive their environment and respond in timely manner to the perceptions. Reactive agents have task-specific competence modules that produce actions for the percepts, as shown in
Figure 5b. These agents are able to react to state changes in their environment, but without reasoning nor planning capabilities, thus they can’t solve a task for which no competence module exists. Reactive agents communicate either with direct messages between agents, where no complex language can be used, or indirectly by changing the environment in a way that can be observed by other agents. Therefore, intelligent behavior of reactive agents is a product of their interactions with the environment. A well-known example of reactive agent architecture is subsumption architecture (Brooks 1986). The behaviors of these agents are defined as rules that map inputs to actions without reasoning. The behaviors are arranged as layers. Lower layers have higher priority and higher layers represent increasingly abstract behaviors. This architecture has only indirect means for agents to communicate by leaving trails of themselves, e.g. messages, into the environment. The other agents at the same environment can observe these trails. Another well-known architecture is presented by Maes (1991). Competence modules compete with each other and the most relevant module executes its action. Modules are internally linked as a dynamic network. The structure of the network changes as a result of gained experience. New modules can be introduced by learning.

The third class, and the stronger notion of agents (Wooldridge & Jennings 1995), is deliberative agents that utilize human-like mentalistic concepts, e.g. knowledge, belief, intention, obligation and emotion, and have related properties and capabilities. These agents are able to decide what goals they want to achieve and how to achieve the goals through a symbolic model of their environment. A well-known example is Belief-Desire-Intention (BDI) agents that operate based on a set of mentalistic notions
Beliefs contain an agent’s views of its environment, desires are the expectations of future states of the environment and intentions are selected actions. A goal is defined as a subset of desires that the agent could act on. Agent, in its decision-making process, reasons about its desires and selects a set of possible intentions. These intentions are combined into plans that are executed leading towards the goals. The observed results of the actions then influence the agent’s beliefs.

The fourth class is agents based on layered or hybrid architectures (Russell & Norvig 2003, Chapter 4) that combine reactive and proactive behaviors through software layers. Two approaches for layered operation have been introduced. With horizontal layering, all layers are connected to the same inputs, but a mediator function gives control of the agent operation to selected layers. With vertical layering, inputs flow upwards from the bottom layer and are dealt by each layer at a time. Outputs then flow downwards to the actuators. Minimally two layers are introduced, the lower layer is reactive and the upper layer proactive. According to Weiss (2013, p.41), layered architecture is currently the most popular general class of agents.

A software agent is situated in an environment. The agent designer decides how the environment is modeled to the agent. Different applications and tasks may result a different model of the same environment. The agent observes the state of the environment with its sensors and produces actions with its actuators that affect the state (Weiss 2013, Chapter 1). Agent environments are characterized by different dimensions from the agent point-of-view (Russell & Norvig 2003, Chapter 2). In a fully observable environment, an agent has access to complete application-specific state of the environment, whereas in a partially observable environment parts of the state cannot be accessed. In a deterministic environment, the next state is completely determined by the current state and the action of the agent. A partially observable environment appears to be stochastic. In an episodic environment, an agent’s actions are independent of each other, i.e. an action does not depend on previous actions. In a sequential environment, the current action affects future actions, e.g. turns in a game. In a static environment, only the agent actions modify the state of the environment, whereas in a dynamic environment other system components also modify the state of the environment. A discrete environment has a finite number of states, percepts and actions. Open environments have no limitations of novel circumstances that may arise. An environment is a multi-agent environment when the agents treat other components as agents.

Brenner et al. (1998, Chapter 4) discuss the differences between reactive and deliberative agents in dynamic and open environments. A challenge with deliberative
agents is that their symbolic model of the world is created at a specific time and is difficult to update. These agents aim at the optimum result that is complex to achieve, but in dynamic environments a fast reaction with satisfactory quality could be a sufficient result. As IoT is an example of a dynamic environment, reactive agents are of particular interest in this thesis and deliberative agents are not considered further. Reactive agents do not possess an internal model of the environment and therefore are architecturally simpler and more flexible in their interactions with the environment. Moreover, a reactive architecture model increases robustness as the competence modules can operate independently of each other.

\section{2.1.1 Multi-agent systems}

An individual agent has restrictions, such as the size of its knowledge base, responsiveness, flexibility and modularity (Nwana 1996, Sycara 1998). When a problem exceeds the capabilities of an individual agent, a number of agents can be integrated into a MAS that provides a framework for the agents to communicate and cooperate in solving problems (Brenner \textit{et al.} 1998, Weiss 2013). A MAS is characterized by Sycara (1998) as a system where each agent has incomplete information or capabilities to solve the problem at hand, has no global control, has decentralized data and where tasks are executed asynchronously. Each individual agent has its own objectives and knowledge, thus coordination is required to combine these contributions to solve the overall problem. A MAS is generally considered robust as a failure of a single agent does not endanger the whole system operation. Agents in a MAS are capable of learning and planning cooperatively, but these cognitive capabilities are omitted in this thesis.

A problem can be presented for agents in several ways (Shehory & Sturm 2014, Chapter 4). Agents may know the problem beforehand, i.e. top-down approach, whereas emergent problems may be appear also in bottom up way. A middleware or an agent can be introduced to divide a problem into subproblems and to find a suitable agent for each subproblem. In an extreme case, solving the subproblems does not require agent interactions at all. When this is not the case, a cooperation protocol for agents is needed (Brenner \textit{et al.} 1998, Weiss 2013). Such a protocol provides structure for agent negotiation, information sharing and conflict resolution.

MAS openness is defined as the ability of agents to join and leave the MAS (Shehory & Sturm 2014, Chapter 4). In an open system, heterogeneous autonomous agents run independently without considering how the system is organized (Weiss 2013, Chapter
2). An example of such a system is the Internet (Sycara 1998). Dynamic openness facilitates agents to join and leave in runtime, where an agent registry is needed for discovering agents. Static openness requires agents to inform others of their presence and capabilities, where no agent registry is used. This kind of MAS is suitable for predictable environments, but insufficient for environments with high level of uncertainty. Offline openness means that the system is restarted whenever agents join or leave, which is not feasible in dynamic environments.

MAS organization describes how agents form as MAS (Weiss 2013, Shehory & Sturm 2014). An organization has its own rules, objectives and authority. Agents are given roles that define their relationships and interactions. A structural organization provides the means for coordinated effort towards MAS objectives, whereas an institutional organization regulates the interactions of agents (Weiss 2013, Chapter 2). MAS organizational types are discussed further in (Shehory & Sturm 2014, Chapter 4). A hierarchical organization emphasizes centralized control, where an agent interacts only with agents in higher or lower levels of the hierarchy. This reduces complexity of agent interactions, but has less dynamicity. A flat organization has no fixed structure, hence agents can interact directly with each other, and dynamically form and adjust organizational structures. In a flat organization, an agent registry is needed and agent communication overheads can increase. A subsumption organization consists of agents that are components of larger container agents, which control the MAS operation. A subsumption MAS is difficult to modify due to the enforced structure. A modular MAS is comprised of modules that can be a stand-alone MAS. Modularity increases flexibility and reduces overall communication overhead.

Different agent communication models have been proposed for distributed systems (Genesereth & Ketchpel 1994, Brenner et al. 1998, Chapter 4). First, in the C/S communication model, agents are considered as concurrent clients and servers for each other. Second, in the RPC model agents invoke procedures on the other parties. Third, the blackboard-based communication model provides a common work area that the agents use to indirectly exchange messages. For cooperation, the blackboard is used to disseminate subproblems and share the results. The benefit of a blackboard is that an agent can access it any time from anywhere, but is based on centralized architecture. Fourth, message-passing is a direct communication model between agents, but also multicast and broadcast messages are possible.

Coordination in a MAS is achieved by means of agent communication with interaction protocols (Weiss 2013, Chapter 3). Agents communicate to inform other
agents about their actions and to exchange knowledge. Communications are typically symmetric, where both parties are in control of the communication (Shehory & Sturm 2014, Chapter 4), but have no control over the autonomous actions of others (Weiss 2013, Chapter 3). With indirect communication, agents release messages into the environment, e.g. trails of themselves, that can be observed by others. With direct communications, agents interact with each other, where cognitive capabilities are needed to maintain the state of the other party. The interaction protocol messages consist of ACL expressions, e.g. communicative acts, based on a common vocabulary that is described in a common ontology (Genesereth & Ketchpel 1994, Weiss 2013). This way, the agents can have meaningful dialogues and build a mental models of each other during the interactions. The languages and protocols are discussed further, e.g. in Genesereth & Ketchpel (1994), Brenner et al. (1998), Weiss (2013), Shehory & Sturm (2014).

To realize interoperability between agents, MAS requires a set of services for agent naming and discovery, communications, coordination, mobility, lifecycle management, security, system resource utilization and system monitoring (Milojicic et al. 1998, Bellifemine et al. 2001, Cucurull et al. 2009). Agent standards targeting these MAS services have been proposed, such as FIPA specifications (FIPA 2002a) and Mobile Agent System Interoperability Facility (Milojicic et al. 1998). FIPA provides standardized abstract architecture for MAS (FIPA 2002a). The FIPA specifications facilitate interoperability between agents and define agent management services (FIPA 2004) and messaging services between agents (FIPA 2002d). The specifications do not address the internal architectures of agents, but a set of mandatory and optional agent attributes are defined, i.e. an agent name and a locator that is the agent’s communication interface specification. The common FIPA ACL (FIPA 2002b) defines communicative acts to facilitate complex interactions between agents that have the cognitive capabilities of deliberative agents. FIPA ACL also has a defined set of message transport protocols, such as HTTP (FIPA 2002c). HTTP is utilized solely as message transport protocol, where the method must be POST (FIPA 2002c). The HTTP request and response content type is required to be Multipurpose Internet Mail Extensions (MIME) multipart. The request body consists of two parts: a message envelope describing transport information and a payload consisting of the ACL message.

The FIPA abstract MAS architecture is shown in Figure 6a. Two discovery services are provided, which may have several instances: service directory that stores the system service descriptions and an agent directory service (ADS) that stores agent descriptions. An agent directory entry has the name of the agent, a locator and optional attributes
that are described in detail in (FIPA 2004). A service entry requires similarly the name of the service, a type and a locator that contains its address and communication interface description. The directories provide functionality to register, modify, unregister and search for entries. A third optional service is the Agent Discovery Service (FIPA 2003) that is a realization of the ADS for ad hoc networks. This service is actually a wrapper for external non-FIPA discovery services for agents in ad hoc networks. This service operates in parallel with ADS realization that handles specifically local agents and services. Again, similar functionality to register, modify registration and unregister is provided for the agents. Message transport service (MTS) supports sending and receiving messages between agents by using transports that are message delivery services. A transport is for example the underlying OS communication interface. FIPA provides a set of standardized envelopes to encapsulate the messages into common formats, such as the Extensible Markup Language (XML).

The FIPA agent platform, depicted in Figure 6b, provides services for the agent. Directory facilitator (DF) is a realization of the ADS in the platform. MTS is defined for agent communications between platforms. Agent management system (AMS) maintains a list of all agents in the platform and controls their lifecycle. AMS can create, invoke, suspend, resume and destroy agents. Agents can interoperate within the AMS and can put themselves into wait state, but only the AMS can wake up an agent in the wait state. Mobile agents are moved and executed by the AMS. Agents in mobile devices
are considered in (FIPA 2002e) that introduces two additional stationary agents in the platform to handle device mobility issues. The first agent monitors the AMS message transport connection performance and the second agent manages network connections based on the measured performance.

A well-known FIPA-compliant agent platform is Java Agent Development Framework (JADE) (Bellifemine et al. 2001, 2007). JADE agents are BDI agents, defined as a set of Java classes that encapsulate agent properties, different agent behaviors, interactions, state and data. Agent interactions are conducted with FIPA ACL interaction protocols, but also other protocols are possible. For agent mobility, JADE provides supporting classes to describe host computers and Java language mechanisms in general can be used to transport Java objects. For integration over the Internet, JADE provides Web service gateway add-on that enables to invoke agent services by Web clients.

### 2.1.2 Mobile agents

Contrary to a stationary software agent, a mobile agent is a software agent that is not bound to the origin computer where its execution begins. A stationary agent communicates by message-passing or by invoking remote procedures in servers, whereas a mobile agent contains the requested procedure as the agent program. A mobile agent is injected into the system by a client. Then, the agent autonomously controls its operation in the server and migrates, by transmitting itself, between servers where the agent program is executed (Cao & Das 2012, Chapter 1). The conceptual origin of mobile agent is in process migration according to Wong et al. (1999). The Emerald mobile object programming language (Jul et al. 1988) includes a primitive “go” that enables an executing process to migrate while carrying its code, data and execution state. The capability of autonomous migration sets mobile agents apart from the other code mobility paradigms. A software agent can be physically moved with its host device, but such mobility is not autonomous for the agent. Early well-known examples of mobile agent technology include Johansen et al. (1995), Tardo & Valente (1996), Gray (1997), Peine & Stolpmann (1997), Wong et al. (1997), Baumann et al. (1998), Glass (1998), Karnik & Tripathi (1998a), Lange & Oshima (1998), Silva et al. (1998), Baumer et al. (1999), Suri et al. (2000), Gray et al. (2002).

The benefits of mobile agents in distributed systems are widely known (Chess et al. 1997, Brenner et al. 1998, Lange & Oshima 1999, Cao & Das 2012). Mobile agents operate asynchronously and autonomously with regard to the origin client. This
eliminates the need for the client to maintain network connections during the agent operation. Migration distributes the agent program execution load to the participating servers, whose functionality can be augmented or updated with the program, e.g. to introduce a new communication protocol. Moreover, by processing data locally at remote servers, the computation load of the client and communication load of the both parties are reduced as only refined data is transmitted over the network. In real-time systems, operational latencies can be reduced when mobile agents act locally at the data source. As intelligent agents, mobile agents can autonomously adapt to the situation by relocating themselves in response to changes in the environment. This capability is not possible for stationary agents. With recursion, mobile agents are able to create child agents or clone themselves to distribute task execution further. These properties increase overall robustness and fault tolerance of the system.

Figure 7 shows generic mobile agent architecture and the components of a mobile agent platform. The architecture follows the code mobility executing unit architecture, as defined in Fuggetta et al. (1998) with an agent identifier and an itinerary. Due to mobility, a mobile agent needs a unique identifier that is not bound to the current host name or address. The code section corresponds to the agent program. Data space contains data and references to the resources used by the program. These resources are abstracted as the sensors and actuators in the software agent architecture shown in Figure 5a. State is the execution state of the program, including its instance variables. Itinerary defines the migration route the agent is expected to follow.

A mobile agent platform requires additional methods, with regard to the platform architecture shown in Figure 6b, to handle agent migration and agent program execution.

In a platform, the lifecycle of a mobile agent starts from creation or cloning. A naming service is needed that provides a unique identifier for the new agents. The EE handles the agent program execution atop platform-specific OS, or a VM, that in turn handles resource management for the agent and the platform. The agent programs are executed as subroutines, threads, processes, objects or as complete programs that rely on the underlying OS or VM services. In multi-tasking platforms, multi-threaded execution is feasible as the agent program typically runs in its own thread and it may launch other threads when needed, for example to communicate. Agent programs are also executed as an independent service or integrated as components into an existing server functionality, e.g. into a Web server (Lingnau et al. 1995, Funfrocken 1997). A migration service is needed to transfer the agent, including an agent state capture service. The state capture can be automatic or controlled by the agent program. Security services, such as authentication and access control, are provided as part of the underlying OS or VM. Agent storage service provides persistent agent operation in the platform, e.g. activation and deactivation of agents without destroying them. Finally, a platform may expose a user interface to monitor and manage its agents.

The platform functionality may be distributed among computers in the network to reduce the hardware requirements of a single computer (Milojicic et al. 1998, Bellavista et al. 2001, Bellifemine et al. 2001, Bergenti & Poggi 2001, FIPA 2004, Muldoon et al. 2007, Wu et al. 2007, Fortino et al. 2008). Services that can be distributed include naming, addressing, discovery, location transparent communication, migration and agent program repository. These services can be implemented as remote agent platform services, system services or integrated into the middleware.

An agent API in the platform provides an interface for the agent to access platform data, invoke services on the platform and to interact through platform input/output (I/O) interface and communication protocols. The interface facilitates platform-specific
direct and indirect agent communication mechanisms, e.g. C/S, RPC, blackboard, publish-subscribe, events and messages. A multi-tasking platform needs to provide an interface for inter-agent communications in the platform, preferably through the same communication interface. Multi-tasking platform should also provide means for non-agent applications to utilize agent services, through the OS services, where resource management is needed.

Mobile agent migration process generally requires five steps (Milojicic et al. 1998, Milanés et al. 2008, Cucurull et al. 2009): request, suspend, transmit, receive and resume. First, an agent requests migration through a program primitive, such as ‘move’, or by calling an agent API method (Brenner et al. 1998, Chapter 4). Second, the platform suspends the agent operation, i.e. its program and possible on-going interactions. The set of agent components that are transferred are selected. Generally, the agent program, selected set of data and execution state are included, but different methods are used. Third, agent components are serialized into platform independent representation for transport. This step is straightforward when the platform has object serialization methods, as in Java. Then, the serialized agent objects are transmitted as a message, but also platform-specific migration protocol can be used (Lange & Oshima 1998). Fourth, the new platform performs authentication, deserializes the message into agent objects and builds the agent. Fifth, the agent operation is resumed by the EE. Different migration procedures, methods and protocols have been extensively studied in the literature, but optimizing the migration procedure is out of the focus of this thesis.

Different methods can be used to transfer an agent. Migration is transparent when the EE fully handles the migration and the agent is not aware of the process after invoking it. The other approach is to involve the agent in the migration process, where the agent is required to have the methods to transform its needed components. The latter approach reduces the functionality of the EE, but is burdensome for the agent programmer. Agent program code can be pushed, e.g. scripts, but the new host can also pull the program code or selected parts from the old host or from a system code repository. To reduce the transmitted agent size, only the program references instead of code objects can be included. For example, in Java, dynamic class loading can be used to retrieve the required agent program through the class references. With resource-constrained platforms, the code objects are required to exist in the platform to make migration possible (Muldoon et al. 2006, Aiello et al. 2011).

The migration destinations are categorized into homogeneous, regenerative and heterogeneous destinations (Brazier et al. 2002, Milanés et al. 2008). A homogeneous
destination requires no changes for the agent as it is executed in the same type of EE across platforms. A regenerative destination requires rebuild of the agent with possibly a set of different components. A heterogeneous destination requires rebuilding the agent to an executable program. Fuggetta et al. (1998) discuss weak and strong migration. In weak migration, only the agent program and data, or their references, are transmitted. In strong migration, the program execution state in a processor is captured and transmitted along. Not all EE’s support strong mobility as low-level access to the OS or VM can be challenging to realize, e.g. with Java as discussed later. The data space can be handled in different ways, where the issue is rebinding to the required resources after migration (Fuggetta et al. 1998). Binding by identifier means that the agent is always bound to a particular resource, e.g. a set of historical data, which can be found with its identifier. Binding by value means that a resource must be of a given type and its value cannot change as a consequence of migration. In this case, the resource data is transferred within the agent. Binding by type means the agent is bound to a resource of a given type, such as a sensor.

Mobile agent communications face the fundamental question whether to migrate into a remote server or communicate with it (Cao & Das 2012, Chapter 2). Both approaches are feasible as a mobile agent may also need to interact with system components that are not part of an agent platform. Challenges in mobile agent communications include asynchrony, timeliness and reliability of message delivery, location transparency and scalability of the messaging method (Rawat et al. 2015). From the agent perspective, the best option is a location transparent primitive, e.g. “deliver” or an API method, where the platform takes care of reliable message delivery. Different solutions have been proposed for tracking the location of mobile agents (Cao & Das 2012, Rawat et al. 2015). First, session-oriented communication protocols establish sessions to channel messages between participants. Second, a central server can keep track of the locations of all mobile agents and deliver messages, but update and message delivery overhead can be large. Distributed servers can form a tree-like structure to deliver message to mobile agents in leaves. Blackboards provide centralized distributed storage for the agent messages. Third, a mobile agent location server, a home server or a proxy maintains location database of its agents. When the server receives message on behalf of the agents, it forwards the messages to the agents. Fourth, mobile agents can leave forwarding pointers or proxies to the hosts each time they migrate. A message is routed along this track to reach the agent. (Cao & Das 2012, Chapter 2) describe migrating mailboxes that follow the agents when requested and agents contact their
mailboxes infrequently. Fifth, broadcasting an agent’s location is a suitable option for
disconnected operations but introduces communication overhead. In general, these
methods do not guarantee message delivery (Cao & Das 2012, Chapter 2) where the
challenge is synchronization between delivery and migration and tracing the agent. A
survey of mobile agent communication protocols is presented by Rawat et al. (2015).

Mobile agent coordination, due to mobility, operates in both spatial and temporal
dimensions (Cao & Das 2012, Chapter 4). Spatial coupling means that the agents share
a common name space, e.g. a blackboard. Temporal coupling requires synchronization
between agents. When agents communicate with each other, they are both spatially and
temporally coupled. For meeting-based coordination, agents join into meetings and
interact only locally within the meeting. Blackboard-based communication is spatially
coupled but temporally uncoupled. If the agents use a local blackboard or tuple space in
the platform, then spatial and temporal uncoupling is achieved. However, with local
blackboard, the platform needs to synchronize the contents with other platforms.

Mobile agent platforms have been integrated with Web technologies, e.g. Lingnau
et al. (1995), Funfroken (1997), Cabri et al. (2000b), Foukarakis et al. (2003), Cao
et al. (2004), Spyrou et al. (2004), Bellavista et al. (2005), Peters (2005), Adacal &
Hachicha (2015), Voutilainen et al. (2016). Generally, Web servers as agent platforms
either execute agents in an external service or the agents are used to implement parts of
server functionality. Mobile agents in these platforms not only retrieve and process Web
service content or provide a service, but also have been extensively used for Web service
composition, orchestration and provisioning. However, in this thesis these mobile agent
application examples are excluded. In these platforms, typically HTTP has a role as
the agent message transport protocol, where the ACL messages are encapsulated for
example into WS-* standard envelopes. The agent APIs may be HTTP- or REST-based,
e.g. Braubach & Pokahr (2013), Jamont et al. (2014), Bennajeh & Hachicha (2015),
where the HTTP methods are used for agent interactions as follows. GET method is
used to retrieve an agent status, possibly as a part of Web service content. HTTP is also
used as the agent transport protocol, e.g. POST and PUT methods are used to push the
agent to the new host, whereas GET method pulls the agent into a new host for example
in a HyperText Markup Language (HTML) document.

Related to standardization, the Object Management Group has proposed a standard
for mobile agent systems (Milojicic et al. 1998). The standard covers agent semantics
and naming, migration, discovery and security, but agent communications are not
considered. The standard relies on more than a decade old version of Common Object Request Broker Architecture (CORBA) and standardization work has not been active recently, according to Cucurull et al. (2009). Similar functionality can be provided with the FIPA specifications.

2.1.3 Programming mobile agents

In this Section, agent platform implementations are reviewed with regard to the language used for the agent program. The focus is on embedded systems and mobile phones as representatives of resource-constrained IoT devices.

Interpreted languages are beneficial with regard to portability between host computers running different OS on different hardware. Platforms have been presented for example by Tardo & Valente (1996), Peine & Stolpmann (1997), Gray (1997), Gray et al. (2002), Chen et al. (2008), González-Valenzuela et al. (2010), Muhametsin (2011), Mitrović et al. (2014), Voutilainen et al. (2016). Agent program scripts are executed in a scripting environment, which may require a preceding step of creating an executable code block with the agent data and state. Scripts are easily transferred as a part of the agent message, but state capture can be challenging as it requires low-level support from the scripting environment that may not be available. Typically, the captured state is stored in a set of instance variables in the script that provides easy restart, but it can be difficult to separate program, state and instance variables from the script. Issues with interpreted languages include limited expressive power, overhead caused by script execution in an external environment and possibly limited features to control script execution. These agent platforms may require pre-installation of additional application-specific code libraries.

Script-based mobile agent platforms have been presented. For embedded systems, well-known TinyOS (Levis et al. 2005) in Szumel et al. (2005), Fok et al. (2009), González-Valenzuela et al. (2010) provides a TinyScript language for agent programs. The programs are then executed in the TinyOS VM. TinyOS does not natively support mobile agents, therefore, agent extensions are introduced to the language that enable autonomous migration and agent interactions. TinyOS does not support strong migration. Another examples are mobile agent programs presented in JavaScript as a part of HTML documents, e.g. Mitrović et al. (2014), Voutilainen et al. (2016). These platforms are natively integrated into the Web. Another benefit is that the agents can contain a user interface that is executed in the Web server.
Java has been the de facto mobile agent programming language from the early days (Wong et al. 1999). Java is portable and platform-independent with neutral bytecode that can be executed in a large number of heterogeneous platforms. Java has dynamic class loading and Remote Method Invocation that provides a transparent way to access remote objects through a local proxy. Java realizes a component-based agent architecture where Java classes encapsulate the agent data and behavior. Naturally, the platforms provide Java-based agent API. The programming model is commonly event-based, where an agent operates in response to lifecycle or interaction events, e.g. Bellifemine et al. (2007), Aiello et al. (2011). Java agents are executed in their own thread that launch other threads as needed. With regard to platform services, a Java Virtual Machine (JVM) needs to provide common interface to hardware-dependent features (Maye 2013), such as physical components and communication interfaces. For agent migration, Java natively supports object serialization. However, standard JVM does not provide methods for Java object state capture, thus migration is weak. Some modified JVMs provide state capture for strong migration, e.g. Suri et al. (2000), Aiello et al. (2011), Lopes et al. (2011). In the case of multithreaded agent migration, typically only the main object thread is transferred and other threads are discarded. A Java agent platform may also have its own agent transfer protocol atop HTTP as in Karnik & Tripathi (1998a), Lange & Oshima (1998). Moreover, a distributed Java software framework may have specific methods to transmit Java objects as in Wu et al. (2007), O’Hare et al. (2012), Lee et al. (2013). Examples are CORBA and more recently Open Services Gateway Initiative that provide standardized services for resource discovery and remote object access. Java is also widely utilized with Web technologies.

The standard Java solutions and JVM may be too resource-consuming for embedded systems, thus reduced JVM functionality agent platforms have been presented (Muldoon et al. 2008, Aiello et al. 2010, 2011, Lopes et al. 2011). These JVMs utilize modified Java bytecode and have restricted Java functionality, e.g. for dynamic memory allocation, garbage collection, method invocation and dynamic class loading may not be available. For example, required Java classes might need to be pre-installed in the platform and Java objects communicate by message passing and not by invoking methods of objects. To address migration overhead and agent transfer size, minimally only an event is sent to the new host that then creates the agent corresponding to the event (Aiello et al. 2011). The event-driven programming model is event-driven facilitates resource sharing as agents are inactive until an event is triggered.
For mobile devices, Java platforms with standard or reduced JVM have been presented, e.g. Spyrou et al. (1999), Lawrence (2000), Tarkoma & Laukkanen (2002), Carabelea & Boissier (2003), Cao et al. (2004), Vasiu & Mahmoud (2004), Ilarri et al. (2006), Muldoon et al. (2007) and Su (2008). A well-known Java agent platform for mobile devices is JADE Lightweight Extensible Agent Platform (JADE-LEAP) (Bergenti & Poggi 2001). JADE-LEAP is based on the split agent functionality: agents with reduced functionality operate on the mobile device and the rest of the functionality, e.g. FIPA-compliant operation, is executed in components at the infrastructure-side servers. Communication is done with FIPA ACL messages that can be buffered to handle network disconnections. But mobile agents can also migrate between the split platforms to deliver requests. This model of split agent functionality has prevailed to this day. Split agent functionality with mutative mobile agents was presented in Muldoon et al. (2007), where the agent functionality is reduced before migration, based on the capabilities of the target device. Fortino et al. (2008) propose an agent architecture that is separated into platform neutral higher level behavioral part and platform-specific lower-level part. Recent Java agent platforms for mobile devices include Su (2008), Ughetti et al. (2008), Agüero et al. (2009), Santi et al. (2010) and Bergenti et al. (2014). Vyroubal & Kušek (2013) present Java mobile agents that can migrate between standard JVM and Android-specific VM platforms with additional bytecode conversion step.

For resource-constrained embedded systems, e.g. WSN nodes (Chen et al. 2007), reprogramming the device firmware with mobile agents faces tradeoffs (Maye 2013, Oliver et al. 2014, Silva et al. 2014). Macroprogramming with high-level language results portable and simpler agent programs, but requires more functionality and resources in the firmware and an additional script interpretation step is needed. The program size may be smaller than with the other solutions, but program execution takes longer, which reduces possibilities to save energy in the node by duty cycling. When the language abstraction level is kept low, flexible agent programs can be implemented that take advantage of the device’s low-level features, but the program size grows correspondingly. Considering migration, the challenge is to decide which parts of the program are migrated (Milanés et al. 2008): complete device firmware, an agent program, parts of the agent program or no program at all if the functionality already exists in the next host. However, WSN nodes are typically designed and deployed for a particular application, thus the device firmware may have strict limitations for tasking and resource pooling. Event-driven programming model is beneficial with respect to resource sharing. The node firmware or OS may have protected areas in the firmware.
that cannot be programmed, limiting the usability of agent programs. Typically, the devices require reboot after reprogramming that interrupts its task, if self-programming the device program memory is not facilitated.

Mobile agent platforms based on the C programming language have been presented as well. C is an efficient programming language with low-level features close to hardware and commonly used with embedded systems. C programs are typically precompiled into platform-specific language representation (Johansen et al. 1995, Peine & Stolpmann 1997). Bagci et al. (2009) present agent programs in C for the well-known Contiki OS (Dunkels et al. 2004). These programs are precompiled into the platform machine language format. In the platform, the precompiled agent program is written to the device program memory and data stored in Random Access Memory (RAM). In Chen et al. (2006a, 2008), Chou et al. (2010), C programs are run atop embedded C interpreters in the devices.

2.2 Internet of Things

Common IoT system architectures are based on layered organization of the underlying network components and three hierarchical layers of components (Borgia 2014, Want et al. 2015), as shown in Figure 8. The layers hide heterogeneity of technologies and network complexities from application developers. An application layer, typically realized as a cloud platform, is the top layer. A network layer contains middleware and network infrastructure components. As discussed later, this layer can have computational components for edge computing, hence it is also called edge layer. The lowest layer is the device layer for the data producing devices, i.e. the things that are typically low-power resource-constrained embedded devices and smartphones.

At the top layer, cloud computing paradigm provides virtual infrastructure of shared servers that provision applications with large-scale computational and data storage resources (Gubbi et al. 2013, Borgia 2014). Situational understanding of the system lies in the cloud and resources are dynamically, without human intervention, allocated for applications and for handling high volume of data traffic from lower layers. Applications connect to these on-demand data streams for information processing, analysis and visualization. This vertical model has led to application silos across the layers by different stakeholders (Borgia 2014). Applications are isolated and rely on proprietary infrastructure and dedicated data producing devices. Interoperability is only considered on the top layer that introduces redundancy to the lower layers as no resource sharing is
facilitated. This increases deployment costs. For cloud-based applications, Web services are used to provide user interfaces.

On the network layer, IoT is the interconnecting architecture across a wide range of technologies for physical and virtual things, services and applications (Atzori et al. 2010, Miorandi et al. 2012, Borgia 2014). Wired and wireless connections are facilitated though middleware and network infrastructure components that include base stations, routers, gateways, access points and proxies. In IoT, middleware is considered a layer of system software that integrates heterogeneous devices into the system (Razzaque et al. 2016). For the applications, middleware provides scalable data and resource management and discovery. Common abstractions and data formats are enforced in the middleware for information exchange and to extract higher-level meanings from the information. For the devices, middleware facilitates service access and system-wide interoperability that hides handling of different communication technologies and protocols. Mobility and location management, context-awareness and adaptivity are desired features of middleware in dynamic environments.

Recently, to augment the cloud-centric application model with resources on the lower layers, edge computing solutions have been presented, e.g. Satyanarayanan et al. (2009), Bonomi et al. (2012), Mach & Becvar (2017). Edge computing (Shi & Dustdar 2016) places application-specific cloud resources to the network layer edge devices into close proximity of the data producing end devices. An edge device is any resource-rich device between cloud and data sources at the network layer that participates in data dissemination, storing and processing (Shi & Dustdar 2016). Edge computing is beneficial for application execution in many ways. High bandwidth is provided for end devices to access edge resources at close proximity. Execution latencies
decrease as round-trip to the cloud is no longer needed. Edge resources can be tailored in respect of dynamic application and user requirements. Backbone network load is reduced as less data is transmitted upstream when parts of the application are handled at the network layer.

The cloud resources at the edge are encapsulated as self-containing VM that are pushed or pulled by code mobility paradigms or invoked with a dynamic synthesis of the VM in the edge devices. Cloudlets (Satyanarayanan et al. 2009, 2015) are self-managing VM-based “data centers in a box” that are executed at the edge devices. Mobile edge computing (MEC) (Mach & Becvar 2017) extends the functionality of cellular network base stations towards MEC hosts that run a collection of VM-based loosely-coupled services. MEC application logic is designed through the interactions of these services (Reznik et al. 2017). MEC allows to relocate application components between hosts in response to user mobility. REST-based interfaces are currently being considered as one of the MEC APIs. Mobile edge computing framework and architectural specifications are currently progressing towards standardization (Mach & Becvar 2017) by the European Telecommunications Standards Institute. Fog computing (Bonomi et al. 2012, Hong et al. 2013, Yi et al. 2015) relies on dense deployment of fog nodes, i.e. edge devices, that provide location-aware services for computation, data storage and networking. Mobile cloud computing (Fernando et al. 2013, Zhang et al. 2016) partitions application-specific tasks across the layers, including mobile devices at the device layer.

Edge computing is a mixture of Remote Evaluation (REV) and Code on Demand (CoD) code mobility paradigms that relocate application execution concerns from the user devices. On the one hand, REV is used to offload computations to the edge that enables the end devices to operate as thin clients and save resources. Offloading requires careful consideration, as it may consume more energy than local execution in the device (Kumar et al. 2013, Fernando et al. 2013). The VM or VM overlay size can be from hundreds of megabytes to several gigabytes (Satyanarayanan et al. 2009), thus VM transport and operations are heavy and time consuming for constrained wirelessly communicating devices. A static offloading decision, i.e. made at development time, does not address changing runtime environment characteristics. Dynamic offloading requires contextual information about resource availability and network characteristics. On the other hand, with CoD, a user invokes the migration of VMs from the cloud into the edge layer.

On the device layer, large amounts of real-time data are generated by large deployments of things. This introduces challenges in data management and transport,
making scalability one of the main challenges in IoT (Guo et al. 2013, Karagiannis et al. 2015). Typically, applications utilize network infrastructure components to disseminate information with publish-subscribe protocols that are considered sufficient to retrieve data streams from data producing devices. However, this can be questioned as opportunistic networks, that provide infrastructureless spontaneous connections of mobile devices, have become the next step in mobile ad hoc networks (Conti et al. 2015). These networks are characterized by a dynamic topology with no fixed infrastructure nor management authority. Connections are not reliable, have low bandwidth and high latencies. On the good side, deployments can be highly scalable and network coverage extended to areas without fixed networks. The detailed discussion of opportunistic networks is outside of the focus of this thesis.

Both the stationary and mobile devices have computational power that facilitates in-network data processing with proximity based communications (Uckelmann et al. 2011, Miorandi et al. 2012, Borgia 2014, Conti et al. 2015, Garcia Lopez et al. 2015, Want et al. 2015). In these devices, CoD paradigm is particularly useful to reduce the volume of application-specific data that needs to be transmitted from the device. Recently, for stationary devices, Docker\(^2\) technology provides a container-based mechanism to distribute application execution into the device layer. Containers encapsulate the application components and their dependencies as an application image that can be build automatically from source code. Containers are centrally orchestrated and can be migrated into a heterogeneous set of devices, provided that they run the same Linux kernel. According to Bellavista & Zanni (2017), Docker technology is feasible option for stationary IoT devices that have enough resources to run Linux OS.

Mobile devices have been used as opportunistic application execution platforms, e.g. mobile fog (Hong et al. 2013) and mobile cloud computing (Lane et al. 2010, Guo et al. 2013, Fernando et al. 2013) that partition applications across the architecture layers. In these horizontal solutions, mobile end devices in a location are opportunistically invited and configured into an application-specific federated computational platform. The roles of end devices are extended from data producers to an opportunistically operating set of data producers, consumers and mediators for each other's. However, the set of participating devices cannot be predetermined in advance. The devices may not be available at the same time nor know each other. Therefore, these application configurations are dynamic and no guarantee of successful execution can be given. This

\(^2\)https://www.docker.com/
results in general increased application management load as the devices need to be aware of nearby resources. Another concern is device mobility due to independent behavior of the device owners.

These challenges in edge computing and opportunistic mobile device use cases pave way for software agent based intelligent operation and interactions in a MAS in the context of IoT (Miorandi et al. 2012, Razzaque et al. 2016).

2.2.1 Use cases for software agents

It is not realistic to assume that a single system component would have complete real-time information about the whole IoT application state, let alone system state, and control the system satisfactorily with regard to all applications’ requirements. Rather, IoT is an example of an open system in dynamically changing environment, where information and control is beneficial to distribute into system components across layers. Novel ad hoc applications and on-demand functionality in IoT are likely to emerge from two directions: top-down in the cloud-centric model and bottom-up in edge computing. As discussed in Weiss (2013) and Shehory & Sturm (2014), software agents and MAS are well suited to such distributed computing environment. Software agents as application components in the context of IoT have been discussed by (Uckelmann et al. 2011, Miorandi et al. 2012, Fortino 2016, Schatten et al. 2016). Characteristically, the cloud-centric vertical interactions can become horizontal and localized within a layer and across layers with software agents. To this day, software agents in the context of IoT are largely unexplored. Lack of agent-related standardization is an obstacle limiting the general utilization of agent technologies.

Previous work, e.g. Muldoon et al. (2008), Vinyals et al. (2011), Fortino & Galzarano (2013), Schatten et al. (2016), discusses challenges in MAS design, organization and implementation in the context of resource-constrained embedded devices. With MAS organizational methods, system devices controlled by software agents can be organized with regard to application requirements. The design and deployment of application-specific agents into these devices in relation to available resources is a crucial concern. Single agents have different strategies of operation and may be competing for the same resources. MAS communication overhead may reduce the benefits of coordinated agent operations in following the application-specific requirements, such as energy efficiency. Another issue is how MAS is integrated into the IoT software framework. MAS applications can be built atop IoT middleware, but also an agent-based middleware
enables programming of applications atop system devices. The other way around, the agent middleware can be extended with system devices as agent platforms.

Use cases for software agents and MAS in IoT and edge computing have been identified, as illustrated in Figure 9. A MAS can operate completely in the cloud side (Kazanavicius et al. 2009, Aversa et al. 2010, Fortino & Russo 2013, Leong & Lu 2014, Manate et al. 2015, Schatten et al. 2016, Hafez & Elgamel 2016, do Nascimento & de Lucena 2017). Agents at the clouds are assigned tasks in data processing and management, as well as in representing, controlling, organizing and monitoring devices on lower layers (SA1). On the middleware layer, software agents (SA2) implement, orchestrate and manage services, process data from things, control deployment of things and monitor resource usage, represent things and encapsulate their heterogeneity (Katasonov et al. 2008, Fortino & Russo 2013, Wang et al. 2013, Bonomi et al. 2014, Leong & Lu 2014, Ayala et al. 2015, Hernández & Reiff-Marganiec 2016, Giordano et al. 2016, Jararweh et al. 2016, Razzaque et al. 2016). Similarly, software agents in the edge layer operate in VMs in edge devices (SA3). Edge devices also host agents that the things are not capable of hosting, e.g. deliberative agents (Sánchez López et al. 2011, Leong & Lu 2014). Cross-layer MAS in IoT (Kazanavicius et al. 2009, Leong & Lu 2014, Schatten et al. 2016) place agents operating different roles in different layers. As an example, physical products have agent-based representations in the cloud (SA1), but also carry mobile agents (MA4) in their memories (Chen 2013).

At the device layer, software agents (SA4, SA5) bring autonomy, adaptivity and self-* properties for things (Sánchez López et al. 2011, Angulo-Lopez & Jimenez-Perez 2012, Chen 2013, Godfrey et al. 2013, Mzahm et al. 2014, Semwal & Nair 2016). In reality, a subset of these agent properties can be implemented due to constrained

Fig. 9. Current software agent (SA) and mobile agent (MA) use cases in IoT.
resources in things. Resource-rich things may host more capable agents (SA5) that perform resource-consuming tasks, such as system configuration, monitoring, device management and complex data processing. In the low-end things, agent (SA4) operation focuses on optimizing application execution in relation to environment characteristics of the system and available resources in the individual nodes or in the whole WSN. Constrained WSN nodes, in particular, benefit from autonomous and adaptive application execution on the dynamic environment (Marsh et al. 2004, Tynan et al. 2005a,b, Payne 2008, Dagdeviren et al. 2011, Vinyals et al. 2011, Vukasinovic et al. 2012). WSN nodes are typically constrained in a number of ways that the agent needs to be aware of. Low-power nodes are commonly battery-operated, which limits their lifetime. Communications are slow and intermittent. Computational power and memory of nodes are greatly limited. Concerning deployments, agents can be placed in different ways. A single agent can operate in the sink node (SA2) and control the node operation. This deployment is easy to program, but it is not scalable when the WSN size grows. A number of agents, e.g. one in each node (SA4), can control the node behavior and collaborate as a MAS to achieve a system-wide goals. These agents can co-operatively reason about the environment based on data collected with their sensors, plan the sensing task execution accordingly, adjust sensing parameters dynamically and optimize the data dissemination path towards the edge layer. At best, agents can proactively decide which tasks the nodes perform and how. This kind of deployment is flexible and scalable, has no single point-of-failure and operates locally even when suffering from disconnections to the edge. However, agent-based interactions increase communication costs and complex interaction protocols may be too burdensome for constrained nodes. In addition, typical WSN node operates on duty cycle, i.e. sleep and sense cycle, therefore its reachability is limited while sleeping.

Mobile devices are scalable and cost-effective way to deploy applications into the environment (Conti et al. 2015). With mobile devices, software agents (SA5) have more processing power and memory for their operations, for example to host deliberative agents (Bellavista et al. 2002, Bergenti & Poggi 2001, Lawrence 2000, Tarkoma & Laukkanen 2002, Carabelea & Boissier 2003, O’Grady & O’Hare 2004, Ughetti et al. 2008, Agüero et al. 2009, Santi et al. 2010, Bergenti et al. 2014). Agents in mobile devices hide environmental complexities from the user, bring in self-* properties and context-awareness for the device operation and enable adaptivity and planning for application execution. However, agents in phones have restrictions as well. The agents should not interfere with the normal use of the smartphone (Lane et al. 2010). In
addition, power consumption and sharing of physical components, e.g. sensors, between multiple applications may become bottlenecks for agent operations. The communication issues identified in mobile computing and opportunistic ad hoc networks (Chess et al. 1995, Bellavista et al. 2006, Conti et al. 2015) still persist. These kinds of networks are not designed with specific application in mind and there is no supporting infrastructure to interconnect devices, nor move data. Network topology is dynamic and partitioned, connections are often unreliable and message routing is dynamic. In these scenarios, the split agent (SA2, SA5) functionality prevails but suffers from disconnections. Agents on the devices would operate on local data, whereas agents on the infrastructure would operate as application-specific proxies or coordinate ad hoc scenarios.

A distinct use case for software agents in IoT is smart objects (Kortuem et al. 2010) with varying degrees of operational capabilities, dynamic interoperability and context- and self-awareness. A smart object has capabilities to observe and react to changes in the environment, interact with other system components and exhibit opportunistic goal-oriented behavior. These properties are commonly associated with software agents (Fortino et al. 2012). The agent-based smart objects facilitate autonomous adaptive operation and ambient intelligence in IoT applications (Angulo-Lopez & Jimenez-Perez 2012, Fortino et al. 2012, López et al. 2012, Wang et al. 2013, Fortino et al. 2014, Hernández & Reiff-Marganiec 2016, Schatten et al. 2016) that benefits from the capabilities of deliberative agents, i.e. reasoning, learning and proactivity. The question and justification for the utilization of smart objects is the same as for software agents: how the applications are implemented to operate in smart ways and how the application execution is distributed with smart objects and other components into the layers of the IoT architecture?

While software agents can handle dynamicity in the resource availability and in the environment, they are tightly coupled into the hosting device. Mobile agents bring in the additional adaptation with migration that enables to relocate application execution. The discussed mobile agent benefits are valid for the opportunistic IoT application with resource-constrained WSN nodes and mobile devices. Mobility and asynchronous operation distributes application execution load and increases robustness, fault tolerance and scalability. Mobile agent operations only affect a targeted set of devices (Szumel et al. 2005, Cecílio & Furtado 2014). In the end, the IoT computing platform may not support software agents, but the devices could still employ a mobile agent platform for their benefit.
The IoT edge applications that utilize mobile agents, generally rely on partitioned application execution, where agents operate either at edge devices (MA1) or end devices (MA2, MA3), e.g. Huang et al. (2013), Angin et al. (2015), Satoh (2016). Mobile agents in mobile devices (MA3), with regard not only to mobile phones but also to other types of mobile devices, have been used to implement applications atop the opportunistic set of devices and to operate as dynamic proxies for the devices in the infrastructure side (Spyrou et al. 1999, Kotz & Gray 1999, Bellavista et al. 2002, Cao et al. 2004, Vasiu & Mahmoud 2004, Bellavista et al. 2006, Muldoon et al. 2006, Su 2008, Urra et al. 2009, Vyroubal & Kušek 2013). Mobile agents at the infrastructure side autonomously migrate according to the user mobility to optimize local resource access for the application of the user, balance application execution load and encapsulate application state for asynchronous operation in the infrastructure side. The benefits of mobile agents for in-network data processing (MA2) are well-studied for the low-end things, e.g. low-power WSN (Qi et al. 2001, Tyan et al. 2005b, Szumel et al. 2005, Chen et al. 2007, Dagdeviren et al. 2011, Vukasinovic et al. 2012, Fortino & Galzarano 2013). On the one hand, mobile agents can be utilized as an integral part of the middleware to enable multiple applications to co-exist in the same set of end devices. On the other hand, mobile agents can be programmed for an individual utility task that operates atop existing edge platform. Nonetheless, similar benefits as in agent migration can be achieved with macroprogramming the devices (Oliver et al. 2014) that is realized through CoD. However, macroprogramming is centrally coordinated in general and targeted for the whole deployment at once and thus enables a single application for the whole system at once. With regard to macroprogramming and management of the deployments by a human operator, it is true that applications can be tailored and optimized for the environment in a way that the solution outperforms autonomous mobile agent-based solution in a number of respects. But, this results in a loss of generality and dynamicity.

2.2.2 Resource-constrained things

IoT applications rely on the data collected by large-scale deployments of physical and virtual things. These deployments commonly take one of two forms: things cover a wide area in a sparse grid of small-sized sensors or things populate densely an area for highly localized measurements. Things have different physical embodiments or virtual representations and are equipped with varying capabilities for communication and
computation (Atzori et al. 2010, Miorandi et al. 2012, Borgia 2014, Sheng et al. 2015). Physical things are constrained in terms of physical size, data storage, power source and energy consumption for computation and communication. Therefore, the operation of things is commonly optimized for the task at hand. In pervasive deployments, things can be deployed into inaccessible locations, where autonomous unattended operation with minimum human intervention is required. In such locations, low-power operation prolongs the thing’s expected lifetime, but nevertheless, things are expected to act, interact, cooperate and adapt in relation to the environment.

The common software and hardware components of a physical thing are shown in Figure 10a. Application code runs atop the OS that runs in a microcontroller unit (MCU). The OS provides an API for programming the device. Things are equipped with sensors to collect data about their environment. Optionally, things are equipped with actuators to manipulate their environments. A transceiver module is integrated for communications. Lastly, things need power supplies, where a rechargeable battery is often used. As an example, typical WSN node consists of sensing unit, processing unit, transceiver and a power unit (Akyildiz et al. 2002). Physical things are commonly realized as low-power embedded systems, consisting of a MCU, optional peripheral components and a power source. MCU hardware architecture is shown in Figure 10b. The MCU components are central processing unit (CPU) to execute programs, program memory to store the application program, RAM for application data storage and I/O interfaces for peripheral components (Peckol 2007, Chapter 1). Additionally, MCU may

---

3The difference to a general purpose computer is that an embedded system is typically created for a dedicated function only and interacts with its environments as a part of a larger system (Peckol 2007, Preface).
have integrated non-volatile memory based on Electrically Erasable Programmable Read-only Memory.

A minimum embedded OS kernel provides methods to execute, schedule, synchronize and communicate between application-specific tasks (Peckol 2007, Chapter 11). Baccelli et al. (2012) and Gaur & Tahiliani (2015) discuss the OS requirements for resource-constrained things. A modular OS and network protocol stacks allow hardware features to be customized for the particular application and their use optimized. OS memory footprint in the program memory and RAM should be minimized to give room for application code and data. On the MCU, low complexity operations facilitate meeting real-time constraints and use of sleep modes to reduce energy consumption. When the OS supports multitasking, multithreading is favorable for sharing the MCU resources. Standard APIs and high-level programming languages are beneficial for simple and fast programming and for cross-platform application development for multiple different hardware platforms. API communication model needs to facilitate sporadic and on-demand connections, instead of continuous connections, to reduce communication energy consumption and take advantage of the sleep mode. Low-level details of the communication protocols are often hidden from application developers. In resource-rich things, a full Internet Protocol (IP) stack can be employed, but transceivers commonly operate with non-IP protocols.

For low-end MCU with 8-bit CPU architecture, well-known general purpose OSs are TinyOS (Levis et al. 2005) and Contiki OS (Dunkels et al. 2004). These OS’s are based on the event-driven programming model and have multitasking capabilities. Low-power operations are supported through methods to reduce power consumption and lightweight communication protocols. Recently, off-the-shelf hardware platforms for things on the low-end category have appeared. A well-known example is the Arduino boards 4 that are based on Atmel’s AVR family of 8-bit MCUs. For example, the Arduino Uno is based on ATmega328 MCU with 32 kB of program memory, 2 kB of RAM and runs on 16 MHz. Arduino firmware and API provide high-level programming abstractions and software libraries to utilize the board features, including general purpose analog and digital I/O to integrate various components. Often, these platforms are bundled with a specific cloud service interface that eases IoT application programming. In this thesis, the Arduino boards are utilized as mobile agent platforms, as presented in Section 3.

4 https://www.arduino.cc/
For more resourceful MCUs, based on 16- or 32-bit CPU architecture, more advanced OSs exist that are commonly known as real-time operating systems. These OS’s guarantee rigid time constraints that are met with deterministic behavior in application execution (Peckol 2007, Chapter 11). Examples include a task-based FreeRTOS\(^5\) and RIOT OS (Baccelli \textit{et al.} 2012) that have been implemented for a large number of MCUs and for common personal computers (PC). RIOT OS features multitasking on a microkernel that can be configured with different hardware functionalities. RIOT supports a full set of common Internet protocol stacks. Raspberry Pi\(^6\) is a well-known example of a single-board computer based on significantly more powerful 32- or 64-bit CPUs with 1 GB of RAM and runs on 1.2 GHz. The OS is based on Unix variants. Raspberry Pi’s are not strictly embedded systems, but nevertheless popular platforms for things today.

The Arduino boards, Raspberry Pi and TinyOS and Contiki OS support application programming with the C and C++ programming languages. High-level programming abstractions are provided by the APIs that limit the possibilities to use advanced MCU features. For the more resourceful MCUs, the OS often supports application programming with a set of well-known programming languages. An example is Java with reduced JVMs that have appeared for 8-, 16- and 32-bit MCUs (Maye 2013).

Smartphones have become attractive IoT application development platforms due to programmability supported by OS, increased processing power with 64-bit CPUs, multiple wireless communication interfaces and a variety of integrated sensors (Kamilaris & Pitsillides 2016). Modern OS’s include Android\(^7\) and iOS\(^8\). Use cases for smartphones have been urban sensing (Lane \textit{et al.} 2010) with mobile devices involved in data collection about the environment and the user behavior. As a general purpose communication interface, Web technologies have been used to realize interactions within the system (Kamilaris & Pitsillides 2016). In this thesis, Android smartphones are used as a mobile agent platform in Section 3.

\section{Resource-oriented architecture}

Moreover, Web technologies facilitate human-machine interactions across the IoT architecture layers from cloud applications to the level of individual things in an open environment (Want et al. 2015). From the beginning, the design goal of the Web was “to be a shared information space through which people and machines could communicate” Berners-Lee (1996)\(^9\). Information on the Web is to be machine-indexed, where search engines and human readable HTML documents are used to realize interactions. All information is identifiable with Uniform Resource Identifiers (URI). Information is linked through hypermedia links that provide structuring of the information in the HTML documents. HTTP is used to traverse the links.

Web services, applications and architectures today rely on two approaches: SOA\(^10\) and REST-based ROA. SOA organizes applications into a set of Web services that rely on standardized WS-* specifications to describe common service features and interaction protocols. SOA technologies rely on XML as the general structured document format. XML-based Web Service Description Language (WSDL) is used to describe Web services in full, including exposed functionality that is accessed with RPCs. Each RPC call has its own WSDL specification that includes the call’s scope, its parameters and return data types. XML-based Simple Object Access Protocol (SOAP) provides the message envelope that is sent as the payload in HTTP requests. This standardized structured approach facilitates programmable Web for machines.

The REST architectural style was presented in Fielding (2000). In this style, the key abstraction is a resource, which is any information that is named and given an identifier, i.e. URI, as a reference. A URI defines the resource and its scoping information in a URI path, e.g. a hierarchy of URIs in a server. REST provides an interface to access a resource through its URI and manipulate the returned resource representation, i.e. the resource value. The URI is a resource identifier that is not bound to a particular representation and does not change when a representation changes. Thus, a resource may have different representations that are negotiated between the client and the server and realized by the server. A representation is data, metadata or mobile code, i.e. instructions how to render the representation from its data.

Fielding (2000) presents the following six REST architectural style design principles for Web architectures that emphasize generality and scalability of communication

\(^9\)The article proposes mobile agents to create user interfaces and distributed applications in the Web (p.74).

\(^{10}\)Here, SOA refers to the Web services based on WS-* standards, although the definition varies (Richardson & Ruby 2007).
interfaces not bound to any communication protocol or deployment of components. First, REST architectures follow the C/S communication model, where separation of concerns enables the development of clients and servers independently. The client-side application logic is separated from the data storage at the server. Second, REST communications are stateless, meaning that the request is self-contained and no session state is saved on the server side. The requests are safe and idempotent, i.e. have no dependencies that would reduce reliability and scalability of the system. Third, REST facilitates caches to expedite network performance for information that is already available in the system. This improves scalability by reducing request execution load. Fourth, a uniform interface guarantees that every resource is accessed with the same semantics. This simplifies interactions and decouples components in a REST-based system architecture. Each request is self-contained, i.e. does not require any additional information of how to process the request. URI identifies a resource but the request URI is separated from the resource representation. A response has the information to manipulate the resource through the representation and it may contain links to other resources. This interface supports the generality of REST as the interface does not determine how the resources are handled by the components. Fifth, layers and intermediate components can be introduced to enable load balancing and to encapsulate legacy systems. The intermediates fully understand the requests. However, Web interfaces are designed for large-grain data transfers and using intermediates may be suboptimal for other types of interactions. A system is considered RESTful when it fully complies with these principles. As the sixth optional principle, a client functionality can be extended with CoD paradigm to execute service code on the client side.

Richardson & Ruby (2007) elaborate further the REST principles as the ROA that is targeted for programmable Web. ROA applies to Web service design and implementation as a universal programming paradigm. Anything that has value for the clients, i.e. every piece of data and all data processing algorithms, can be exposed as Web resources and identified with unique URIs. A resource can be exposed through multiple different URIs that have different semantic meanings. Linking through URIs provide addressability that enables resource discovery. When URIs follow a well-known structure, i.e. hierarchy, machine clients can create their own entry points and build applications automatically by linking to representations with the uniform interface. For stateless communications, application state is kept on the client side and separated from the resource state that is maintained on the server side. Such asynchronous communications increase scalability and fault tolerance.
HTTP methods\textsuperscript{11}, with well-known common semantics, convey the operation that is executed for a resource given in the request URI. The server fulfilling the request extracts the meaning of the request from the combined semantics of the method and the URI to create a representation. Additional URI query parameters in the request provide additional information on how to build the representation, e.g. for an algorithmic resource. HTTP request headers can be used to negotiate client and server capabilities with regard to the request, e.g. the requested content format. HTTP response codes provide means for the client to understand what happened with its request and how to approach the result, e.g. when the server fails. When request semantics are maintained with safe and idempotent methods, ROA with HTTP is guaranteed to provide the same simple vocabulary and a uniform interface for both clients and servers.

SOA and WS-* specifications have been compared with REST and ROA principles in traditional Web services (Richardson & Ruby 2007, Pautasso \textit{et al.} 2008, Garriga \textit{et al.} 2016), in mobile Web services (Hamad \textit{et al.} 2010, AlShahwan \textit{et al.} 2011, Mizouni \textit{et al.} 2011, Hameseder \textit{et al.} 2011, Aihkisalo & Paaso 2012) and in embedded Web services (Schull \textit{et al.} 2006, Yazar & Dunkels 2009, Shelby 2010). SOA and ROA both provide Web-scale interoperability, but ROA is arguably more scalable as the application states are stored on the client side. Generally, REST is found to be a better solution for ad hoc interaction scenarios due to the simpler semantics, whereas SOAP is more process-oriented with advanced interaction features that are commonly beneficial in enterprise-scale systems (Richardson & Ruby 2007, Pautasso \textit{et al.} 2008).

SOA is feasible for the programmable Web as services expose their methods as RPC calls and interface program code can be generated automatically from WSDL descriptions with any programming language. However, the RPC calls rely on system specific constructs that are implemented without uniformity and with questionable reusability. Moreover, RPC requests are transmitted with XML-based SOAP and a reply is expected back in the same format. This introduces large overhead and complexity in message processing for resource-constrained devices. A SOA service is accessed through a single URI that is the end-point for all RPC calls, which means that SOA services are not addressable nor SOA resources linked as they do not have individual URIs and representations. SOA messages can be transmitted with any protocol. HTTP is utilized as a message transport protocol over the Internet. Typically, SOA services are requested through a HTTP method, i.e. POST, regardless of the encapsulated operation.

\textsuperscript{11}https://tools.ietf.org/html/rfc2616
ROA, in turn, takes advantage of the combined semantics of HTTP methods, URIs and return codes. The extensive set of HTTP features can be used to describe the requested operation for the resource. This leverages well-known semantics of HTTP and URIs for application protocols. With the REST uniform interface, self-descriptive URIs in Web documents are easy to discover and follow by machines. REST interactions are transparent and cacheable in intermediate network components. With HTTP as the application protocol, REST interactions are more lightweight, no document envelopes are enforced and content format is negotiable. Fewer bytes are transmitted and less request processing is needed on the client and server sides, which is beneficial for resource-constrained embedded devices. In this thesis, ROA is considered a feasible solution for resource-constrained IoT devices.

2.3.1 Embedded Web services

When Web technologies are integrated into embedded networked devices, as for example in Debaty & Caswell (2001), Kindberg et al. (2002), Kamilaris & Pitsillides (2016), their resources become searchable and browsable with common Web tools. Moreover, the resources hosted by devices can be automatically indexed and transparently added to applications. These devices are able to run a limited Web server that expose representations of themselves, e.g. dynamic sensor data, on a Web page (Agranat 1998, Borriello & Want 2000, Delin 2002, Wilde 2007, Priyantha & Kansal 2008, Stirbu 2008). The Web page may additionally provide a user interface to control the device or its resources. Web frameworks for embedded devices include the well-known project JXTA-C (Traversat et al. 2003) and SenseWeb (Grosky et al. 2007), REST-based frameworks (Dawson-Haggerty et al. 2010, Ostermaier et al. 2010) and SensorThings API (Open Geospatial Consortium 2017).

For the programmable Web, embedded Web services facilitate building automatically distributed applications through bi-directional end-to-end connectivity (Shelby 2010, Castellani et al. 2011, Ishaq et al. 2013, Sheng et al. 2013). However, the current Web protocols with stateful HTTP connections atop full Transmission Control Protocol (TCP) stack and the required request-response semantics are difficult to utilize due to large overhead (Priyantha & Kansal 2008, Shelby 2010, Groba & Clarke 2010, Castellani et al. 2011, Colitti et al. 2011b). Therefore, small protocol overhead and message payloads are favored to reduce complexities of request handling and to avoid message fragmentation. A Web server is expected to be always ready to accept requests, but
in reality, the embedded devices are largely battery-operated, trying to optimize their power consumption by sleeping most of their lifetime with the transceiver turned off (Priyantha & Kansal 2008, Colitti et al. 2011b, Kovatsch et al. 2011). For this reason, the C/S model for Web interactions is not well supported. Instead, publish-subscribe and eventing are the preferable interaction models. Moreover, the wireless communication environment is characterized by frequent topology changes, low throughput and high packet loss (Shelby 2010). Interactions in such networks are commonly asynchronous and short-lived as it is challenging to transmit large amounts of data. The User Datagram Protocol (UDP) protocol is better suited for such environments, being resistant to topology changes, having less overhead for short transactions and being unaffected by high packet loss rate (Castellani et al. 2011).

Optimization of different aspects of embedded Web services has been proposed as well. Optimized duty cycling, reduced TCP/IP stack and reduced payload formats have been suggested in Kang et al. (2006), Priyantha & Kansal (2008), Duquennoy et al. (2009a). pREST (Drytkiewicz et al. 2004) realizes a simple get, set and invoke functionality to access resources in embedded devices. A pREST request is used to transmit control commands in a simple XML document schema. Efficient XML Interchange\(^\text{12}\) format defines a reduced XML schema that relies on device profiles describing allowed message options (Castellani et al. 2011, Kyusakov et al. 2014). Castellani et al. (2011) present BinaryWS with a reduced implementation of the HTTP header and introduce new HTTP methods for publish-subscribe interactions. However, the generality of these approaches is questionable without standardization. In addition, gateways (Luckenbach et al. 2005, Trifa et al. 2009, Dawson-Haggerty et al. 2010, Qin et al. 2011) translate protocols and expose resources in constrained networks, e.g. WSN, into the Web with RESTful interfaces. The WSN itself may internally utilize proprietary protocols.

Recently, REST based WoT architecture has been proposed to integrate embedded devices into the Web. WoT application model is a Web mashup with ad hoc integration of both physical and virtual things and existing Web services (Guinard et al. 2009, 2010a,b, Groba & Clarke 2010, Zeng et al. 2011, Heuer et al. 2015, Mashal et al. 2015). Event-driven interaction model is preferred due to the limited capabilities of things (Duquennoy et al. 2009b). WoT is based on flat architecture where things are both clients and servers for each other and capable of both direct and indirect interactions.

\(^{12}\)https://www.w3.org/XML/EXI/
Indirect interactions utilize gateways that expose RESTful interfaces for the connected non-IP devices, e.g. WSN nodes. HTTP is the application protocol, where common data formats include JavaScript Object Notation (JSON) and microformats (Guinard et al. 2011). WoT gateways and proxies handle resource indexing for search engines, cache content of things, balance request handling load and provide protocol translation with stateless communications (Duquennoy et al. 2009b, Trifa 2011).

### 2.3.2 Constrained RESTful Environments

IETF has established a working group CoRE\footnote{https://datatracker.ietf.org/wg/core/charter/} that proposed a set of simplified Web protocols and a REST-based framework for low-power devices on constrained networks. The aim is to facilitate short-lived interactions, optimize protocol overhead and payload encoding, implement security features and resource discovery mechanisms (Shelby 2010, Colitti et al. 2011b, Castellani et al. 2011, Ishaq et al. 2013, Sheng et al. 2013). With a new standardized Web protocol for resource-constrained devices, programmable Web becomes possible the same way as Web services today.

Request for Comments (RFC) 7228\footnote{The CoRE RFCs and Internet Drafts can be found in https://datatracker.ietf.org/wg/core/documents/} defines terminology for the framework. Constrained networks have low bitrate or throughput, suffer from high and highly varying packet loss, deteriorate with large packet sizes, have asymmetric link characteristics and lack advanced network services. Constrained devices have limited CPU power, RAM, program memory and power resources, making access to the device resources limited. Nevertheless, these devices host one or more resources that represent sensors, actuators, data and possibly other information. As with Web services, these devices do not simply upload data to the cloud, but are expected to interact and access resources on the other devices by direct queries or by the publish-subscribe pattern. The RFC 7228 defines categories for the devices based on their hardware resources. Class 0 devices have less than 10 KiB of RAM and less than 100 KiB for program memory and are not able to communicate directly in the Internet in a secure manner. Class 1 devices have around 10 KiB of RAM and less than 100 KiB for program memory and employ a full protocol set to communicate in the Internet. Class 2 devices have at least 50 KiB of RAM and 250 KiB of program memory and are capable of supporting the protocol stacks used with common PCs, but still benefit from energy efficient communication protocols. In this
thesis, the embedded devices utilized in the real-world evaluations are Class 0 devices that communicate with a subset of CoAP functionality. The mobile devices used in the evaluation are in the Class 2 category.

CoRE has defined CoAP (RFC 7252) as a generic Web protocol for constrained devices. CoAP complies with the REST principles and meets the requirements for low protocol overhead, low parsing complexity and operational simplicity. To reduce underlying communication protocol overhead, CoAP is based on UDP and not on TCP. The devices are expected to act in both roles of Web clients and servers. CoAP communications are designed to be asynchronous, where the interaction model is request-reply. Four CoAP methods are defined that are semantically similar to corresponding HTTP methods and are considered safe and idempotent. GET is used to retrieve a resource representation. POST method is used to send data to be processed on the server and as a result, new resource may be created on the server. PUT is used to update or create the targeted resource with the included new content. DELETE requests a deletion of the targeted resource in the host. PATCH and FETCH methods are defined in RFC 8132, but are not utilized in this thesis. Similarly as in HTTP, the interactions are built on CoAP methods, URIs and response codes. Due to the asynchronous communication model, each interaction has an identifier to separate it from other interactions. Advanced CoAP features include caching, publish/subscribe (RFC 7641), group communication (RFC 7390), multicast messaging, security and blockwise transfers (RFC 7959). In this thesis, a subset of CoAP functionality is implemented for the class 0 devices.

CoAP message structure is shown in Figure 11. A CoAP message consists of a header, a token, optional set of options and a payload. The message header includes three fields, in addition to protocol version: message type, method or response code and message identifier. The message type is either confirmable message, non-confirmable message, acknowledgement message or interaction reset. The method code describes the CoAP method. CoAP defines three sets of response codes that are discussed in detail in the RFC 7252: success, client error and server error. Message identifier is used for reliable messaging. Next, the token is used as the interaction identifier that is independent of the message identifier. CoAP option format includes an option number,
the length of the option field and the option value. Option numbers utilize delta encoding, where the same option can be included in a message multiple times, e.g. to describe a URI hierarchy. A set of option numbers is standardized. The following standardized options are utilized in this thesis. The Content-Format option describes the content media type of the message payload, for example XML or JSON. The Uri-Path option gives the target resource URI and the Uri-Query option describes additional arguments utilized in the request processing. CoAP specifications allow the use of non-standardized private message options. A set of these options are defined for the proposed mobile agent architecture as described in Chapter 3. CoAP option value can be either a sequence of bytes, a non-negative integer or a string. The RFC 7252 defines maximum length for CoAP message being 1152 bytes, including maximum payload size 1024 bytes. The idea behind this limitation is to avoid fragmentation of the message into several packets in the underlying network transport layer.

The CoAP interaction model follows the request-response model of HTTP. When a server receives a request, it handles it and returns a response, possibly with a payload. The message token identifies a request-response interaction, regardless of the message identifier that is used for reliable messaging. A GET request has a URI defined with Uri-Path and Uri-Query options and may have additional CoAP options. If the request is successful, a response is received with a response code. Typically, the resource representation is piggybacked as the payload. Non-confirmable messages do not require a response. If the request is not successful, error codes are returned. Similarly, POST and PUT methods use a URI to identify a resource and the request data is given in the payload. A DELETE uses a URI to identify the resource to be deleted. Full CoAP messaging model and semantics are discussed in the RFC 7252. Examples of CoAP messages are shown in Chapter 3.

The CoRE framework defines a set of services to facilitate interactions between devices and for integration into the existing Web. A resource directory is a service that facilitates dynamic resource discovery. Devices are expected to register their resources to the directory and maintain their dynamic resource descriptions. The DRD (Liu et al. 2013a) is a realization of the resource directory utilized in this thesis. Further details of the DRD operation are outside the focus of this thesis. In addition, the devices are expected to facilitate direct resource discovery by providing a list of their resources through a well-known standardized root URI. The CoRE Link Format (RFC 6690) is used to describe the exposed resources in a standardized way. Each resource has an URI, which can be a hierarchy. Resources may have additional attributes that describe
resource metadata and access interfaces. Two additional attributes are used in the proposed framework. The attribute Resource Type gives a human understandable description of the resource and the attribute Content Type defines the media type of the resource content. Another standardized CoRE framework component is a proxy that translates between HTTP and CoAP requests (RFCs 7252 and 8075). A proxy implementation is presented in Chapter 3.

CoAP has been evaluated generally (Villaverde et al. 2012) and in comparison with other Web protocols (Colitti et al. 2011a,c; Lerche et al. 2012, Ludovici et al. 2013, Sheng et al. 2013, Levä et al. 2014, Daniel et al. 2014). The lowered hardware requirements and reduced communication overhead reduce communication energy consumption as expected. This is particularly observable when the frequency of C/S connections increases in the system, high volumes of data are transmitted or push model is used to transmit data. Kovatsch et al. (2011) evaluate CoAP based communications with regard to optimizing duty cycling in an embedded device. The results show that power consumption can be reduced significantly. Karagiannis et al. (2015) discuss that the publish-subscribe model is more energy efficient than CoAP as the request-response interactions require more bandwidth. The publish-subscribe model is based on brokers receiving messages from publishers and dispatching them to subscribers of specific topics and without direct interactions between subscribers. It is unclear how a mobile agent could migrate using such a protocol even though resource retrieval is facilitated by subscription. An example of such publish-subscribe protocol, targeted for resource-constrained devices, is the Message Queue Telemetry Transport (Banks & Gupta 2014) protocol. The protocol requires both clients and servers to store session states, which is not RESTful.

One implementation of the CoRE framework is Californium (Kovatsch et al. 2012) that contains a thin server for embedded devices to provide access to their resources, and client-side scripts for the application logic in resource-rich devices. Castro et al. (2016) propose executing applications in a client-side sandbox in a browser that provides the protocols for the constrained network. CoRE proxy implementations have been proposed, in Lerche et al. (2012), Bandyopadhyay & Bhattacharyya (2013), Ludovici & Calveras (2015), for transparent access and caching of WSN node resources for the Web. Internally, the WSN nodes communicate with CoAP or with any other deployment-specific protocol. This approach allows to notify clients of WSN resource updates, removing the need for client polling and for synchronized updates.
3 Research contributions

This chapter summarizes the contributions of the publications presented for this thesis. Section 3.1 presents a resource-oriented reactive mobile agent architecture for heterogeneous and resource-constrained IoT devices. Section 3.2 presents a mobile agent software framework IoT. Section 3.3 presents two mobile agent platforms for low-power embedded devices and for modern smartphones. Section 3.4 presents the evaluation of two IoT applications implemented with reactive mobile agents in the software framework.

3.1 Resource-oriented mobile agent architecture

The first research question (RQ1) deals with realizing mobile agents in resource-constrained heterogeneous IoT devices. In practice, such mobile agents should be hardware platform, OS and programming language independent to maximize their utility. To answer the question, a resource-oriented reactive mobile agent architecture was developed. Two implementations of the reactive architecture are presented for heterogeneous IoT devices. Reactive approach for agent architecture was selected due to limited resources in expected mobile agents hosts, i.e. the IoT devices. Hosting deliberative agents would require significantly more computational and communication resources from the devices, which can’t be anticipated in current IoT systems. A reactive agent uses sensors to acquire perceptions that are mapped into actions by the agent program. This operation is realized in the context of IoT as follows. The agent program implements the application-specific tasks assigned for the agent. The program acquires its inputs from three sources: the current agent state, the sensors of the host thing and agent-based interactions. These inputs are optional as not all of them are required. The program produces an action that is realized either as physically by the hosting thing’s actuators, as interaction through the thing’s communication interface, as agent mobility or as an agent state update. A program can result a number of actions of these kinds at once.

The architecture of a resource-oriented mobile agent is based on resource abstractions. The properties of a resource-oriented mobile agent in the context of IoT are shown in Table 1. The resource-oriented mobile agent architecture proposed in Publication I is shown in Table 2 and the corresponding URI schema is shown in Table 3. The agent
Table 1. Resource-oriented mobile agent properties that support IoT device heterogeneity.

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent and its components are abstracted as resources or references to resources.</td>
</tr>
<tr>
<td>Agent resources are identified with URIs.</td>
</tr>
<tr>
<td>Agent resources are addressable and linkable through a URI hierarchy.</td>
</tr>
<tr>
<td>Agent program controls the whole agent’s state as its representation.</td>
</tr>
<tr>
<td>Agent resource representations are not bound to any content format.</td>
</tr>
<tr>
<td>Agent interactions comply with RESTful interfaces.</td>
</tr>
<tr>
<td>Minimal agent has small size and memory footprint.</td>
</tr>
<tr>
<td>Minimal agent has state and optional references to other components.</td>
</tr>
<tr>
<td>Agent resources and state are usable in any programming language.</td>
</tr>
<tr>
<td>Agent program is not bound to any programming language or OS.</td>
</tr>
</tbody>
</table>

Table 2. Agent architecture for resource-oriented mobile agents.

<table>
<thead>
<tr>
<th>Agent Component</th>
<th>Element</th>
<th>Subelement</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>Agent Identifier, i.e. resource name</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Language Identifier</td>
<td>Code block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reference URL</td>
</tr>
<tr>
<td>Resource</td>
<td>Local</td>
<td>Itinerary: List of resource URLs</td>
</tr>
<tr>
<td></td>
<td>Remote</td>
<td>List of resource URLs</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td>List of resource URLs</td>
</tr>
<tr>
<td>State</td>
<td>State data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Local data</td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>Metadata items</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. The agent URI schema.

<table>
<thead>
<tr>
<th>Component</th>
<th>URI schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent URI</td>
<td>scheme://[host_address]/[agent_name]</td>
</tr>
<tr>
<td>Agent program</td>
<td>scheme://[host_address]/[agent_name]/code?language=[language_id]</td>
</tr>
<tr>
<td>Agent resources</td>
<td>scheme://[host_address]/[agent_name]/local/[resource]</td>
</tr>
<tr>
<td></td>
<td>scheme://[host_address]/[agent_name]/remote/[resource]</td>
</tr>
<tr>
<td></td>
<td>scheme://[host_address]/[agent_name]/static/[resource]</td>
</tr>
<tr>
<td>Agent state</td>
<td>scheme://[host_address]/[agent_name]</td>
</tr>
<tr>
<td>Agent metadata</td>
<td>scheme://[host_address]/[agent_name]/meta</td>
</tr>
</tbody>
</table>

architecture defines how agent components are described as resources and how the agent can be addressed and linked with a URI and a representation. A hierarchical URI schema provides individual URIs for all the agent’s components. The agent resource representation, i.e. its state, is built with the agent program and the other resource descriptions are also defined by the agent. The agent program is a resource that
may have multiple representations based on the agent programming language. The agent can be encapsulated in a structured document that conveys its components as resource references or their representations. The interactions of an agent are conducted with manipulating any resource in the system, in the environment or of another agent, through the resource abstraction in RESTful manner. An agent can be considered a resource-oriented agent when it has these properties and its interactions are RESTful.

The existing SOA-based agents differ from this definition significantly. An agent, the functionality and interactions of which are defined through WS-* documents is not resource-oriented, because its URI does not expose a resource that is addressable and linkable, but expose an entry point to a service. Moreover, the agent interactions are not based on a uniform RESTful interface. Another REST-based approach for agent platforms is to use wrapper interfaces that translate external HTTP requests into an internal request, such as FIPA ACL, and vice versa (Braubach & Pokahr 2013, Pico-Valencia & Holgado-Terriza 2016). However, the REST properties are not fully maintained in the wrapper interface and in request translation and handling. A resource URI still describes an entry point to a service, even when the HTTP method semantics are followed to access it. A URI includes a RPC method with its parameters. This kind of interfaces are called hybrid REST and RPC style (Richardson & Ruby 2007, p.16) as they are partially RESTful with regard to the semantics of HTTP methods and URIs (Garriga et al. 2016).

The agent URI scheme (Table 3) provides a unique identifier for the agent. Provisioning of the identifier is the responsibility of the agent programmer or the agent platform that creates the agent. An agent is addressable and linkable through the current hosting device address, the host platform root URI and the agent URI. Any agent component can be addressed by adding corresponding URI elements. The agent state is the agent program state and is addressed through the agent URI. Optionally, query parameters can be added to the request for rendering of the representation. For example, the agent program language may be addressed with a language identifier. The actual request content format is negotiated between the client and agent host and depends on the supported protocols, e.g. HTTP and CoAP. Individual resources URIs can have also a protocol scheme to address the resource heterogeneity in IoT. A rule is that if the URI does not have host address but only the resource name, the platform needs to search the resource location from the resource directory.

In the architecture (Table 2), the URI section contains the agent name as its resource identifier. The state section contains the current state and result of the agent program.
that is the resource representation of the agent. The content of the state is defined by the agent programmer and it can contain local execution data, e.g. instance variables that are used to rebuild the state. REST allows a resource representation to contain the instructions, i.e. code, how to render the representation from its data. Therefore this approach is RESTful. A state cannot be a reference, however, because in distributed systems the agent object and its separated state are difficult to maintain synchronized.

The agent program is included in the agent as a code block or a reference Uniform Resource Locator (URL). The format of the code block content is not enforced and can be application-specific, for example, a script or a serialized object. To support heterogeneous devices, the code section allows multiple agent programs written in different programming languages, each with a well-known language identifier, e.g. “python”, “javascript” and “intelhex”. The different code blocks are considered different representations for the same resource that is the agent program. If multiple agent programs would each be considered a different resource, then each of them can be used to produce a different state for the same agent resource. The agent as a resource must have only a single state, regardless of how it is created. Reference URLs assist in reducing the transmitted agent size, as discussed in previous Sections. The referenced program can be already installed on the host or pulled from the previous host or from a system code repository. All versions of the agent program must operate on the same set of resources, data and state. The platform is responsible for constructing an executable program by using a version.

The resource section defines three types of resources that are local, remote and static. Each type has its own section that contains a list of resource URLs. The local resources describe the agent’s itinerary, i.e. the agent migrates according to these URLs to access the resources in the given hosts. The local list imposes no restrictions on how the URLs are utilized, e.g. migration can be done in any order and the order can be determined at runtime. Additionally, in the Publications I and II, two migration policies are defined. The first migration policy, called Task, dictates that each resource host is visited only once. A resource reference is removed from the list after a visit. When the list is empty, the agent is destroyed. The second migration policy, called Service, dictates that the resource hosts are visited in rotation and no reference is removed from the list. If a migration policy is to be used, the policy is identified in the agent metadata section.

The remote and static resources are typically hosted in servers that do not have an agent platform, but generally, any resource can be accessed remotely even when migration to the host is possible. It is assumed that remote resources are accessed once
before the agent program execution in each iteration. Static resources are resources that
do not change during the lifetime of the agent, e.g. historical data. Once accessed, a
static resource can be saved as local variables to the state section and removed from the
list. The resource lists do not impose any order on accessing the resources. It is assumed
that the agent platform knows how to handle the resource representations and to deliver
them to the agents. If no address is given for the resource, a search is conducted in the
resource directory.

The separation and definition of these three resources types support agent’s adaptive
operation. The agent can modify resource types to reflect availability and migration
possibilities. The agents can decide dynamically at runtime how to access the resource,
i.e. with remote request or by migration, and modify the resource type accordingly.
Also, access to unresponsive resources can be controlled by modifying the list.

The optional metadata section contains agent information, e.g. the agent owner
and its home location, and possible additional execution data such as used ontology,
migration policy, or agent execution or resource retrieval costs.

Next, two implementations of the resource-oriented mobile agent architecture
are presented: a JSON structured document and a CoAP message. Considering
IoT interoperability, a structured document provides uniform representation for an
agent, is portable and easy to transmit with any protocol. The agent components are
straightforward to separate into a hierarchical document structure. JSON (RFC 7159\textsuperscript{15})
is a well-known data interchange format, today integral part of standardized Web
services. A JSON document describes a JSON instance that contains key-value pairs
that can be nested. A key is a string and a value is either a string, a number, an integer,
a Boolean, an array or a JSON object. With key-value pairs, the mapping of agent
components into JSON is straightforward as keys indicate the different components, as
in Table 2. The value content format is not enforced, but generic numeric types, plain
text characters and hexadecimal byte array for binary data are preferred. For example,
an agent identifier is given with key “name” and the corresponding value is a URI string.

The CoAP implementation addresses interoperability and device heterogeneity
by providing minimized representation of the mobile agent in a standardized format.
Mapping of the architecture to CoAP message format is presented in Table 4. The
CoAP representation of the mobile agent requires a set of application-specific non-
standardized message options, as shown in uppercase in the Table 4, in addition to the
\textsuperscript{15}https://tools.ietf.org/html/rfc7159
Table 4. Mapping of the mobile agent architecture into CoAP message format.

<table>
<thead>
<tr>
<th>Component</th>
<th>Option identifier</th>
<th>Option value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata</td>
<td>Content-Format</td>
<td>COAP_AGENT_TASK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COAP_AGENT_SERVICE</td>
</tr>
<tr>
<td>Code</td>
<td>COAP_OPTION_PROGRAM</td>
<td>Program block</td>
</tr>
<tr>
<td></td>
<td>COAP_OPTION_PROGRAM_REF</td>
<td>Program reference URI</td>
</tr>
<tr>
<td>Resource</td>
<td>COAP_OPTION_NEXT_ADDR</td>
<td>Local resource URI</td>
</tr>
<tr>
<td></td>
<td>COAP_OPTION_REMOTE_RES</td>
<td>Remote resource URI</td>
</tr>
<tr>
<td></td>
<td>COAP_OPTION_STATIC_RES</td>
<td>Static resource URI</td>
</tr>
<tr>
<td>Payload</td>
<td>State values as byte array</td>
<td></td>
</tr>
</tbody>
</table>

standardized options. The agent-specific CoAP message options are defined as follows. The standardized Content-type option is used to describe the message as a mobile agent message and to define the used migration policy. The option Program encapsulates the agent program in the Intel HEX format. Currently only native machine language format of the MCUs is supported, therefore no language identifiers are used. The option Program Reference gives the reference URI for the agent program. Regarding the different resource types, each type has its own option: the option Next Address for the local resource, Remote Resource for remote resources and Static Resource for the static resources. The value is the resource URIs that possibly contain the host addresses. CoAP allows several instances of the same option in a message, therefore resource lists can be added to the message. A protocol schema is not defined for the options containing URIs or URLs as only CoAP communications are allowed. The option values are byte arrays that follow the bit and byte order of the ATmel AVR architecture. The 8-bit Arduino boards that are used in evaluations, are based on this architecture. Illustrations of two example mobile agents and their resource bindings are shown in Figure 12.

To provide an evaluation metric for the proposed agent architecture, with regard to agent platforms for embedded devices, the sizes of different mobile agents are considered. In Publication I, the mobile agent size as a CoAP message was 66 bytes that is well below the maximum CoAP message size. The agent was designed with one local and one remote resource and an agent program that was a straightforward copy of local resource data to the agent state with C standard library function memcpy. The program size was 20 bytes. Corresponding size of similar JSON mobile agent, with the agent program written in JavaScript and encapsulated in a HTTP request was 586 bytes. The size of the JSON mobile agent in Publication III, used for sensor data collection and
averaging the data values, was 1961 bytes. In comparison to optimized script-based mobile agents, the maximum size of a mobile agent in González-Valenzuela et al. (2010) was set to 170 bytes and in Fok et al. (2009) size of agent with a bytecode program starts from 120 bytes. A serialized Java mobile agent size in a WSDL document starts from tens of kilobytes in Artail & Kahale (2006). Mobile agents in structured documents have been presented from the early days on, e.g. in the MIME format, separating agent program and data (Lingnau et al. 1995). Examples of mobile agent encapsulated in XML documents include Brazier et al. (2002), Steele et al. (2005), Muldoon et al. (2006), Artail & Kahale (2006), Wu et al. (2007), Chen et al. (2008), Chou et al. (2010). With Web technologies, HTML and HyperText Markup Language version 5 (HTML5) documents have been used in Mitrović et al. (2014), Voutilainen et al. (2016). These documents contain generally the same agent architectural components or a subset of the components, e.g. only a reference to the agent program. Reported XML document envelope sizes for mobile agents start from 300 bytes (Wu et al. 2007, Chen et al. 2008, Chou et al. 2010). An FIPA ACL message size in a XML document can be hundreds of bytes (Chen et al. 2008).

Regarding interoperability and agent operations in resource-constrained heterogeneous IoT devices, the architecture is hardware platform, OS and programming language independent and can be presented in different data formats, e.g. a structured JSON document and a CoAP message. The architecture does not impose limitations on how the agent program is presented and supports multilanguage agent programs. A similar solution was proposed in Peine & Stolpmann (1997), Gray et al. (2002), with platforms providing a set of services for multiple agent program languages that are interpreted or running atop a VM. However, the agents still operated on a single specific language. The lightweight CoAP presentation permits agent migration and operations in the lowest category, i.e. class 0, resource-constrained IoT devices as defined by IETF CoRE. The presented CoAP mapping provides minimally a very small mobile agent...
size that can be transmitted as a single message. From the architectural point-of-view, the limiting factors for mobile agent operations are the maximum CoAP message size versus the agent size, where the goal is to avoid message fragmentation if not specifically requested. The components contributing the transmit size of an agent are the program, number of different resources and the state. The program and required resources can be referred to, which reduces the transmitted agent size significantly. The metadata component in the agent enables additional parameters to be passed. In general, the proposed architecture enables building the agent executable from its components and system resources, regardless of their specific implementation and how they are utilized. Lastly, the architecture does not impose weak or strong migration. As CoAP message are transmitted atop UDP, additionally multi-hop migration becomes possible.

In conclusion, with regard to the RQ1, the proposed reactive mobile agent architecture has the required components to enable its autonomous operation and is defined according to the REST and ROA principles. The agent and its components are transparently exposed as addressable and linkable resources identified with a URI hierarchy. The agent program state represents the agent as a resource. A RESTful uniform interface for the agent as a resource and agent-based interactions are realized through a subset of the standardized HTTP and CoAP request-reply semantics.

### 3.2 Mobile agent software framework

RQ1 also deals with enabling mobile agent operations at the device layer of the common layered IoT system architecture. As discussed, IoT systems consist of heterogeneous devices across the layers that utilize distributed services. From software development point-of-view, software components in IoT operate on devices with different OS’s and hardware capabilities, which makes their integration challenging. A software framework provides means to integrate software components, enables them to communicate and simplifies the creation of distributed applications with common abstractions and reusable components (Fayad & Schmidt 1997). A framework should avoid application-specific details and be extensible for new functionality. The presented framework in this Section enables the autonomous operation of the resource-oriented reactive mobile agents presented in the previous Section. The ROA approach provides the underlying principles for the framework utilization in IoT application development.

The software framework relies on two existing standardized frameworks. On the one hand, the IETF Core framework integrates resource-constrained devices into IoT
systems with RESTful interfaces and provides an optimized interaction Web protocol for the devices. On the other hand, the FIPA abstract MAS architecture specifications address the services needed by an agent to discover system resources and to interact in the system. A mapping is presented as to how a minimum set of FIPA specifications can be implemented with the CoRE framework services and software components. The FIPA specifications are not strictly followed with regard to agent interaction specifications that include FIPA ACL, its request-reply semantics, agent dialogues and common ontologies. Instead, the CoRE framework imposes RESTful interfaces where HTTP and CoAP request-reply semantics are used, interchangeably, to enable reactive mobile agent operations. The proposed mapping of FIPA MAS components into CoRE components is shown in Figure 13. The realization of each FIPA component is described next.

![FIPA ABSTRACT ARCHITECTURE](image)

**Fig. 13. Realization of the FIPA MAS abstract architecture with the IETF CoRE framework.**
Message Transport Service (MTS) provides a message transport service. The FIPA (2002d) specification declares that the underlying OS message transport mechanism is a sufficient MTS. Therefore, each interacting system resource is required to host a CoAP server that sends messages directly to the system components and receives messages. An agent calls a platform agent API method to request a message, with the resource URI or URL as a parameter, to be sent. The server compiles the corresponding CoAP message based on the semantics described in Table 5 and sends the CoAP message with UDP. Reliable message delivery is based on the syntax of CoAP methods and response codes. After a response is received, it is parsed and the payload returned to the agent. The details of the agent platform API and corresponding messages are given in the next Section. Contrary to the FIPA specification, multiple recipients are not allowed for a message, the agent is not expected to parse the message. This simplifies the requirements for agent operation. Additionally, the CoAP server maintains a list of hosted resources, which is required by the CoRE framework.

The RESTful API, shown in Table 5, is described in detail in Publications I and II. The purpose is to provide same HTTP and CoAP semantics for the agent resource access as with any other Web resource. The API is based on the HTTP and CoAP semantics. The methods are universal communication primitives and a hierarchical URI schema consists of the host address, platform root URI, agent URI and possible query parameters. The supported content formats in the framework are JSON for HTTP and byte array for CoAP. POST method semantics allow creation of a new resource, where the server is responsible for definition of the URI of the new resource, PUT method requires that the client knows the server-side URI of the resource (Richardson & Ruby 2007, p.220). Here, the agent URI is known by both, but its place in a root URI hierarchy is determined by the server, i.e. the agent platform. Therefore, PUT is not implemented in the framework. The request response codes and reply payload indicate success or failure of the requested action, as described in the HTTP and CoAP specifications.

The FIPA specifications (FIPA 2002a, 2004) require directory and services for the agents and for system services. Optionally, agent discovery service operates as a wrapper for non-FIPA agents. The corresponding component in the CoRE framework is the resource directory that is realized as the DRD (Liu et al. 2013a) in the presented framework. The DRD is based on P2P networks, but details of its functionality are outside the focus of this thesis. Agents and services are both considered as resources in the DRD that are accessed through the RESTful interface. The hosts are
Table 5. REST-based API to control agents and access their resources.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Retrieve agent state or any component state directly or via proxy</strong></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>GET scheme://[host_addr]/[agent_name]</td>
</tr>
<tr>
<td></td>
<td>GET scheme://[proxy_addr]/[agent_name]</td>
</tr>
<tr>
<td>Component</td>
<td>GET scheme://[host_addr]/[agent_name]/[segment_name]</td>
</tr>
<tr>
<td>Program</td>
<td>GET scheme://[proxy_addr]/[agent_name]?type={identifier}</td>
</tr>
<tr>
<td><strong>Control agent directly or via proxy</strong></td>
<td></td>
</tr>
<tr>
<td>Inject agent</td>
<td>POST scheme://[proxy_addr]/[resource]/[agent_name]</td>
</tr>
<tr>
<td>Store program</td>
<td>POST scheme://[proxy_addr]/repo/[agent_name]?type={identifier}</td>
</tr>
<tr>
<td>Delete agent</td>
<td>DELETE scheme://[host_addr]/[agent_name]</td>
</tr>
<tr>
<td></td>
<td>DELETE scheme://[proxy_addr]/[agent_name]</td>
</tr>
<tr>
<td>Delete program</td>
<td>DELETE scheme://[proxy_addr]/[agent_name]?type={identifier}</td>
</tr>
<tr>
<td><strong>Agent registration and discovery with the DRD</strong></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>POST scheme://[drd_addr]/[agent_name]</td>
</tr>
<tr>
<td>Unregister</td>
<td>DELETE http://[drd_addr]/[agent_name]</td>
</tr>
<tr>
<td>Modify</td>
<td>POST scheme://[drd_addr]/[agent_name]</td>
</tr>
<tr>
<td>Search</td>
<td>GET scheme://[drd_addr]/[agent_name]</td>
</tr>
<tr>
<td></td>
<td>GET http://[drd_addr]?attribute_name=agent_attribute</td>
</tr>
<tr>
<td>Description</td>
<td>&lt;/agent_name&gt; resource_type=&quot;rt&quot;;ct=&quot;content_type&quot;</td>
</tr>
</tbody>
</table>

required to register and unregister the agents during migration process. DRD exposes RESTful HTTP and CoAP interfaces for resource registration, search, modification and unregistration. A resource registration entry is given in CoRE Link format. It is assumed that the DRD address is globally known in the system.

The framework contains a CoRE framework specific component, a proxy (RFC 8075), operating as a reverse proxy to integrate CoRE resources to the Web and to perform required protocol translations for bi-directional interactions, possibly over disparate networks. Currently, GET, POST and DELETE methods are translated between the protocols for any resource. The proxy supports conversion between content type, which is currently provided for agent messages between JSON and CoAP. The proxy is an additional service that can be located through the DRD, but its address can be globally known, and hence requests to locate external remote resources are directed to it. When a request with a URI without address is send to the proxy, it locates the resource from the DRD and forwards the message to the corresponding host. The handling of POST requests is more advanced as the proxy is capable of creating mobile agents based on POST requests. If a POST request containing a resource URI and additional URI
element is sent to the proxy, it is assumed that the additional second part is an agent URI. The proxy then performs two searches. First, a search is conducted to the code repository to locate an agent program based on the agent URI. The second search tries to find the resources corresponding to the first part of the POST request URI. If the program and resources are found, the proxy creates a mobile agent in JSON or CoAP format containing the agent program and the resources URIs. Another use case for this functionality is mobile agent migration from CoAP network to the Web or vice versa. The agent messages are translated accordingly between JSON and CoAP, where the proxy replaces the agent program with a version that runs on the target platform, identified by its resource entry in DRD.

Code repository is a framework service that is not defined by either the FIPA or IETF CoRE. Its purpose is to store and provide at request the mobile agent programs in any programming language. Agent programmers, or clients, send the particular program to the service using the agent name and corresponding programming language identifier. Currently defined programming language identifiers are “python”, “javascript” and “intelhex”. A compiler service was implemented for the repository that is used to compile C agent programs to machine language programs in the Intel HEX format.

Regarding the RQ1, the proposed framework provides minimum FIPA services for agent discovery and interactions. The interactions between agents and other system components are realized based on the ROA principles. No other software agent framework currently provides a combination of these two standards. The defined RESTful API fulfills the requirement for uniform interface and seamlessly integrates of mobile agents as system resources into existing IoT systems. Each agent as resource is discoverable, addressable and linkable. An example of intermediate components are application-specific proxies that understand the request-reply semantics of interactions and can modify resource requests as needed that further increases interaction capabilities in the framework. These properties enable intermediate components that take advantage of the ROA approach to implement advanced features for agent operations in the future. In the framework, FIPA ACL is not considered as meaningful dialogues between agents require building mental models of the conversation parties and maintaining the state of the conversation. These capabilities are generally not available for reactive agents, but are common with deliberative agents. Also, this kind of interactions are not RESTful, as the server, i.e. resource host, should not maintain interaction state. Nevertheless, reactive mobile agent operations can be enabled in the framework with a RESTful interface, which provides an answer for the RQ1.
3.3 Mobile agent platform

Mobile agent operations in IoT require agent platforms on a heterogeneous set of resource-constrained devices. Device heterogeneity is demonstrated in this thesis on the device layer of the IoT system architecture with two different agent platforms. The first platform is targeted for low-power embedded systems as a resource-constrained IoT device. The second platform is targeted for smartphones as a mobile IoT device, which have more resources for application execution and communications but are still limited with regard to energy consumption.

An agent platform is the physical platform, where the agents exist as objects and whose lifecycle is managed by the platform software. In the software agent framework, the agent platforms are the host computers for agents. The FIPA specification for agent platforms (FIPA 2004) describes the minimum services required (Figure 6b) for an agent platform: an agent management system (AMS), a directory facilitator (DF) and a message transport service (MTS). The specification does not address internal design and architecture of the platform. Also, the platform functionality is allowed to be distributed among hosts. A mobile agent platform requires an EE to execute mobile code and a set of services for handling agent migration, agent program state capture and agent object serialization and a transport protocol. In this thesis, REST and ROA principles constitute the agent platform design principles to enable the autonomous operations and interactions of the resource-oriented mobile agents.

In the presented platform, the reactive mobile agent architecture is represented by a data structure that contains the agent components, as described in Table 2. The data structure is built from the received agent message that stores the agent in device memory, controls the agent in different stages of its lifecycle, retrieves the resources it needs and runs its program. The contents of the data structure are updated by the platform as a result of the agent program execution and interactions. As only the contents of the agent data structure are considered in migration and not its execution state as a process, only weak migration is supported in the platforms.

The lifecycle of the reactive mobile agent is defined as shown in Figure 14 as a finite state machine (FSM). The lifecycle is a simplified version of the suggested FIPA agent lifecycle (FIPA 2004) and does not consider the suspend and transmission states of an agent. Suspend state is not required as reactive agents by definition react to their perceptions and can’t initiate actions otherwise. In the platform, agents’ operations are resumed immediately after receiving the agent message and thus considered always
active. Agent transmission states are not needed as migration is transparent from the agent point-of-view and the MTS transmits the agent as any other message.

The lifecycle is the following. First, when an agent message is received, the agent is deserialized to the corresponding data structure and its state becomes IDLE. Then, the agent state becomes PREPARE while it waits for the retrieval of the local and remote resources. Once all the resources have been retrieved, the state becomes READY. The state changes to EXECUTE when the agent program is running in the platform. After execution, the state becomes STOPPED. If the agent requests migration or the migration policy requires a move, the agent is serialized into a message and sent to the next host. If the agent stays, the state is changed to PREPARE to restart the next iteration.

The two agent platforms are realized in this thesis that provide a RESTful agent API for both HTTP or CoAP servers. The API defined in the Publications I and II is shown in Table 6. The supported message types are resource retrieval, control of a resource and agent migration. The semantics of the API follows the HTTP and CoAP protocol specifications, as described in the previous Section. In addition, the server maintains descriptions of the platform resources that are exposed through the root URI with a well-known URI schema\(^\text{16}\). This is required by the CoRE framework and makes the resources of the platform discoverable.

The method GET is used by the EE to retrieve a resource representation regardless of the resource type or location. An incoming GET request is allowed for a hosted resource, e.g. sensor data or hosted agent. POST request has several use cases. It may be a request to manipulate an external resource by an agent, a request by the EE to the DRD to register a hosted agent or a request for migration of the hosted agent by an agent or the EE. For resource manipulation, the request may contain optional attributes that instruct how the request is executed at the host. Incoming POST message is allowed for

\(^{16}\text{https://tools.ietf.org/html/rfc5785}\)
Table 6. RESTful API to control and access platform resources.

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve local or remote resource representation</td>
<td></td>
</tr>
<tr>
<td>Local</td>
<td>GET scheme://[resource_name]</td>
</tr>
<tr>
<td>Remote</td>
<td>GET scheme://[host_addr]/[resource_name]</td>
</tr>
</tbody>
</table>

Control local or remote resource

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local POST</td>
<td>scheme://[resource_name]?attribute_name=[attribute]</td>
</tr>
<tr>
<td>Remote POST</td>
<td>scheme://[host_addr]/[resource_name]?attribute_name=[attribute]</td>
</tr>
<tr>
<td>Migrate POST</td>
<td>scheme://[host_address]/[agent_name]</td>
</tr>
</tbody>
</table>

manipulation of a platform resource and to receive an agent in migration. Outgoing DELETE requests by the EE are allowed to unregister an agent from the DRD, but otherwise not allowed, e.g. an agent cannot request a deletion of a resource. Incoming DELETE request is allowed for the hosted agent resource, when the request comes from a system service. The corresponding agent is deleted from the device memory and unregistered from the DRD.

The agent interactions are realized through resource requests with the GET method or by resource manipulation with the POST method, as shown in Table 6. However, as described, currently an agent cannot request a resource representation at runtime but the EE retrieves required resources for it before execution. Figure 15 illustrates the API use, as presented in Tables 5 and 6, with the agents presented in the Figure 12. AGENT1 requires remote resources (1) from NODE2 and from an integrated service WEBSERVICE. AGENT1 also uses the state of the AGENT2 as a remote resource. The AGENT2 uses a resource in NODE2 as a remote resource (3). AGENT1 does not migrate from NODE1, but the AGENT2 migrates between the platforms (3), starting from platform PHONE1.

The embedded mobile agent platform, its EE and the agent API are realized as follows. In the device RAM, a memory section is reserved as a shared memory that is accessed by both the EE and the agent program with a common set of pointers. Each resource type and the agent state have their own reserved memory locations. Initially, when an agent message is received, the corresponding agent data structure is built with the agent resource lists and the agent state is written into the shared memory. When the EE retrieves resource data, local or remote, it stores the values into the memory through the corresponding pointer in the shared memory. Once the resources have been retrieved, the agent program is written into the device program memory to a reserved memory.
Fig. 15. Agent interactions of the two example mobile agents AGENT1 and AGENT2.

location. The EE then executes the agent program through a function pointer of the reserved location. The agent program accesses the data in the shared memory through the set of pointer variables. An agent can request migration with an agent API call that is provided through a known function pointer. After the agent program finishes, the updated state is stored in the shared memory from where it can be retrieved by the EE. Now, the agent is registered into the DRD as an updated state is provided in a new host. For migration, the agent state and optional resource values are retrieved from the shared memory and with the agent data structure, compiled into an agent CoAP message. The EE modifies the local resource list according to the migration policy. After successful migration, the agent is removed from the device memory and unregistered as a resource.

The architecture of the embedded mobile agent platform is shown in Figure 16a and its operation is shown in Figure 16b.

The embedded platform supports five agents in the memory at once. This number is selected based on experiments with the presented real-world prototypes that each agent would consume about 200 bytes of device RAM, totaling one kilobyte of RAM that is
typically available in most MCUs. However, the maximum memory consumption of a number of agents is not tested in the platform due to varying resources in agents and in the platform. Each agent can be independently in any state of the lifecycle. Agent execution is synchronized based on the sensor data readings once every second, after which the agents in the ready state are executed sequentially one by one. The platform is
implemented with C for the ATmega 1284p\textsuperscript{17} MCU. In the platform, the data format is a 16-bit numerical value for the agent state, resource data and sensor data. The reason for this limitation is the hardware design of ATmega MCU that has integrated 10-bit analog-to-digital converters, whose full range of values can be represented with 16 bits. When the data value range is set, there is no need to consider different types of variables in the agent message, nor in the shared memory.

The platform is not based on any embedded OS, but implemented as device firmware. Therefore, it does not support typical embedded OS features such as application-specific events, multitasking, resource management or inter-process communication. The platform supports mobile agent program written only in the AVR machine language that are transmitted in the Intel HEX format. ATmega MCUs have self-programming capability\textsuperscript{18}, i.e. a program in the memory boot section can write into the device program memory. This feature enables runtime execution of the remote code. This solution does not require an underlying VM to run agent programs, which saves hardware resources.

The agent platform for mobile devices is implemented with Java for Android OS version 4.0 that executes agent programs in a third party scripting environment SL4A\textsuperscript{19} that is run as a service in the device. The detailed implementation of the original platform is given in (Ramalingam 2013). SL4 relies on pre-installed Interpreter components to execute scripts of each allowed language. The agent platform allows agent programs written either in Python or JavaScript. The Java class encapsulating SL4 operations communicates with the application by using Android Intents that implement a messaging service between different applications and components. Additional API method to send Intents from the executable script was implemented for the scripting environment in Ramalingam (2013).

The mobile platform architecture is shown in Figure 17a and its operation is shown in Figure 17b. Internal operation is similar with the embedded platform, except that communication is handled through a HTTP server, and agents are executed in threads. Each agent is encapsulated in its own Java agent handler class, i.e. a thread, that contains the agent data structure. Once the resource values have been retrieved, the EE builds an executable script from the data and the agent program by placing code variables containing the resource values into the script. The script is then saved as a file into the device and its information sent to the scripting environment for execution as an Intent.

\textsuperscript{17}http://www.microchip.com/wwwproducts/en/ATmega1284p
\textsuperscript{18}http://www.atmel.com/images/doc1644.pdf
\textsuperscript{19}https://github.com/damonkohler/sl4a
The agent program results are returned in an Intent posted from the script as its last instruction. By receiving this Intent, the agent handler updates the agent data structure with the agent program results.

The scripting environment is blocking and agents are executed sequentially one by one. Agent execution is synchronized through the arrival of new sensor data that is controlled by the sensor controller. The default period is once per second, but agents can
set their own sample rates for data retrieval. The HTTP server is run as an Android 
service and each request launched from the platform is handled in its own thread. The 
platform contains also a sensor controller service that controls the sensors, i.e. turns 
them on and off and sets the sample rate, and stores the sensor data into a database. The 
sensor data can be then accessed by the agents as local resources from the database. 

Regarding the FIPA AMS requirements, the minimum three services are implemented 
for the presented platforms: AMS, MTS and DF. The realized AMS contains the EE and 
the CoAP or HTTP server as MTS (Figures 16a and 17a) and manages agents throughout 
their lifecycles. The platforms do not support creation and cloning of the mobile agents 
or a persistent storage for the agents. The realization of the DF functionality is split. 
The well-known universal URI schema serves as the entry-point for resource discovery 
on the platform. A DRD peer is available as a service, although it would be possible to 
run a DRD peer in the resourceful mobile devices. Both the well-known URI and DRD 
are accessed through the uniform RESTful interface, but currently, the DF feature of 
subscribing for resource descriptions is not supported. The MTS specification FIPA 
(2004) states that basic service provisions to send and receive messages with OS specific 
functionality are sufficient implementation. CoAP or HTTP is used as the MTP. 

The presented platforms control reactive mobile agent execution based on a sim-
ple FSM. The program execution is blocking and no runtime program state capture 
mechanism is provided. Therefore, agent programs are restarted and initialized with a 
weak state after migration at each iteration. The platform design provides asynchronous 
interactions between agent and other system services through the RESTful API. In-
teractions between agents can be realized through manipulation of a resource, but 
direct communications are not supported. The reason is that direct interactions require 
cognitive capabilities for representing the other party. Technically, additional agent 
states in a FSM and non-blocking execution of agent programs are required for direct 
interactions. These features could consume significantly more memory in runtime, 
especially in the embedded platform, as the amount and types of interactions of several 
simultaneous agents could possibly be substantial. In theory, it would be possible to use 
the platform resources, e.g. shared memory or database, for agent interactions, but 
the problem is synchronization between the agents. Currently, the messages would be 
available in the next iteration of the agent execution. Moreover, the shared memory in 
the embedded platform could be considered a local tuple space (Cabri et al. 2000a), i.e. 
blackboard, but mechanisms to synchronize the contents are not considered.
The agent platforms presented in previous work commonly have more hardware resources for the agent EEs that are run atop full-fledged OS’s, in VMs and in containers attached to (Web) servers, e.g. Aversa et al. (2010), Lee et al. (2013), Fortino & Galzarano (2013), Bergenti et al. (2014), Mitrović et al. (2014), Voutilainen et al. (2016). Typically, these platforms are capable of hosting several deliberative agents and supporting full FIPA standards. Moreover, multitasking is provided in embedded OS’s (Dunkels et al. 2004, Levis et al. 2005) with eventing or in an embedded VM (Fok et al. 2009). Naturally, the Android OS provides significantly more built-in features to execute mobile agents and their interactions. However, the design goal was a minimum reactive mobile agent platform for resource-constrained embedded devices, where the initial aspiration was to develop similar functionality for both platforms. With regard to agent programming languages, JavaScript as mobile agent programming language provides seamless integration with Web technologies. However, a drawback is the execution overhead caused by the scripting environment used in the mobile platform.

REST-based APIs for agent platforms have been proposed in previous work as well, e.g. Feldmann (2007), Cucurull (2008), Braubach & Pokahr (2013), Pico-Valencia & Holgado-Terriza (2016). As discussed, these solutions are not strictly RESTful but can be considered conceptually more as non-RESTful or REST-RPC hybrid architectures. For Web integration, REST interface is considered a wrapper for agent interactions and HTTP is downgraded into a transport protocol. In WoT mobile agent platforms (Mitrović et al. 2014, Voutilainen et al. 2016), the HTTP semantics are similarly used in interactions. The agents in these platforms are implemented as HTML or HTML5 documents, but the autonomy is limited as the agent lifecycle is tied to the existence of the host Web document that are retrieved from the Web servers. Without migration autonomy, these solutions by definition belong to the CoD or REV code mobility paradigm solutions.

The mobile agent migration in the presented platforms is transparent and weak. Strong migration is difficult due to device heterogeneity as common format would be needed to transmit execution states of programs that are in different languages and run in different hardware. A solution would be a common VM as in Gray et al. (2002). As the mobile platform supports one execution thread and messaging threads are active only before the agent execution, the challenges related to multithreaded agent migration are avoided. The proposed architecture enables to transfer only the state and the resource references to reduce the transmitted agent size. Typically, in embedded devices, the program memory is significantly larger than RAM. Thus, lightweight migration would
enable the platform to host sizable agents that migrate in multiple messages. The tradeoff is that when the agent migrates as self-contained executable unit, the migration procedure is more reliable in network with intermittent connectivity. In previous work, reliability is increased with intermediate docking stations (Lee et al. 2013) to store the agents until they can reach the next host. This feature can be achieved through intermediate framework components that cache the agent message.

In conclusion, with regard to RQ1, the presented mobile agent platforms propose an answer on how to realize the operations of reactive mobile agents in resource-constrained heterogeneous IoT devices. The platforms provide a RESTful API and a minimal set of FIPA services that enables to integrate agents into the software framework.

3.4 Energy efficiency with mobile agents

RQ2 targets energy efficiency of mobile agent based IoT application execution in low-power IoT devices. In order the explore this question, two common IoT applications are considered: mobile crowdsensing with reactive mobile agents, as described in Publication III and IoT edge computing with reactive mobile agents, as described in Publication IV. Common resource-constrained devices for data acquisition in IoT applications include both mobile devices, e.g. smartphones, and stationary WSN nodes. The selected applications rely on both types of devices operating separately, but also cooperating in an application scenario. These device types have also been utilized as mobile agent platforms in the previous work, as discussed in Section 2.1.3. Thus, the selected applications provide means to validate the results with regard to previous results and to study the energy efficiency of mobile agent based application execution from multiple viewpoints. Moreover, the obtained results are compared against existing non-agent approaches in the same IoT applications. Energy efficiency is measured with real-world experiments and scaled up with simulations.

The first application deals with mobile crowdsensing implemented with mobile agent as a MAS. Mobile crowdsensing targets large-scale sensing applications with smartphones that are exploited as mobile sensors (Lane et al. 2010, Guo et al. 2015). With user mobility, smartphones provide a highly scalable participatory sensor network that benefits from the user’s contextual understanding of the application requirements. The existing crowdsensing frameworks coordinate participation centrally. Crowdsensing campaigns are largely executed with self-contained smartphone applications or with task offloading frameworks based on CoD paradigm. The collected data is then
analyzed in a cloud platform. The existing solutions that provide optimizations of the application execution and data transfer in the devices, target the particular application implementation solely. Due to the opportunistic nature of participation and network connectivity issues, in-network data processing is beneficial for application execution (Conti et al. 2015). A common factor in crowdsensing applications is that reducing the amount of transmitted data contributes towards less energy consumption, in spite of the different smartphone hardware, wireless communication technologies and environment, different applications requirements and participant behavior. Therefore it is justified in the evaluation to focus on in-network data processing.

Publication III demonstrates how mobile agents can be used in mobile crowdsensing applications atop the presented framework. The idea is that campaigns are designed as a MAS with campaign-specific agents that have tasks, rules of operation and interactions. The tasks of the agents in crowdsensing are related to initiating data collection, in-network data processing and controlling application execution in an opportunistic set of participating devices. Mobile agents are used to implement these tasks with the benefit of mobility in comparison with software agents in general. Mobile agents relocate the role execution in response to resource availability, network characteristics and participants’ actions.

The presented software framework supports autonomous operations of mobile agents in crowdsensing as follows: the mobile devices are required to register their availability as agent platform with their sensors into the DRD. A campaigner designs the mobile agents with the specific task, i.e. agent program, and defines task-based interactions through resource references. The DRD provides real-time information of available devices with the required resources that the mobile agents can use as platforms. As resource availability is fluctuating due to mobile device owner mobility and unpredictable actions, either the executing agents or the campaigner needs to periodically check the DRD to find suitable participants. Based on campaign-specific launch criteria, e.g. a set of suitable participants visit a target location, the campaigner injects mobile agents into the system. Mobile agents then operate autonomously and asynchronously in the available devices until resources are available or the campaign is finished. Due to unpredictability in the resource availability, the agents can be programmed to upload their results to the campaigner in real-time, which mitigates the possibility of lost results.

To address energy efficiency of mobile agent based crowdsensing campaign execution, a set of crowdsensing scenarios were designed and simulated with the NetLogo
Table 7. Simulated mobile crowdsensing scenarios.

<table>
<thead>
<tr>
<th>Id</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1.1</td>
<td>Data is collected in all locations and uploaded in real-time.</td>
</tr>
<tr>
<td>T1.2</td>
<td>Data is collected in previously unvisited locations and uploaded in real-time.</td>
</tr>
<tr>
<td>T2</td>
<td>Data is collected in all locations and uploaded once, when leaving the area.</td>
</tr>
<tr>
<td>T3.1</td>
<td>Mobile agents as stationary agents process and upload data in real-time.</td>
</tr>
<tr>
<td>T3.2</td>
<td>Same as T3.1, but data collected and processed only in previously unvisited locations.</td>
</tr>
<tr>
<td>T4.1</td>
<td>Mobile agents migrate to devices with data, process and upload in real-time.</td>
</tr>
<tr>
<td>T4.2</td>
<td>Same as T4.1, but data collected and processed only in previously unvisited locations.</td>
</tr>
<tr>
<td>T5</td>
<td>Mobile agents share data and results with different number of requests and requestors.</td>
</tr>
</tbody>
</table>

platform (Wilensky 1999) in Publication III. A target area was defined, where simulated participants moved and collected arbitrary sensor data according to the scenario goals. In all the scenarios, the participants behaved the same way that was initially randomly set up for the first scenario. The number of participants, up to 4500, varied as the scenarios took different time to finish. Total energy consumptions of the participants’ devices were collected during the simulation. The energy consumption parameters for the simulation were determined based on real-world experiments with the agent platform for mobile devices and as reported in Rice & Hay (2010).

The scenarios, listed in Table 7, demonstrate different aspects of mobile agent-based energy efficiency with in-network data processing, information sharing and migration. Scenarios T1 and T2 are the baseline scenarios without mobile agents that demonstrate two extremes in existing mobile sensing applications with regard to data upload to the cloud. In scenario T3, the mobile agents are stationary agents, operating in the device without migration. Scenario T4 demonstrates mobile agent migration. Scenario T5 addresses mobile agents sharing information with each other in a MAS.

The energy consumption results of the simulated scenarios are shown in Figure 18. The scenarios T1.2 and T2 correspond to the self-contained optimized crowdsensing application scenarios, demonstrating generally the lowest energy consumption. The results show that adaptive sensing, e.g. collecting data only in previously unvisited locations by any participant, contributes to energy efficiency significantly. With mobile agents, the device sensors as resources can be controlled adaptively, which requires MAS coordination. In the simulations it was assumed that when the mobile agents upload data to the campaigner, they receive information of unvisited location in their route. The mobile agents do not interact with each other in this respect, which would generate additional overhead and affect the energy consumption. Scenario T3.1 demonstrates
Fig. 18. Detailed energy consumption of the scenarios with different number of mobile agents (1-90), as reported in Publication III.

execution overhead of mobile agent as stationary agent, where the total scenario energy consumption is found significantly larger than in the baseline scenarios. This matter is discussed in detail in Publication III, where it is found out that the third party scripting environment used in the agent platform was not optimized resource-wise. However, with adaptive sensing, this overhead can be reduced, as shown in results of the scenario T3.2. Scenarios T4.1 and T4.2 demonstrate the effect of migration with an increasing number of mobile agents in the same campaign setting. The migration policy of agents was to prioritize migration into devices with most unprocessed data in ascending order. This would require MAS coordination by the agents and in the scenarios the campaigner provides this information to the agents. Again, this is done to minimize the MAS interaction overhead in this respect. With adaptive sensing in scenario T4.2, total energy consumption stays consistently at the same level as the baseline scenario T1.2, even with the agent execution and migration overhead. This happens because the task has a fixed amount of data that can be collected once. This scenario demonstrates an crowdsensing application, where mobile agents are inactive and are briefly activated in response to specific events.

Figure 19 shows the simulation results with regard to agent-based sharing of a number of raw data items and state, i.e. task results, between mobile agents in a MAS. When a Wi-Fi connection is already open, the amount of transmitted data in the range of several kilobytes has little effect on the energy consumption, as discussed in Rice &
Hay (2010). The results indicate that the number of individual requests and requestors are the most significant factors in sharing overhead, because communication energy consumption increases with the overhead of each connection. It is further observed that sharing a small amount of information with a small number of requestors (1-10) introduces up to 10% overhead on the total campaign energy consumption of scenario T4.1, as calculated from the results shown in Figure 18.

To verify the simulation results, a small-scale real-world campaign is implemented in Publication III. The real-world campaign, as shown in Figure 20, is intentionally designed with two mobile agents to demonstrate in-network sensor data processing and information sharing in real-world settings. The evaluated application is a pedestrian flock detection, as discussed in Publication III. Three different scenarios were evaluated to provide data of mobile agent based energy efficiency. First, a baseline R1 scenario addresses crowdsensing application without mobile agents. In this scenario, participating devices upload all their data to the backend. Second, two mobile agent based scenarios are evaluated. In scenario R2, mobile agents operate as stationary agents in the devices. In scenario R3, the mobile agent “wifi_vector” migrates between a set of devices and processes their sensor data. Another mobile agent “similarity” operates as a stationary agent in one device, retrieves the result of the “wifi_vector” agents from the other smartphones as remote resources, processes further the results and uploads them to the backend. The energy consumption of the campaign was evaluated with three different data upload intervals: real-time, 20s, one minute, four minutes and 15 minutes. The two intervals, real-time and 15 minutes, are selected to correspond to the simulated scenarios.
The energy consumption data in the participating devices was collected with the Powertutor\textsuperscript{20} application for Android smartphones.

The results of the real-world evaluation are shown in Figure 21 that shows total energy consumption in each scenario with different data upload intervals. The black line shows the mobile agent platform energy consumption. It is observed that the mobile agent execution overhead is quite large as demonstrated in the difference between baseline scenario R1 and the stationary scenario R2. The reason for the large overhead is again the utilized third party scripting environment. The energy consumption of the scenarios R1 and R3 stays approximately at the same level across different data upload intervals. With scenarios R2 and R3, it is observed that significantly less energy is consumed when the agent migrates into the device in infrequent intervals. These findings suggest that a method to reduce the agent execution overhead is to migrate the agent into the device infrequently, if the application requirements allow this.

Concerning the amount of transmitted data in the real-world campaigns, in-network processing is efficient in reducing the data size (Figure 21). This benefit is naturally application-dependent as different applications have different requirements for the data. Again, as presented in Rice & Hay (2010), decreasing the amount of transmitted data has little effect on the communication energy consumption but keeping the connections open has significant effect, which should be minimized. With regard to agent migration, a simple migration policy was utilized in Publication III. In this policy, the mobile agent

\textsuperscript{20}http://ziyang.eecs.umich.edu/projects/powertutor/
size is compared against the target remote resource data size. If the agent size is smaller, the agent migration is justified energy-wise. This policy was successful in reducing the migration energy consumption, but the migration decision becomes more complex when multiple remote resources are to be accessed.

The second evaluation application scenario for mobile agents deals with IoT edge computing. As discussed in Section 2.2, the REV and CoD paradigms are commonly utilized in edge computing to offload tasks to the edge layer or to the devices. Both paradigms reduce the data transmission load of the data producing devices by relocating the application execution to where the data is located. Publication IV similarly demonstrates how edge computing platforms can be extended with mobile agents to include the data producing devices at the device layer. In addition, as discussed in Section 2.2.1 and in Chess et al. (1997), Lange & Oshima (1999), Hurson et al. (2010), mobile agents have been utilized to execute various networking functions that can further optimize the system operation with regard to dynamic characteristics of networks. However, evaluation of such agent functionalities is outside of the focus of the thesis.

The idea is to demonstrate how mobile agents can optimize data transmission energy consumption in low-power IoT devices with two methods. First, the amount of
transmitted data is reduced with in-network data processing in a WSN nodes. Second, the mobile agents control autonomously the host device, i.e. WSN node, resources for the application execution. The common centralized approach to control node operations suffers from communication delays and consumes power in intermediate nodes that transmit the command to the target node in multi-hop deployments. One method to avoid these problems is software agents operating in the nodes that control its resource consumption, e.g. by controlling node sleep cycle (Tynan et al. 2005a). In Publication IV, the WSN node transceivers are exposed as resources, which allows mobile agents hosted in the device to control the transceiver duty cycling. The evident drawback is that the interactions of the nodes are limited while the transceiver is sleeping. However, in related work, this approach is commonly used with real-world WSN deployments today to reduce node energy consumption. The utilized duty cycle is predetermined and optimized for the particular application by the designer or controlled centrally by an application component, such as sink node. With mobile agents, this optimization can be done autonomously and dynamically with regard to applications’ requirements and resource availability.

For the evaluation, an example edge application is designed with two mobile agents as a hierarchical MAS that operates on low-power data producing devices. A set of mobile agents (MA1) migrates into each WSN node to process sensor data. To realize agent interactions, this data is requested from the WSN nodes by a set of another mobile agents (MA2) that operate in the mobile devices. This mobile agent processes its own sensor data, aggregates the data from different devices and uploads results to the edge. The mobile devices operate as Internet gateways for the WSN nodes. This edge application MAS, as shown in Figure 22, was evaluated in Publication IV with large-scale simulations based on real-world experimental data.

The simulation parameters were collected with Arduino Mega2560 boards as WSN nodes, running the embedded mobile agent platform. The nodes communicate with

Fig. 22. A MAS edge application based on mobile agents (MA1 and MA2), as reported in Publication IV.
with CoAP by using the XBee\textsuperscript{22} S2 transceivers. The energy consumption parameters were determined in real-world settings by using a power monitor hardware device\textsuperscript{23}. The results, as shown in Figure 23, indicate that in comparison with a single data item upload, mobile agent operation in the platform introduces about 1% overhead to the total device energy consumption. The measured energy consumption consists of agent migration into the device, agent program execution and agent uploading the result to the edge. If the mobile agent state is requested by another device, additional overhead about 1% is introduced to the energy consumption. With regard to communication energy consumption in the device, the migration, execution and upload overhead is 7% and with a remote request about 10%. As the transceiver is already on, multiple remote requests received during the online period would not similarly increase the energy consumption when the data size is kept small. As discussed in Publication IV, the XBee protocol imposes limitations, such as low throughput and small message frame size, which leads to unoptimal transceiver use for this kind of WSN applications. The hardware design is used on purpose as it corresponds to a real-world IoT system deployments, where a variety of devices commonly utilize off-the-shelf hardware components that may communicate with proprietary non-IP protocols.

The mobile devices run the agent platform and communicate with HTTP atop Wi-Fi, for which the energy consumption parameters were reused from Publication III and from Rice & Hay (2010). For the simulations, the smartphones are equipped with additional XBee transceivers, enabling them to operate as an Internet gateway for the WSN. The

\textsuperscript{22}https://www.digi.com/xbee

\textsuperscript{23}https://www.msoon.com/LabEquipment/PowerMonitor/
Table 8. Simulated edge computing scenarios.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1.1</td>
<td>All devices upload all raw data to the edge. No mobile agents are used.</td>
</tr>
<tr>
<td>E1.2</td>
<td>Same as above, WSN node transceivers 50% duty cycling.</td>
</tr>
<tr>
<td>E2</td>
<td>Mobile agent based data processing in all devices, data size reduced to 3%, WSN node transceivers 50% duty cycling.</td>
</tr>
<tr>
<td>E3.1</td>
<td>Mobile agent based data processing in all devices, data size reduced to 3%, WSN node transceivers 50% duty cycling.</td>
</tr>
<tr>
<td>E3.2</td>
<td>Same as above, but data size not reduced.</td>
</tr>
<tr>
<td>E4</td>
<td>Mobile agent based data processing in all devices, processed data in WSN nodes are retrieved remotely by agents in mobile devices.</td>
</tr>
</tbody>
</table>

XBee transceivers then introduce corresponding overhead into the energy consumption of the devices.

To evaluate mobile agents for optimizing edge application energy consumption, large scale simulations were conducted with the NetLogo platform. Table 8 shows the simulated scenarios. Each scenario was designed with 100 WSN nodes and 20 mobile devices. First, scenario E1.1 is a baseline scenario, where all devices upload all data to the edge without optimizations. Scenario E1.2 is the same except 50% duty cycling is used with the XBee transceiver. This cycle comes from the fact that XBee radios need about 5s to reconnect to the WSN to send data. Hence, 50% cycle is achieved with 5s sleep period and 5s reconnecting period, allowing 10s data upload interval. Scenario E2 demonstrates the effect of mobile agent based in-network data processing without duty cycling. Scenarios E3.1 and E3.2 study the combined effect of mobile agent based in-network data processing and duty cycling as an example of agent-based control of the device operation. In scenario E3.1, data size is reduced to 3% of the original by the agent-based data processing. This value is a selected parameter for the simulation that demonstrates the extreme case, where almost all data are discarded by agent-based in-network data processing. The scenario E3.2 demonstrates execution of a mobile agent task, that does not reduce the transmitted data size at all, but uses the same duty cycling as in scenario E1.2. Therefore, scenario E3.2, indirectly shows the mobile agent execution overhead when compared with scenario E1.2. Scenario E4 evaluates the described MAS with two mobile agents, as shown in Figure 22.

The simulation results are shown in Figure 24. Scenarios E1.1 and E2 show how mobile agent based in-network processing reduces the total communication energy consumption 15%, when the transmitted data size is reduced to 3% of the original. When comparing the scenarios E1.1 and E1.2, it is observed that duty cycling reduces the
energy consumption 55% even when transmitting all data. It appears that duty cycling is an efficient mechanism to control energy consumption when the XBee transceiver can be kept on the minimal amount of time. The results of scenario E3.1 show the effect of combined in-network data processing, again reducing the data size to 3% of the original, and 50% duty cycling. The total energy consumption is reduced 66% of the original energy consumption in scenario E1.1. The result of scenario E3.2 shows the agent execution overhead being about 1% in the total energy consumption, when compared with the result of scenario E1.2. Thus, the two extremes of in-network data processing are shown in the results of scenarios E3.1 and E3.2. Any agent task based data size reduction falls between these extremes in this simulation setup. The results of scenario E4 show the overhead of the MAS operation, i.e. data sharing between agents, combined with the effect of using a second transceiver in the mobile devices. In comparison with the baseline scenario E1.1, still up to 51% reduction in total energy consumption is observed. This is due to the significantly reduced WSN node energy consumption, even when the mobile device energy consumption increases. Scenario E3.1. demonstrates the effect of similar data size reduction, thus the comparison of these results shows the overhead of data sharing with the second transceiver in the mobile device. The WSN node energy consumption stays about the same, as data upload to the edge is replaced with a remote request in this scenario.

These results are supported by previous research in mobile agent based energy efficiency in the context of WSN, mobile computing and Web services, as reported in Straßer & Schwehm (1997), Bandyopadhyay & Paul (1999), Hagimont & Ismail
Mobile agents are a feasible solution when a significant amount of data is to be transferred, as generally communication energy consumption and data transfer time are reduced. The available network bandwidth affects this result, e.g. with limited bandwidth the benefit of in-network data processing is amplified. However, also as shown in the results of this work, mobile agent execution and migration overheads needs to be taken into account. Related to the duty cycling in WSN, with less data to transmit, the less time the transceiver needs to be kept on (Qi et al. 2001). However, in many of the reported results, mobile agents were studied with simulations or executed in a resource-rich desktop PCs of that time.

The results acquired in the simulations and real-world experiments with the two IoT applications provide empirical evidence for RQ2 as follows. The proposed reactive mobile agents are able to operate on heterogeneous low-power and resource-constrained IoT devices. The agent operations are justified due to saved communication energy consumption with in-network processing and controlling the device operation. When the device resources, e.g. sensors and transceiver, can be exposed for the agents, adaptive sensing methods can be implemented in an autonomous and decentralized way with mobile agents. These methods are commonly utilized today in an application-specific way in real-world WSN deployments. However, this approach leads to reduced interaction capabilities outside the application in question. The results in low-power devices show that mobile agent migration overhead is comparable to sending and receiving a single small-sized message. Sharing of data and results for agent-based interactions in a MAS introduce insignificant overhead into the total application energy consumption when the number of interactions is kept small.

The energy consumption of individual heterogeneous devices varies significantly, based on application requirements, selected hardware components and battery state, e.g. Pathak et al. (2012). With mobile agents, these devices are utilized opportunistically and agents can optimize the particular application execution with regard to the available resources in the device and in the local environment. The evaluation shows that overhead introduced by agent execution in a platform can be addressed with different designs of agents and their interactions in a MAS. As a conclusion, to answer RQ2, the presented results show that mobile agent based distributed IoT application execution in low-power devices is beneficial with regard to energy efficiency.
4 Discussion

This thesis examines mobile agents in the context of IoT. The research contributions are reflected in this Chapter. Section 4.1 summarizes and analyses the presented results from Publications I to IV. Section 4.2 discusses generalization and limitations of the results. Lastly, Section 4.3 addresses the significance of the results and future work.

4.1 Analysis of the results

With regard to RQ1, i.e. to enable mobile agents in a heterogeneous set of resource-constrained IoT devices, Publications I and II presented two research contributions. First, a resource-oriented mobile agent architecture abstracts the agent and its components as system resources. This enables to utilize the mobile agents, alike any other system resource, for that matter, through a uniform RESTful interface. The second contribution is a ROA-based mobile agent software framework that relies on two existing standards: FIPA MAS abstract architecture and IETF CoRE framework. The framework realizes a RESTful API, based on the universal HTTP and CoAP semantics, that supports agent interactions and straightforward IoT application development with existing Web technologies. As outlined in Section 1.2, the thesis does not critically examine these underlying technologies and their analysis is omitted. Instead, the focus is on how to enable mobile agent operations in IoT using these technologies.

Due to the REST principles and operational semantics of HTTP and CoAP, in this thesis, the mobile agent architecture is defined for reactive mobile agents solely. The presented architecture relies on the sensors and actuators of the host device through resource abstraction and a RESTful interface. These resources enable the reactive agent to observe and act in its environment. Furthermore, the agent’s resources are categorized into different types, which enables runtime decisions on how to utilize the resources to optimize the agent operations. The operations of a reactive agent are also straightforward to implement for resource-constrained IoT devices that are the presumed agent platforms in this thesis. It should be noted that the architecture presented is platform-, OS- and programming language-independent. Two representations of the reactive mobile agent architecture were shown with common data formats: a JSON document and a CoAP message. The compliance with common standardized message formats is beneficial for interoperability. JSON is a native format for Web technologies.
The CoAP message format, in its turn, enables agent operations in resource-constrained IoT devices such as 8-bit Arduino boards, as an example of CoRE class 0 devices. Due to small protocol overhead, CoAP format is also beneficial for operations in a challenging wireless environments, as it possible to transmit the full agent in a single message and avoid message fragmentation for reliable communications. However, the communicating devices must also understand the non-standard CoAP message options that describe the agent architecture.

The presented mobile agent software framework provides required minimum services for agents, as stated in the FIPA MAS abstract architecture specifications. These services are implemented using the IETF CoRE framework components. The CoRE resource directory encapsulates several FIPA directories as resources, which simplifies their use with the RESTful interface. The Code Repository service for the agent programs is a useful addition to support migration between heterogeneous platforms and to reduce the transmitted agent size with references to its components. The CoRE framework introduces intermediate components, such as proxies, that provide a standardized way to integrate systems over disparate networks and protocols. Proxies are a necessity for large-scale IoT systems today with devices relying on heterogeneous communication technologies. These software components are straightforward to extend with new services for agents, even when the agent platforms have limitations. For example, proxies can be used to implement mobile agent home servers with the additional benefit that they already interconnect disparate network parts.

Embedded Web services are a feasible approach for Web integration of resource-constrained IoT devices. With the resource abstraction and the presented RESTful API, reactive mobile agents, IoT devices, system services and existing Web services can interoperate seamlessly. In the API, widely known HTTP and CoAP semantics are extended for mobile agent operations. These request-reply interaction capabilities are sufficient for reactive agents to operate in the dynamic IoT environments, where short-lived ad hoc interactions are an effective way of communication. The REST principles dictate stateless communication, which means that interactions as requests are self-contained. The application and interaction state are kept on the client side and only the resource state is maintained by the server. An agent can choose through its program execution and current state how a request is answered. The resource abstractions provide different ways to interact in an environment, e.g. to actuate a remote resource, but more complex interactions would require cognitive capabilities for the agents. The benefit of SOA and WS-* for agent platforms is that interactions are explicitly defined and
it is guaranteed that both parties agree with the interaction protocol. In this respect, ROA-based frameworks need common ontologies to define the semantics of agent interactions. In the end, the operational semantics of RESTful interactions are found simpler and less resource consuming than in the SOA and WS-* approaches in the related work. Generally, this means that less computation and communication resources are needed in the interacting devices and in network infrastructure. These features facilitate straightforward implementation of Web-integrated distributed IoT applications.

Full FIPA-compliancy is challenging due to constrains in the presumed agent platforms. The main difference between the presented agent framework and the FIPA-compliant agent frameworks is lack of FIPA ACL and related ontology support. The agent interaction semantics are fundamentally different with HTTP and CoAP in comparison with the rich vocabulary of the FIPA ACL. However, as discussed, the presented approach is feasible for reactive agents where interactions are facilitated through the resource abstractions. The presented mobile agent lifecycle is a subset of the FIPA agent lifecycle, but enables reactive mobile agent operations that are sufficient in this thesis. The FIPA specifications state that the agent lifecycle can vary. Moreover, the FIPA specifications provide a set of elements and attributes for agents and framework services that were not considered in the presented minimum implementation. In comparison with the related work, the Java-based agent platforms typically implement full FIPA specifications for the agents, where it is expected that the agents are deliberative agents. The Java-based agent architectures address device heterogeneity with reusable programming abstractions, i.e. Java classes for agents and interactions, and a common bytecode format. However, the existing Java-based architectures and platforms do not similarly address resource limitations and resource-constrained embedded devices, Java object execution and migration overheads are too high due to hardware limitations.

IoT device heterogeneity is further addressed by implementation of two platforms for reactive mobile agents: the embedded agent platform and the agent platform for mobile devices. The platforms provide agent execution management and a minimum set of FIPA specified services for the agents through the CoRE framework components. The available resources in the embedded devices limit the agent operations, whereas the mobile platform has significantly more resources. The platforms support weak migration as capturing and resuming a process state would be difficult in a set of heterogeneous platforms that do not execute agents with a common VM. Another limitation is the energy saving operations in these devices, e.g. duty cycling and adaptive sensor control, imposed by the device, OS or application designer. These features are justified to
prolong the device lifetime and are therefore commonly imposed, regardless of whether the application is agent-based or who is given the control of these components. With Web services, it is commonly assumed that a Web server is always on and requests are answered. However, this requirement is relaxed with embedded Web services.

The embedded agent platform was implemented for ATmega MCU family, e.g. Arduino boards, which support self-programming that is a crucial feature for utilizing mobile code in runtime. Due to this reason, the mobile agent program needs to be precompiled into the MCU native machine language format. This is an efficient solution to execute the agent program as no interpretation step is needed as in VM-based platforms. But, this solution also results in a larger agent program than a program written with an optimized VM instruction set. A compiler system service in the framework is feasible for automatic creation of agent programs targeted for embedded devices. The embedded platform does not utilize an embedded OS that would support eventing, multi-tasking, resource management nor inter-process communication. The agent platform for mobile devices is an Android application that can exploit a full set of OS features, such as multithreading. This platform supports agent programs written with common scripting languages, but attention must be paid to the agent execution overhead as well. Common scripting languages for the agent programs are beneficial for integration with existing systems, such as the Web.

This thesis provides as answer for RQ1 in Publications I and II with the presented reactive mobile agent architecture, the software framework and the implemented agent platforms that enable reactive mobile agents’ operations on a heterogeneous set of resource-constrained IoT devices. No other mobile agent platform today, as of September 2017, outside the presented work, is ROA-based nor provide means to build agent-based applications with embedded Web services.

The RQ2 considers the energy efficiency of reactive mobile agents for application execution in low-power IoT devices. Publications III and IV provide evaluations with real-world experiments and large-scale simulations. As a realistic IoT application scenario, Publication IV presented a system deployment, where WSN nodes operate with a non-IP proprietary radio that requires a gateway to connect to the Internet. Energy efficient operation is the norm in such resource-constrained IoT devices and therefore justified as an evaluation approach. Energy efficiency is addressed by comparing operations of mobile agents in a MAS in the data producing devices with straightforward data upload to the backend system. This comparison shows the benefits of reactive mobile agents in the selected IoT applications.
The evaluation results collectively show that reactive mobile agents can improve energy efficiency in the low-power IoT devices with two methods for adaptive sensing: in-network data processing and controlling the device operation via duty cycling. Although duty cycling is a common solution to optimize application-specific energy consumption of WSN deployments, this feature is problematic with regard to the device interactions. Here, the agents can autonomously negotiate the shared resource optimization. The results show that the overhead caused by reactive mobile agent interactions in a MAS is insignificant even for the resource-constrained devices. However, as noted in Publications III and IV, the same application can be designed in a number of ways with agents in a MAS. Optimization of the resource usage becomes a MAS and application design issue, i.e. what are the roles, tasks and interactions of the agents. Sharing of data and results in a MAS increase the utility of the data, which is an important consideration in distributing application execution in dynamic environments opportunistically. However, in IoT, different interaction overheads are amplified by lack of resources in the participating devices. With appropriate mobile agent and MAS design, challenges in the agent operation, such as execution overhead or system resource access, can be addressed.

Regarding the selected agent simulation platform, the following can be said. NetLogo is a general purpose agent platform and, according to Allan (2010), specifically suitable simulation platform for mobile agents that interact locally over short periods of time. This corresponds well to the expected operational environment with resource-constrained devices in IoT. Additionally, NetLogo provides high-level primitives and structures for agent programming and also turtle-based visualization of agents and their interactions. Today, Logo is an old educational programming language, but several agent platforms still use it as the programming language (Allan 2010). Logo simplifies the simulation programming in comparison with, for example, XML-based agent platforms.

With regard to simulating the real-world networking environment, advanced network simulators facilitate modelling of the complex and dynamic characteristics of the environment. Even with a substantial programming effort, that takes into account different wireless communication technologies, conclusive evaluation of the benefits of mobile agents would be challenging as all the parameters affecting energy consumption should be considered. This includes device hardware, application requirements and operating environment characteristics that in IoT are numerous, vary significantly and not practicable to consider in their entirety. However, as presented, the simulation results in this thesis are based on parameters collected in real-world experiments.
This thesis provides an answer for the RQ2 with the results presented in Publications III and IV. The results demonstrate that distributed IoT application design and implementation with reactive mobile agents improve energy efficiency with in-network data processing, agent migration and controlling the device resources, in comparison with the traditional approaches in the same IoT application execution. These results also support the existing mobile agent evaluation results presented in the literature, but are unique in the context of embedded Web services for IoT.

4.2 Limitations and generalization

As presented in the previous Sections, the proposed approach for reactive mobile agent architecture and software framework follows REST and ROA principles and implements a set of FIPA specifications. This contribution enables a standardized way to integrate these agents into IoT. The evaluated real-world applications demonstrate energy efficiency in distributed application execution with reactive mobile agents that enable both vertical and horizontal interactions at the IoT device layer. A limitation is that deliberative agents are not considered in the thesis due to resource limitations in the presented agent platforms.

Code mobility is no longer considered “exotic” (Picco et al. 2014), as CoD and REV paradigms are commonly used today. Smartphone applications are commonly downloaded from application stores and self-contained applications are dynamically transferred in networks as VMs. The benefit of CoD is dynamic applications created by dynamically linked modules (Carzaniga et al. 2007). The benefit of REV is ad hoc applications, where the service functionality is provided by the moved code (Carzaniga et al. 2007). Arguably, CoD and REV paradigms have fewer issues than mobile agents with stricter management and reduced infrastructure requirements (Vigna 2004). Mobile agents and their interactions may be difficult to design, develop and test in open dynamic environments (Vigna 2004, Carzaniga et al. 2007). A limiting factor is that only a subset of system resources is available for agents (Vigna 2004), but it is also possible to develop systems with “too many” agents (Wooldridge & Jennings 1998). The operational overhead of autonomous mobile agents is difficult to predict and analyze due to dynamism of the environment, where reliability and fault tolerance are crucial concerns. Another challenge is security, where agents and agent platforms need to be able to trust each other. However, mobile code security in general is a complex issue. CoD and REV operations are controlled by an application component, whereas mobile
agents emphasize autonomous and asynchronous operation with the help of system services and interaction protocols.

Wooldridge & Jennings (1998) discuss the pitfalls of agent-oriented development. Agent systems are more complex than distributed systems that are also far from solved. Agent frameworks should be built on existing technologies as much as possible. The focus should not be on developing increasingly complex AI aspects, while forgetting general issues of distributed systems. This is particularly observed for open and dynamic IoT, as discussed in this thesis. Related to AI in general, the two opposites are agents that have “too much AI” capabilities for the particular set of problems and agents that have “no AI” at all and thus cannot be justified (Wooldridge & Jennings 1998). Importantly, the reasons for developing a new agent architecture need to be well-justified (Wooldridge & Jennings 1998). Why is a new architecture needed? Is it possible to adopt existing off-the-shelf solution instead? Is the solution novel and usable? In this thesis, the selected approach integrated existing well-known technologies with real-world challenges identified for distributed IoT systems and architectures. An answer for these questions was provided with a novel ROA-based mobile agent framework and a resource-oriented reactive mobile agent architecture implementation that rely on existing standardized technologies. Lastly, real-world evaluations confirmed the benefits of the presented mobile agent technology in this context.

The proposed mobile agent architecture and software framework are based on the requirement to support heterogeneous low-power resource-constrained things that operate in an open and dynamic environment. The architecture supports reactive mobile agents that can be minimized in size and whose interactions are ad hoc and short-lived to optimize communications. Limited resources in hosting devices reflect to the functionality offered by the agent platforms and only a minimum subset of FIPA specified features are implemented. Considering the RESTful agent API, stateless HTTP and CoAP semantics are largely not compatible with stateful FIPA ACL (Poslad 2007), which limits the possibilities for agent interactions. However, the ROA based architecture does not impose limitations related to the implementation of agent programs nor interactions through the resource abstraction.

To generalize the contributions of this thesis, the following can be said. REST and ROA combined are today a universal paradigm to build Web-integrated IoT applications and services. The lightweight Web protocols and CoRE framework components provide interoperability for the resource-constrained things with existing Web resources, applications and services. On the other hand, the FIPA agent standards
support integration and interoperability of agents in an open system (Poslad 2007). The combination of these standards provides a well-defined software framework in IoT for reactive mobile agents that focuses on interoperability. The presented RESTful API follows standardized Web technologies. The purpose of the API is to facilitate building distributed applications that utilize mobile agents as software components in heterogeneous set of things. The platforms for reactive mobile agents are developed for Android smartphones and Arduino embedded boards that are today well-known commercial and open device brands.

The evaluation results give empirical evidence of the benefits of reactive mobile agents in real-world distributed IoT applications that rely on data collection. Mobile agents have been previously utilized in a variety of distributed applications, as discussed for example in Hurson et al. (2010). Within the scope of a thesis, it is not practicable to consider them in their entirety, but to focus on selected aspects in the evaluation applications. Autonomy and energy efficiency are important considerations as things are commonly deployed pervasively in the environment, often at inaccessible locations. When evaluating software agent systems, a pitfall is to confuse simulation results with more complex real-world environments (Wooldridge & Jennings 1998). This thesis aims to avoid this very issue as the simulation parameters are collected with real-world experiments and simulations verified in real-world settings. Although, the parametrization of the environment has room for improvement as only the energy consumption of execution of a small-sized program and straightforward data transmission are considered from the point of view of resource-constrained devices. As discussed, the benefits of mobile agents in real-world are a sum of several factors related to available resources in the host devices and the dynamic characteristics of the environment.

4.3 Significance of the results and future work

“\textit{The Internet of Things is not just a barcode on steroids. It has the potential to change the world, just as the Internet did. Maybe even more so.}”

Kevin Ashton, 1999

Both AI and the IoT are seen growing technology trends for the foreseeable future, as predicted by Gartner\textsuperscript{24}, for example. The benefits of AI are seen on transformation from standalone IoT devices towards collaboration of interactive things. Software

\textsuperscript{24}\url{http://www.gartner.com/newsroom/id/3482617}

108
agents in IoT expand the capabilities of things and software components in all layers of IoT system architectures towards autonomous intelligent behavior (Uckelmann et al. 2011, Miorandi et al. 2012, Fortino 2016). In addition, the benefits of agent-oriented application development have been well acknowledged. This research field is starting to gain momentum with an increasing number of scientific publications that target software agents in IoT edge computing, as an example.

From the early days on, it has been difficult to find the “killer application” for mobile agents (Chess et al. 1997, Lange & Oshima 1999, Carzaniga et al. 2007, Picco et al. 2014). The argument is that there is no application that cannot be built without mobile agents. In other words, everything that can be implemented with mobile agents can be implemented with other technologies as well. Therefore, the research field has focused on finding application domains to where mobile agents as a distinct technology bring benefits. Individual benefits of distributed applications are plausibly addressed better by optimized solutions, but an open and secure mobile agent framework could possibly address several concerns at once. Cao & Das (2012) suggests that mobile agents should be used as a complementary technology enhancing distributed applications. As discussed, in the context of IoT, this means enabling interoperability that is a step forward from the application silos.

The results presented in Publications I and II are unique. Currently, no other work addresses mobile agents in REST based software framework that complies with FIPA specifications and standardized IoT software framework, where low-power resource-constrained things are considered as the agent hosts. The results for reactive mobile agent based IoT applications presented in Publications III and IV are topical. Mobile crowdsensing is one of the principal data collection methods in IoT that rely on human-machine interactions, where mobile agents converge the unpredictable behavior of humans with application requirements. Edge computing is an emerging paradigm in IoT to leverage application resources into the close proximity of end devices, where reactive mobile agents aim to optimize the application-specific data collection with the available resources, as presented in Publication IV. In both application areas, energy efficiency of the data producing devices is a significant concern for which mobile agents provide a solution.

The developed mobile agent software framework enables to further study the proposed reactive mobile agent architecture towards the capabilities deliberative agents and richer interactions and verify the obtained results in real-world settings. Another use case for the results presented in this thesis is distributed mobile agent-based IoT
applications in the real-world open and dynamic environments. Particularly, a baseline has been established for mobile crowdsensing applications in the intersection of human behavior and software intelligence. For data collection in IoT edge applications, a cross-layer MAS based on mobile agents opens up the vertical architectures for large-scale optimization of operational and environmental issues.
5 Conclusions

The IoT vision proposes a global network for distributed systems in an unforeseen scale. IoT systems rely on interconnected data producing things, services and applications. It is anticipated that IoT systems become so large that centralized control of system operation is no longer feasible. Therefore, novel approaches to distributed application control and execution into different layers of IoT system architectures have been proposed. The goal is to decentralize control in a way that dynamic resource availability and environment characteristics are in real-time used to guide and optimize application execution. This requires that the predominant model of centralized vertical IoT applications built onto silos are extended with capabilities for horizontal interactions and with methods for opportunistic participation.

In this context, reactive mobile agents are studied in this thesis to implement distributed IoT applications and to explore how those applications can benefit from both vertical and horizontal interactions. This thesis examined the use of REST and ROA architectural principles to provide a mobile agent software framework with existing standardization efforts, the IETF CoRE framework and FIPA abstract MAS architecture, to extend its usability. The framework provides a RESTful API and system components to build opportunistic agent-based applications in the open and dynamic IoT environments. In addition, a resource-oriented reactive mobile agent architecture was presented that enables RESTful interactions for the agent and other system components within the framework. Analysis was conducted to evaluate the architecture and the framework in real-world settings. The results indicate that the presented solutions successfully enable reactive mobile agent operations in a set of resource-constrained low-power IoT devices.

To provide empirical evidence of the benefits of reactive mobile agents for IoT application implementation, large-scale simulations and real-world experiments with the framework were conducted. The target applications were mobile crowdsensing and edge computing that are topical examples of distributed IoT applications. The results show that mobile agent technology can increase the energy efficiency with distributed execution of these applications, while operating autonomously and asynchronously in a set of low-power resource-constrained IoT devices.
The presented architecture, framework and obtained real-world results are a valuable addition to the active theoretical studies in the field of software agents and MAS. This thesis contributes towards the utilization of a classical AI paradigm, mobile agents, for distributed application design and implementation. As such, this area is today a field of increasing importance in IoT.
References


Mitrović D, Ivanović M & Badića C (2014) Delivering the multiagent technology to end-users through the web. Proc. 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14), Article No. 54.


Original publications


Reprinted with permission from IEEE (I), Springer (II, III) and IOS Press (IV).

Original publications are not included in the electronic version of the dissertation.
630. Palosaari, Jaakko (2017) Energy harvesting from walking using piezoelectric cymbal and diaphragm type structures
635. Ylimäki, Markus (2017) Methods for image-based 3-D modeling using color and depth cameras
641. Vuokila, Ari (2017) CFD modeling of auxiliary fuel injections in the blast furnace tuyere-raceway area
642. Vallivaara, Ilari (2018) Simultaneous localization and mapping using the indoor magnetic field
644. Pramila, Anu (2018) Reading watermarks with a camera phone from printed images
Teemu Leppänen

RESOURCE-ORIENTED MOBILE AGENT AND SOFTWARE FRAMEWORK FOR THE INTERNET OF THINGS

UNIVERSITY OF OULU GRADUATE SCHOOL, UNIVERSITY OF OULU, FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING