Self-adaptation for Internet of Things applications

Francisco Javier Acosta Padilla

ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale MATISSE

présentée par

Francisco Javier Acosta Padilla

Préparée à l'unité de recherche IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

Self-adaptation for Internet of Things applications

Thèse soutenue à Rennes le 12 décembre 2016

devant le jury composé de :

Emmanuel BACCELLI
Chargé de recherche à INRIA Saclay/ *Examinateur*

Isabelle BORNE
Professeur de l'UBS/ *Examinatrice*

Johann BOURCIER
Maître de conférénces de l'Université de Rennes 1 / *Co-directeur de thèse*

Didier DONSEZ
Professeur de l'Université de Grenoble 1/ *Rapporteur*

Stéphane FRÉNOT
Professeur de l'INSA Lyon/ *Rapporteur*

Frédéric WEIS
Maître de conférénces de l'Université de Rennes 1 / *Directeur de thèse*

A Mis padres

Francisco Javier Acosta Saludado y Alma Rosa Padilla Gutiérrez

This work was partly funded by CONACyT, to whom I thank for their support and assistance during these 3 years.

Acknowledgements

And here we are... at the beginning, but at the end...

After all these years, I have finally the space to thank all those persons who were with me during this adventure. First of all, I would like to thank the jury members for reviewing the thesis, your comments helped me to achieve this work.

I would like also to thank my advisers, Frédéric Weis and Johann Bourcier, who for all these 3 years supported me as much as possible, even when things seemed to be very complicated to solve, they were there regardless of our own expertise (ELF loader, remember?).

Thanks too to my current team at Inria Saclay and Freie Universität, you were also very helpful and important for me this last year.

Thanks to Emmanuel Baccelli, for understanding and giving me the time and advises to finish this work.

I will also thank all the members of the DIVERSE team (formerly TRISKELL on my first year), who were also very helpful on several aspects, but especially to teach me a lot about informatics. Through all the coffees, seminars and office talks, I learned a lot about you, about integrate myself as a foreign in this very open community which welcomed me with open arms. I will thank particularly Jean-Émile Dartois and Inti González for their very very precious help, without you guys, this thesis could not be achieved, THANK YOU!

But DIVERSE was not only about work, I must say that it was there where I met the most important and close people who were for me a primal professional support, but also an invaluable personal support. Thank you again Jean-Émile, for being the very first person who welcomed me and introduce me to the team, for teaching me about modelling and informatics, to giving me your friendship. I'd like also to thank Erwan Daubert, who was always open and available to discuss (with a lot of patience) to an informatics ignorant, who gave me lots of hints and knowledge about the business, but also offered me his friendship. Thanks to Jose, Mai, Sana, Francisco, Inti, Ivan, Mauricio, Bosco, Marco, Natalia, Suresh, Aleks, Yannis, Dionysos, Ilias, Rima, Simon and all those who offered me their friendship, which was essential for me at all times. I'll always remember those barbecues, football matches, 4 esquinas matches (Marce! I don't forget you!), and... yes, lots of parties!

Special thanks to the Dr. Walter Rudametkin and his beloved wife, for giving me their outstanding support. Marcia, Walter, ¡los quiero mucho!

Thanks to my family, who were always there in the good and the bad times, even from far, I can feel your support.

Merci Maude!

Merci Céline!

Merci Élodie!

Thanks to all my Mexican friends in Rennes! (Special mention to Raíces Mexicanas... Sí señor!)

Danke schön mein Schatz!

Finally, thanks to all of you, you know who you are, you know how you helped me, and you know that everyone of you has a special place in my heart.

MERCI! GRACIAS! DANKE! THANK YOU!

Résumé en français

Nos vies quotidiennes sont entourés par de nombreux dispositifs qui embarquent des unitées centrales de traitement (CPU) et une capacitée de stockage d'informations (mémoire). Ces appareils jouent un rôle important dans nos activités quotidiennes, tels que les téléphones intelligents [90], car nous les utilisons pour nous aider à accomplir de nombreuses tâches (obtenir l'information de transport public en temps réel, météo, e-mail, messagerie instantanée, etc.). En outre, d'autres types d'appareils prennent part de ce nouveau monde numérique, car ils sont introduits sous forme de *"objets connectés"*, qui sont destinées à communiquer non seulement avec les personnes par le biais des interfaces homme-machine, mais aussi à envoyer et recevoir directement des informations provenant d'autres objets similaires. En effet, les moyens de communication ont évolué depuis l'introduction de l'accès Internet aux appareils autres que des ordinateurs, en utilisant la même infrastructure et protocoles. Ce nouveau accès à Internet a apporté des nouvelles possibilités pour les dispositifs capables d'exécuter le protocole Internet standard (IP). Cependant, ces nouveaux objets connectés ne sont souvent pas en mesure de mettre en œuvre ce moyen standard de communication, en raison de leurs contraintes en puissance du processeur et de mémoire, confiant leur moyen de communication aux dispositifs plus sophistiqués (*i.e.* téléphones intelligents, des tablettes ou des ordinateurs personnels) visant un accès fiable à Internet.

L'***Internet des Objets*** (IdO) vise à fournir une connectivité Internet à ces dispositifs à ressources limitées, en introduisant des protocoles plus efficaces tout en restant interopérable avec les protocoles internet actuels. En effet, ces nouvelles capacités de communication rendent ces objets accessibles de partout, maintenant capables de fournir des services du type web. Par conséquent, les objets faisant partie de l'IdO sont destinés à offrir des services web d'une manière standard, en tirant parti des API tels que REST [40]. Cependant, très peu de services sur Internet sont destinés à être statiques. Au contraire, ils sont très dynamiques et ont tendance à évoluer très rapidement en fonction des besoins de l'utilisateur. Cela répresente un grand défi dans l'utilisation de dispositifs contraints dans cet environnement, car ils ont été conçus pour exécuter des systèmes d'exploitation et des applications qui ne sont pas capables de fournir des caractéristiques dynamiques. Ainsi, des nouveaux défis dans la recherche sur la façon dont ces objets peuvent s'adapter aux nouvelles exigences apparaissent. En fait, l'adaptation dynamique des systémes logiciels est un domaine de recherche bien connu, et plusieurs œuvres proposent des approches différentes pour fournir des solutions à ce problème. Cependant, la plupart de ces solutions sont destinées aux machines non contraintes telles

que les serveurs et plates-formes de cloud computing.

Les principales différences peuvent être reconnues comme suit :

- **Mémoire :** Kilo-octets au lieu de Giga-octets,
- **Énergie :** Miliwatts au lieu de Watts et
- **CPU :** Megahertz au lieu de Gigahertz.

Motivations

Ce travail de recherche vise ṗroposer un nouveau moyen de rendre possible le comportement dynamique et des capacités d'auto-adaptation dans les dispositifs de l'IdO de bord (*class 2* [11] au maximum), en tenant compte de leurs ressources très limitées de mémoire et d'autonomie énergétique . Les motivations de cette thèse se fondent dans la nécessitée réelle de ces dispositifs d'être hautement adaptables, puisque leur utilisation peut varier de façon importante dans un court laps de temps. En effet, plusieurs domaines ont besoin du logiciel qui peut être modifié en fonction de divers facteurs externes, tels que les *Espaces intelligents* (villes intelligentes, bâtiments, maisons, voitures, etc.). Ces environnements sont soumis au comportement humain, donc les exigences sur les informations nécessaires à partir d'objets tels que des capteurs, et les actions effectuées par des actionneurs, peuvent varier très rapidement. Ainsi, le logiciel en cours d'exécution sur ces appareils doit être facilement modifiable, sans avoir besoin d'une intervention physique tels que le changement manuel de firmware ou le remplacement de l'appareil.

Le coût en temps et en efforts pour changer le logiciel en cours d'exécution dans les appareils faisant partie d'un espace intelligent peut être très élevé, et devrait être réduit puisque les estimations sur la croissance des appareils dans l'IdO dans ces environnements est exponentielle, ce qui rend impossible d'adapter manuellement chaque appareil. Par conséquent, de nouvelles approches fournissant des mécanismes de déploiement dynamique et automatique dans les environnements de l'IdO sont d'un grand intérêt. En effet, ces efforts peuvent contribuer à la création d'une infrastructure IdO envisageable, ce qui porte à une utilisation plus efficace de l'énergie pour les activités humaines, ainsi que d'un mode de vie plus confortable.

Challenges

Cette thèse propose l'adoption des approches génie logiciel, et plus spécifiquement, de l'ingénierie dirigé par les modèles telles que les modèles en temps d'exécution [77] pour gérér la très grande couche logicielle présente dans un environnement IdO, en tenant compte des ressources limitées et l'autonomie énergétique typique des dispositifs de l'IdO.

Les approches existantes provenant de la communauté du génie logiciel pour gérer des grandes plateformes logicielles distribuées et hétérogènes, sont destinés à leur utilisation sur des ordinateurs et des serveurs puissants, qui ne sont pas au courant de l'utilisation de la mémoire et de la puissance de traitement nécessaire pour faire fonctionner leurs implémentations.

En effet, l'IdO peut être considéré comme une plateforme logiciel distribuée, donc ces approches de gestion sont à considérer pour leur adoption et mise en œuvre sur les appareils de l'IdO. Cependant, la nature de ces dispositifs composant l'IdO diffère extrêmement des machines sur lesquelles ces approches sont normalement mises en œuvre. Les dispositifs IdO sont des nœuds très contraints comportant quelques Ko en RAM et quelques centaines de ROM pour le stockage de programmes, ainsi que des petites unités centrales fonctionnant à très basses fréquences.

Ainsi, les implémentations directes des approches du génie logiciel ne peuvent pas répondre à ces contraintes.

Les premières difficultés rencontrées par ce travail de recherche, que l'on a reconnu comme des défis *intra-nœud*, peuvent être résumées par les questions de recherche suivantes:

- QR1 : Est-il possible d'adapter une approche modèles en temps d'exécution pour la gestion de la couche logicielle dans les environnements IdO?
- QR2 : Est cette approche assez petite en termes de mémoire et d'utilisation du processeur pour permettre l'évolutivité?

En répondant à ces questions, nous pouvons continuer à explorer les possibilités offertes par l'utilisation des modèles pour gérér la couche logicielle des grands systèmes distribués. En effet, les modèles en temps d'exécution proposent l'utilisation d'un modèle de composants pour permettre des fonctions d'adaptation sur la plate-forme en cours d'exécution, en modifiant le modèle réfléchi et en fait adopter ces modifications sur le système sous-jacent. Ces modifications visent à affecter le cycle de vie des composants logiciels afin de les adapter aux nouvelles exigences.

Ainsi, un troisième défi peut être mis en évidence pour son étude dans cette thèse:

- QR3 : Comment pouvons-nous décomposer un système informatique en composants logiciels et modifier son cycle de vie grâce à un modèle en temps d'exécution?

Enfin, comme la décomposition d'un système nécessite la distribution des composants entre les nœuds concernés, une attention particulière devrait être mis au moment de les distribuer dans un réseau IdO. En effet, comme la topologie du réseau IdO manque de la robustesse et la bande passante trouvée dans les réseaux Internet communs, en raison des exigences de faible puissance pour les interfaces réseau, une énorme quantité de trafic pour la distribution des composants doit être évitée. Ce dernier défi, la perspective *inter-nœud*, peut être représentée par la dernière question de recherche :

- QR4 : Comment distribuer *efficacement*, déployer et configurer les composants logiciels pour les appareils IdO.

Contributions

Le résultats de cette thèse sont deux principales contributions qui visent à fournir un moteur d'exécution des modèles en temps d'exécution qui soit capable de reconfigurer et de déployer des composants logiciels sur les environnements IdO.

Première contribution : Un moteur de modèles en temps d'exécution représentant une application de l'IdO en cours d'exécution sur les nœuds à ressources limitées. La transformation du méta-modèle Kevoree en code C pour répondre aux contraintes de mémoire spécifiques d'un dispositif IdO a été réalisée, ainsi que la proposition des outils de modélisation pour manipuler un modèle en temps d'exécution. Cette contribution répond aux questions de recherche 1 et 2.

Deuxième contribution : découplage en composants d'un système IdO ainsi qu'un algorithme de distribution de composants efficace. Le découplage en composants d'une application dans le contexte de l'IdO facilite sa représentation sur le modèle en temps d'exécution, alors qu'il fournit un moyen de changer facilement son comportement en ajoutant/supprimant des composants et de modifier leurs paramètres. En outre, un mécanisme pour distribuer ces composants en utilisant un nouvel algorithme appelé *Calpulli* est proposée. Cette contribution répond aux questions de recherche 3 et 4.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Our everyday lives are surrounded by plenty of devices embedding central processing units (CPU) and information storage capacities (memory). These devices play an important role in our daily activities, such as smart phones [90], as we use them to assist us on many tasks (public transport schedule information, weather, e-mail, instant messaging and so forth). Moreover, other type of devices is taking part of these new digital world, as they are introduced in the form of *"connected objects"*, which are intended to communicate not only to people through human-machine interfaces, but also to directly send and receive information from other similar objects. Indeed, the communications means have evolved since the introduction of internet access to devices other than computers, using the same infrastructure and protocols. This access to the Internet brought new possibilities to the devices able to implement the standard Internet Protocol (IP). However, these new connected objects are often not able to implement this standard way of communication, due to its poor CPU power and memory constraints, relying the communication to more sophisticated devices (*i.e.* smart phones, tablets or personal computers) aiming to reach the Internet.

The ***Internet of Things*** (IoT) aims to bring Internet connectivity to these resource constrained devices, by introducing more efficient protocols while being interoperable with the current IP. Indeed, this new communication capacities make this objects reachable from anywhere, now enabled to provide services in a web fashion. Therefore, objects being part of the IoT are intended to offer web services in a standard way, by investing APIs such as REST [40]. However, very few services on the Internet are meant to be static. On the contrary, they are highly dynamic and tend to evolve very quickly depending on the user's needs. This brings a big challenge for the use of constrained devices in this environment, since they are supported by operating systems and applications that were not conceived to provide dynamic characteristics. Thus, new research challenges on how these objects can adapt to new requirements appear. In fact, dynamic adaptation on software systems is a well-known research area, and several works propose different approaches to provide solutions for this problem. However, most of these solutions are intended for non-constrained machines such as servers and cloud computing platforms.

The main differences can be recognized as follows:

- **Memory:** Kilobytes instead of Gigabytes
- **Energy:** Milliwatts instead of Watts
- **CPU:** Megahertz instead of Gigahertz

1.1 Motivations

This research work aims to propose a way to support dynamic behaviour and self-adaptation capacities on edge IoT devices (*class 2* [11] as the biggest), taking into account their very constrained resources on memory and energy autonomy. We motivate this thesis by the real need of these devices to be highly adaptable, since their usage can vary in an important way in a short period of time. Indeed, several domains require software that can be changed depending on various external factors, such as *Smart Spaces* (Smart cities, buildings, homes, cars, and so on). These environments are subject to human behaviour, thus requirements on the needed information from objects such as sensors, and actions performed by actuators, can vary very quickly. Thus, the software running on these devices should be easily modifiable, without the need of physical intervention such as manual firmware change or device replacement.

The cost in time and efforts to change the software running on devices being part of a smart space can be very high, and should be reduced since estimations on the growth of IoT devices in these environments are exponential, making impossible to manually adapt each device. Therefore, new approaches providing dynamic and automatic deployment mechanisms on IoT environments are of high interest. Indeed, these efforts can make an IoT infrastructure a conceivable reality, bringing a more efficient use of energy for human activities, as well as a more comfortable lifestyle.

1.2 Challenges

This thesis proposes the adoption of Software Engineering approaches, and more specifically, Model Driven Engineering approaches such as Models@Runtime [77] to deal with the large software layer present in an IoT environment, taking into account the constrained resources and energy autonomy typical of IoT devices.

The existing approaches coming from the Software Engineering community to manage large, distributed and heterogeneous software platforms are intended for their use on powerful computers and servers, that are not aware of the memory usage and processing power needed to run their implementations.

Indeed, as the IoT can be considered as a distributed software platform, such management approaches are worth considering for its adoption and implementation on IoT devices. However, the nature of these devices composing the IoT differs extremely from the machines on which these approaches are implemented. IoT devices are very constrained

nodes featuring few KB on RAM and some hundreds of ROM for program storage, as well as small CPUs running at very low frequencies. Thus, direct implementations of Software Engineering approaches cannot fit these constraints.

The first challenges faced by this research work, recognized by *intra-node* challenges, can be summarized on the following research questions:

- RQ1: Is it possible to adapt a model@runtime approach for the software layer management on IoT environments?
- RQ2: Is this approach small enough on memory and CPU usage to allow scalability?

By answering these questions, we can continue to explore the possibilities given by the use of models to deal with the software layer of large distributed systems. Indeed, the models@runtime approach proposes the use of a component model to enable adaptation features on the running platform, by modifying the reflected model and actually enact these modifications on the underlying system. These modifications aim to affect the software component's life-cycle in order to adapt it to new requirements.

Thus, a third challenge can be highlighted for its study in this thesis:

- RQ3: How can we decompose an IoT system into software components and modify its life-cycle through a model@runtime?

Finally, as decomposition of a system requires the distribution of such components among the concerned nodes, special attention should be put when they are disseminated in an IoT network. Indeed, as the IoT network topology lacks of the robustness and bandwidth found in common Internet networks, due to the low power requirements for network interfaces, a huge amount of traffic for components distribution should be avoided. This last challenge, the *inter-node* perspective, can be represented by our last research question:

- RQ4: How to *efficiently* distribute, deploy and configure software components for IoT devices?

1.3 Contributions

The outcome of this thesis are two main contributions that aim to provide a complete models@runtime engine able to reconfigure and deploy software components on IoT environments.

First contribution: A models@runtime engine able to represent an IoT running application on resource constrained nodes. The transformation of the Kevoree metamodel into C code to meet the specific memory constraints of an IoT device

was performed, as well as the proposition of modelling tools to manipulate a model@runtime. This contribution answers RQs 1 and 2.

Second contribution: Component decoupling of an IoT system as well as an efficient component distribution algorithm. Components decoupling of an application in the context of the IoT facilitates its representation on the model@runtime, while it provides a way to easily change its behaviour by adding/removing components and changing their parameters. In addition, a mechanism to distribute such components using a new algorithm, called *Calpulli* is proposed. This contribution answers RQs 3 and 4.

1.4 Plan

The reminder of this thesis is organized as follows:

Chapter 2 introduces the new paradigm of the Internet of things. It describes the composition of an IoT infrastructure and gives some examples of existing similar ones. Afterwards, we describe some current solutions aiming to support software deployment on such infrastructures. Finally, a state of the art of the current IoT solutions and protocols is discussed.

Chapter 3 gives the definitions of software deployment on three contexts: on monolithic, homogeneous systems, on non-monolithic heterogeneous and distributed systems, and finally on IoT systems. Moreover, it puts special attention on the benefits of software decoupling in components, and presents an implementation example of this decomposition. In addition, it introduces the use of models@runtime along with a component model to support Dynamic Adaptive Systems, such as the IoT.

Chapter 4 presents our design and requirements towards an implementation of models@runtime for IoT devices. It presents how the Kevoree approach, an implementation of the model@runtime paradigm, can be adapted for its use in constrained nodes, for which Kevoree was not initially designed. First, we take into account the challenges of adapting such implementation in very constrained nodes, the inter-node concerns, which are not able to run JVM or code interpreters. Second, since the networking capabilities of these devices is also limited, we need to redesign the way this models will interact with huge and constrained IoT networks, presented as the intra-node concerns.

Chapter 5 presents the intra-node challenges and introduces the minimal requirements to run the previously designed models@runtime implementation, including the design of a representative IoT device meeting these minimal capabilities. Afterwards, an implementation running on a typical IoT Operating System is proposed and tested on our designed device, highlighting the main challenges of the implementation. Indeed, a large-scale method for testing is also important, thus the use of a testbed including hundreds of nodes is proposed. Two critical needed missing features on the IoT testbed available for large scale experiments are presented, as well as the technical contributions to provide such missing features. An evaluation of our models@runtime approach is

performed to show its feasibility and scalability on a large-scale IoT testbed.

Chapter 6 describes our final contribution, giving details about the software component decoupling of an IoT application, followed by the proposition of a new algorithm to distribute components taking into account the energy constraints and the inter-node network topology. Two evaluations of this algorithm, an empirical and a theoretical, are performed to show its benefits.

Chapter 7 concludes this thesis by summarizing the results of our research and highlights the benefits of using models@runtime on IoT environments to support dynamic behaviour.

Chapter 8 gives several perspectives of future research related to this thesis.

Part II

Background, context and state of the art

Chapter 2

The Internet of Things (IoT)

The Internet of Things (IoT) is a paradigm to which we put more and more attention in the scientific community [5]. Isolated pieces of the IoT are already present in everyday life, such as environmental sensor networks in streets, personal smart phones, RFID tags in many retail stores, and so on. The existence of this devices is becoming important, since the services provided are now part of a new modern environment, facilitating the most common human tasks. This plethora of objects, or *small computers*, embed software in many levels, from the very small sensor to smart buildings equipment. Regarding the ubiquitous computing vision of Mark Weiser [110], modern cities are being equipped with complex software deployments, to integrate plenty of new services, such as traffic monitoring, instantaneous weather, public transport availability, food services, and so on.

This chapter aims at describing the current requirements of an Internet of Things infrastructure, which represents our case of study. Moreover, these requirements will establish a reference to state our research problems, based on a state of the art on current methodologies and approaches which aim to build such infrastructure. Indeed, a focus on a particular use case on building automation is performed, in order to highlight the current challenges related to a network of connected objects, and its exploitation in form of distributed applications. All these requirements are then analysed to review the current approaches to provide connectivity between the most resource constrained devices, as well as their capacity to run complex software. We explain then why this features are an essential part of an IoT infrastructure. Finally, an overview of the current IoT OSs giving these facilities is presented, as well as the application development and deployment under such OSs.

2.1 The IoT at a glance

Nowadays, the current Internet, or the Internet of people, is driven by a set of technologies which were developed, extended and improved for the sake of human beings. These ***classical*** Internet technologies are those which are used, for instance, in com-

puters, tablets, smart-phones, smart TVs and streaming boxes, just to name a few. Users reach this Internet in form of web pages, audio and video streaming, mobile applications, and so on. In the literature, these are called ***web services***, which are provided from other machines known as ***web servers***. Web technologies are very well standardized, which allows a direct use of web content for any device implementing such standards.

On the other hand, systems that do not provide direct communication to the user, but on the contrary, communicate exclusively with other machines, should use different means to reach the Internet. For this, a new infrastructure that provides connectivity to devices of this kind is then necessary, and must leverage, as much as possible, existing frameworks to ease their integration and exploitation. That's why the Internet of Things should make use of different approaches, since the nature of their data and provided services differ significantly from classical web services. The ***web of things*** [34] is the goal to achieve for this new infrastructure.

2.2 Towards an infrastructure for the IoT

An IoT infrastructure can be defined as a set of interconnected objects that supply one or more Internet protocols (IP), in order to exchange information about the tasks for which they were programmed, being part of a highly connected environment including several and very different domains.

This infrastructure must be able to support connectivity and interoperability for a huge number of devices of all nature, including, for instance, the services mentioned in the introduction of this chapter §2. Furthermore, compatibility with the current Internet basis should be mandatory, in order to take advantage of the robustness of the network. Large scale infrastructures featuring a huge quantity of heterogeneous participants exists, and are being currently deployed. The smart city is one of the largest infrastructures already available, thus we can study its main properties and issues.

2.2.1 Overview of a smart city environment

In a smart city deployment [47], inhabitants can take advantage of the services derived from different interactions performed between each participant. This is what an IoT infrastructure aims to provide, making possible such interactions.

In figure §2.1 we can observe the diversity of participants and their very particular nature, which comes from small embedded environmental sensors to connected cars and smart grids.

All the characteristics of this scenario can be decoupled by sectors. For instance, the electricity network is being monitored and controlled by the smart grid [38]. This provides a better understanding to the electricity consumption, which allows a fine control of the production. Moreover, electricity leaks can also be found, resulting in instant en-

Figure 2.1: *A Smart City deployment.*

ergy savings when the detected problem is solved. In the domain of public transport, the services are provided through the tracking of buses and the availability of bikes and electric cars, which are present in several modern cities. This allows people to organize their tasks, for instance, while using a bus to move from a place to another according to timetables consulted in real-time, saving time and efforts. In very polluted cities, environmental sensors are of huge utility. Indeed, while moving inside a city using a bike or scooter, these sensors play an important role, since the availability of routes less polluted can be provided by these sensors. Finally, another very complex environment, buildings and houses, are part of this infrastructure. This part is one of the most important, since commercial and residential buildings are the most energy consuming components of a city [87]. Indeed, to monitor and control these buildings can lead to significant energy savings. Savings are achieved when a building, can turn on/off the lights when needed, as well as controlling the temperature of a room only if someone is inside, adjusting it to an optimal level. This is possible thanks to sensors that provides information about the environment (presence, temperature) and to actuators that change it (valves for radiators, lights). The infrastructure is then crucial to interconnect all these domains, which until now are working separately.

A typical use case can be when an inhabitant of a building would like to know if the electric car is fully charged and ready to go, while checking on the smart phone the traffic conditions from there to the working place. At the same time, the smart grid can use the electric car as a storage of the renewable energy produced elsewhere, and publish its performance. Moreover, this system is able to propose other options for transportation,

since it is connected to the public mobility network, which provides real time information about buses, public bike availability, and so on. In addition, the user can also know which are the conditions of air quality and temperature, using the smart city's weather service. All this ***"connected world"*** will bring many improvements to life quality, as well as a better performance of energy consumption and production, which is a major concern.

The need of an IoT infrastructure which manages all these connections is then justified, acting as a catalyst to provide new services. Traditional setups can actually offer services such as availability of parking places in a mall, however, it is not possible to know where the places are, nor to be reserved in advance. Connecting through the IoT these two entities (a parking and a car) will allow to do it, resulting in efficient distribution of traffic and time savings.

2.2.2 A Building Automation use case: improving efficiency

As stated in the previous subsection, buildings are of vital importance in a modern smart city. These buildings offer several services to their inhabitants, such as Heating, Ventilation and Air Conditioning (HVAC), lighting, parking places and waste disposal, just to name the most common ones. Moreover, some buildings are able to produce their own electricity, making use of Photo-Voltaic (PV) panels and small windmills.

Management of all this features must be done in all buildings in order to increase efficiency, and bring a more comfortable life to its occupants. Building Automation provides all the necessary pieces to support such infrastructure, making easier its administration. Furthermore, *Building Managers* supervise these buildings to refine settings and track performance. Thus, efficiency is the main challenge for Building Managers. The human surveillance efforts in addition to management algorithms running on top of the automation layer would deal with this challenge.

As residential and commercial buildings are the most energy consuming components [87] of a city, solutions to this important problem must be found. Moreover, nowadays we are aware of the importance of natural resources, and the impact that waste and misuse of these resources have in global warming, pollution and climate change. Thus, efficiency in resources usage is of vital importance. In order to cope with this situation, new policies for resource savings must be enabled. To do that, a building needs to be monitored with sensors and controlled through actuators. As we stated previously, these Building Management Systems (BMS) are built on top of many devices which are spread inside buildings. Plenty of different sensors, such as temperature, humidity, air quality, luminosity, etc., are exploited to gather different data that could be used instantly to control or regulate a given service. Instant data such as presence and temperature, can lead to an instant reaction through actuators. For instance, when a presence is detected in a corridor, ballasts are triggered to turn on the lights. Moreover, a room in which a presence is detected and a very low temperature is sensed, an actuator will immediately open a valve to heat the place. In addition to the instant energy savings given by such actions (turn off the lights when nobody is in a room and reduce the heating), the

need of more versatile sensors and actuators embedding IoT technologies comes with the evolution of services, in which a simple sensor can embed algorithms to predict the ideal temperature conditions to save even more energy, by analysing data coming from a weather station or even anticipate the occupant's arrival by getting information from his car.

This scenario is only possible if an infrastructure composed of all the features described above is deployed in current buildings, as well as fitted by default in new ones to be IoT ready. Let's take the example of an old building that is being refurbished to meet the new energy consumption requirements. First of all, the building must be analysed A sensor deployment is then necessary to find its current performance and the parts that need urgent attention, like structural damages or thermal leaks. Once the information is gathered and the possible reparations are done, a second step is to replace all the previous sensors, or their firmware, for new ones, which will perform the supervision and control of the new added features. This building which already have sensors and actuators, can change its use and consequently its behaviour For instance, a complete floor full of offices that changes its occupants, from a logistics enterprise to a government agency. A complete change in the policies for the services provided by the building will be needed, forcing the deployment of new configurations for all sensors and actuators that could be present in the floor, such as new access controls, HVAC requirements and parking assignment, just to name a few. With this very changing conditions, buildings cannot be efficient if the automation is done in such a statically way, and even less if the building continue to be unaware of external events.

Thus, the IoT find its place in the applications cited previously, since in addition to provide connectivity to the rest of external entities (sources of information), it can leverage the full capacities of the devices deployed in buildings to expose its own services. Indeed, a building with such capabilities can enable the evolution of its services, allowing to be updated or new ones to be added. Special attention must be put in the design of devices needed to perform these new tasks, in which the IoT capabilities will be deployed, since they should be ready for more dynamism, e.g. provide plug and play sensors and actuators.

Next Subsection will give an insight about the current deployed devices, and how they evolve according to new needs.

2.2.3 Devices used in Building Automation (BA)

A building must be equipped with sensors and actuators that make possible to supervise and control its services, which were already mentioned above. This is known as Building Automation (BA). Small, lightweight and often battery powered, these sensors and actuators need components that should be cheap and robust, since some environments can be difficult to access or require special installation [113]. The fact of being powered by batteries adds much more flexibility of placement, since they can use wireless communications, sacrificing autonomy. Thus, a BA device must offer a maximum of autonomy to avoid maintenance costs (*i.e.* frequent battery replacement), which can

be costlier than wiring up such device. Most battery powered devices are sensors rather than actuators, which are less numerous in buildings. In a different way, actuators are often forced to be wired, since it is complicated to use batteries for high energy consuming tasks, such as motor control for valves or blinds, for instance. On the other hand, actuators like lighting ballasts or ventilation engines are inherently connected to the main building's electricity network, thus, they do not present autonomy issues. One interesting approach to increase self-sufficiency is the use of energy harvesting [51]. These devices are able to run without batteries or cables, since they are equipped with energy harvesters which produce their own energy. Indeed, energy harvesting can be done by mechanical effort (*i.e.* pushing a switch), through small solar panels, or even with thermocouples. Many devices of this kind are available on the market from lots of manufacturers, and specialized enterprises are in charge of their deployment. In the most common cases, this work consists on the physical installation, configuration and tests of the functionalities which are preloaded by devices' manufacturers.

We can now depict in figure §2.2 the hierarchical organisation on which our previously described IoT devices are placed, showed as IoT technologies and Smart Objects.

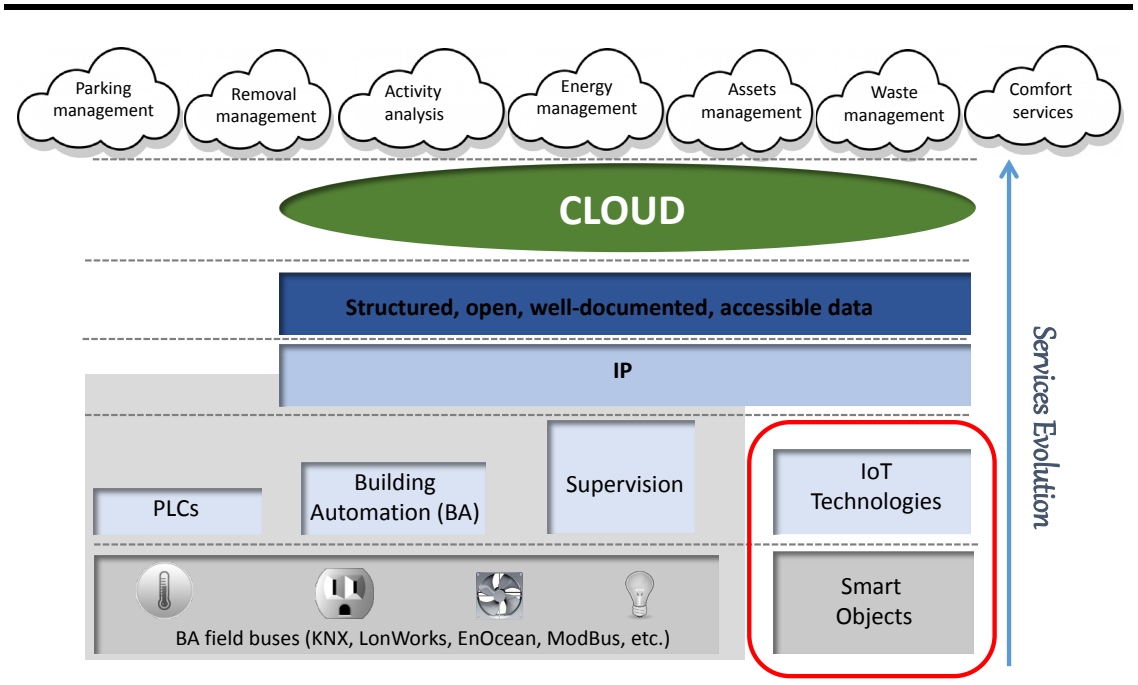


Figure 2.2: *Smart Building environment.*

Indeed, we found in the first place the common sensors present in a building, such as temperature, humidity, luminosity, gas, and so on. It also includes all meters like electricity, water and gas, which are the common services available in a building. In addition, this layer contains all the necessary actuators to control these services, the most common being valves for heating, dampers for ventilation, lights, blinds, and so on. Afterwards, there is an automation layer, which obtains the instantaneous data coming

from sensors and meters that, following the programmed algorithms in a Programmable Logic Controller (PLC), execute actions *i.e.* open/close a valve or turn on/off lights. A supervision layer then store all the generated data locally, to provide visualizations of the state of the building, either present or past. Supervisors are often deployed in industrial PCs, servers, or another high-end computer. This is required due to the very complex data processing needed to represent all the functional features present in the building, as well as a very rich dynamic visual imagery offered to buildings managers.

Once a deployment of this kind exists in a building, we can define it as a **Smart Building**. In this new environment, we can found several technologies that require many skills and players, since complexity is divided in different levels.

A deep change in these devices is then needed in order to meet the new requirements that IoT connectivity imposes. Indeed, *"intelligence"* cannot be achieved with the existing sensors and actuators by themselves, entrusting this activity to more sophisticated components. This adds lots of complexity which is difficult to manage, and can be avoided if it is distributed to the devices of the first layer. Replacement of this layer with new devices which are able to process much more information than the current ones will be needed, facilitating the implementation of IoT communication protocols.

2.2.4 Communication between Building Automation devices

While sensors and actuators can send and receive data, it must be transmitted using a specific communication way. In this context, wired solutions provide a very robust and safe way to transmit data between devices being part of a smart building. Indeed, the needed communication protocols require only a very small bandwidth, since the messages coming from sensors are, by definition, very short (*i.e.* a temperature measure). However, sensors are often needed in places which can be predefined, in order to provide a more accurate sensing. This becomes a problem when wires are difficult to install, due to the possible routing through walls and ceilings. Several devices can use wireless communication to transmit their data directly to actuators, through a process called *pairing* and *commissioning*. This is very useful when instant data is needed, *i.e.* a presence to turn on lights. However, wireless communications need a certain quantity of energy to transmit radio waves over the air, and sensors equipped with wireless transceivers are often battery powered. Thus, energy consumption for data sharing is important. Fortunately, the small size of the data messages that come from sensors needs a very short communication time, allowing a small energy consumption. Moreover, batteries should last as long as possible, sending data periodically (*i.e.* one message every 30 minutes or so) with no need of acknowledge or feedback.

This communication approaches can be reused *as is*, in an IoT infrastructure, since the physical medium used to transmit messages is the same. However, a more detailed energy management for wireless devices will be needed, since IoT protocols are more complex and could increase the network traffic.

2.2.5 Software for Smart Buildings

All the infrastructure described above is able to produce many important data. New services can be provided if all these data are analysed, then structured to give a better understanding to end users. Typical exploitation of this services are, for instance, smartphone or tablet applications that can display a building behaviour, using historical data. Furthermore, devices can be controlled remotely using this kind of applications, *i.e.* turning on/off a light using a phone or a remote computer. Alerts of unexpected building behaviour are also very useful, thanks to a software layer that can be used to send SMS to notify about an event of this kind. These new services need a software management that must run inside the building, either using computers, PLC, or other type of centralized server. IoT devices will need to embed this kind of services in a distributed way. It means that access to the instant data and, in the best case, a minimum of historical data, is needed from the devices itself, being independent of centralized servers.

Therefore, an important effort in software development is needed in all levels of a BA deployment. From embedded software to supervision engines, the presence of algorithms which manage the low-end devices information, such as sensors and actuators, are required to provide an easy to use interface between all the participants. Indeed, reconfiguration and deployment of new software for these devices is also useful to follow the building's evolution of services, such as environmental data, parking control, lighting status and so on, typical of a smart building. Thus, a transparent way to access services directly from the sensors and actuators could be the most efficient approach. Moreover, allowing a direct communication at this level would enable access to the configurations and embedded software already installed, offering a remote way to make modifications. With a deployment of this kind, interoperability between the other actors in a smart city should be easier, by using standard protocols. Indeed, standardization of protocols for BA and other city services will be needed, in order to take advantage of the software capabilities provided directly by sensors and actuators.

2.3 Current IoT solutions

As described in the previous section, a smart city will include plenty of services coming from very different providers. Depicted in figure §2.1, the heterogeneity between actors in a smart city requires software development for different platforms. Indeed, development environments can differ drastically from a scenario to another. Following the example given previously, the smart building is one of the main sectors which need more attention, since energy savings can be more considerable than any others. New buildings must be equipped with all the necessary IoT infrastructure already described in the preceding sections, in order to be efficient. However, the current issue consists in upgrading existing buildings to this new claims. This is very challenging, since classical solutions for BA are developed in a statically fashion, without taking into account the future evolutions that can be performed on buildings. Furthermore, the very complex deployment of these systems make their adoption very costly, since configuration and further changes must be made by experts. Existing deployments in some cities can

provide similar services as stated above, but the current complexity of the existing infrastructure makes its exploitation very complicated. In order to ease and enable all these new capabilities, a new infrastructure based on the IoT must be proposed. As stated in subsection §2.1, web services are the most common way to share web content in the classical Internet. Thus, providing web APIs relying in the IoT are essential. Indeed, the deployment of these new features could be very difficult, since the heterogeneity of the participants complicates the development of standard communication protocols and cross-platform web applications.

Solutions that come from the Information and Communication Technologies (ICT) field will be studied throughout this thesis, in order to find a new way to develop, deploy and maintain this IoT infrastructure. In particular, the research done in Software Engineering (SE) could be useful to propose new solutions to the issues mentioned above, since solutions for distributed systems, which are very close to the IoT environment, have been studied since long time by this branch. For instance, Meta-Modelling frameworks are already used to perform smart grid data analysis [57], which seems to be very convenient and efficient. This approach offers an abstraction level that is easier to understand and manipulate, hiding the complexity of the underlying software infrastructure which is more difficult to manage by hand. Therefore, *smart* objects participating in the IoT embedding middleware which provide these abstractions would ease the task of managing software deployment and its evolution.

2.3.1 A smart object approach

Most of the devices used in BA are built using microcontrollers (Microcontroller Unit, MCU), since they are very cheap components which can easily be programmed to perform small measurements or activate actuators, as well as implement one or more communication protocols. However, current communication between BA devices make use of field buses implementing protocols which are often proprietary, thus adaptability at the firmware level is almost impossible. While reconfigurations are possible, the very complex protocols make this task very difficult, much more when equipment from different constructors is present. Indeed, current BA deployments seem to be very difficult to adapt and reconfigure, thus a new *smart object* approach should be proposed. Since the most viable way to allow evolution of services resides in the adaptation capabilities of low-end BA devices, the research efforts on this thesis will be conducted to this particular case. Therefore, the needed hardware for these smart objects should be as generic as possible, since changes on their behaviour by reconfiguring them or deploying new software services are necessary. The Internet of Things is then the mean to reach these devices, using standard Internet protocols. However, embedding evolution capabilities into this kind of devices is not an easy task. Indeed, developing software for embedded systems due to their constrained computational resources is already difficult, while deployment of this software is also challenging since the quantity of devices could be very large to perform it manually. A study on these low-end devices' constraints is performed in the next Subsection.

2.3.1.1 Focusing on constrained devices

As explained in subsection §2.2.3, the lower layer of a BA infrastructure shown in figure §2.2 is composed of devices which have low cost, flexible placement and rapid software development as their main features. Since the goal is to enhance these devices with web features, a new BA layer must be proposed to replace the existing one, in which web services could be deployed. However, due to the constrained computational resources of these objects, **classical** implementations of Internet protocols cannot be deployed on this constrained environment. We highlight *classical* because interoperability between objects is easier to achieve using the Internet infrastructure that already exists.

The available resources will depend on the function that the object was intended to perform, and will also be driven by a specific placement and connectivity. The usage of these objects can be very diverse, from common sensing/actuating tasks (temperature, humidity, air quality, HVAC, access control) to small algorithms that provide basic functionalities, such as thermostats, coffee machines, traffic monitors, and so on. For instance, objects used for environmental sensing applications often need to be installed at a specific place [113], which can be hard to reach. This forces to either reach the sensors with a specific cable, for both power and network connectivity, or use wireless communications. On the other hand, for devices typical of home appliances or smart meters, the placement is often already defined, thus they can be easily connected to an existing network.

For the first kind of applications mentioned above, wireless communications seem to be very convenient, while in the second example, a wired connection can already exist or could be easy to provide. The communication method will determine the way we power the device, either using batteries or the electricity network. Thus, two kind of objects with different hardware capabilities can be present in this IoT infrastructure, which we can separate into:

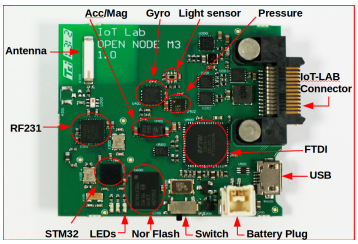
- **Constrained devices**(*i.e.* [59]). These are battery powered wireless devices which have no more than 1MB of RAM and 2MB of ROM (program data), offering a very efficient energy management. The mono-core CPU clock runs at less than a hundred megahertz.

- **Non-constrained devices**(*i.e.* [42]). Wall-powered System on a Chip (SoC) devices embedding several megabytes of RAM (hundreds or thousands), having an external storage for program and user data. Connectivity is often achieved by a wired Ethernet interface. Its CPU can run at several hundred megahertz, which results in a high-power consumption. Multi-core versions also exist, which increase even more the energy consumption.

Non-constrained devices are often able to run modern implementations of Internet protocols, since the hardware capabilities provided by objects powered by the electricity network are usually high. On the other hand, constrained devices, which are often battery powered, does not provide enough computational resources to implement current

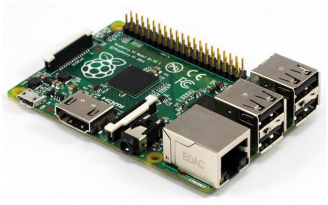
Internet protocols *as is*. However, they provide more flexibility of installation, due to their physical size and cost.

The M3 open node



- ARM® 32-bit Cortex®-M3 CPU
- 72 MHz maximum frequency, 1.25 DMIPS/MHz
- 512 Kbytes of Flash memory
- 64 Kbytes of SRAM
- NOR Flash 128Mb

The Raspberry Pi model B



- 700 MHz ARM11 ARM1176JZF-S core
- Broadcom VideoCore IV,OpenGL ES 2.0,OpenVG 1080p30 H.264 high-profile encode/decode
- SD card up to 64GB
- 256MB of SDRAM

Figure 2.3: *Constrained vs. non-constrained device.*

Figure §2.3 shows a resource comparison between these two kind of objects. We can highlight the big difference between them, as much in memory as in processing speed. While the price and the resources of SoC systems is very attractive, power consumption still a main drawback. Such kind of devices would not last a day running on small batteries, since the big amount of memory and the high-speed processor are very power consuming.

The Internet Engineering Task Force (IETF) working group *light-weight implementation guidance (lwig)* proposes a classification in the RFC7228 [11], which separates constrained devices into classes, as shown in table §2.1. For instance, objects such as the M3 open node [59], are considered the upper limit of a constrained device, being of class 2, which is a reference in this thesis as an IoT platform for experimentations.

The very low-cost of MCU based objects, coupled with their very low-power capabilities, make them very easy to produce and could be spread in most of physical environments, such as forests, streets, buildings and homes, since it is possible to make them run in batteries. Moreover, its computational capacities offer enough processor speeds and very complete instruction sets that make them suitable for the deployment of algorithms with considerable complexity. In this thesis, we will put emphasis on MCU based ***smart objects*** [67], since most of the scientific challenges come with the constrained resources in memory and energy of such devices.

Name	data size (e.g., RAM)	code size (e.g., Flash)
Class 0, C0	<<10KiB	<<100 KiB
Class 1, C1	~ 10 KiB	~ 100 KiB
Class 2, C2	~ 50 KiB	~ 250 KiB

Table 2.1: *Classes of constrained devices.*

2.3.1.2 The very constrained nature of MCU based smart objects

Web technologies are nowadays well investigated and offer many tools for fast development and maintenance, however, they rely on modern equipment: fast microprocessors with many cores, gigabytes of RAM and several terabytes in hard disks. While computers, mobile phones, and similar devices grow constantly in computing capacities, for microcontrollers the evolution is much less evident.

One of the main reasons for this slower evolution, is the cost of MCUs. In order to maintain a very low price for these devices, their size should be kept small. Since SRAM (the volatile memory type used in MCUs) takes a lot of space in the chip, it is not possible to add more without increasing the cost of the chip, which is not wanted in a very competitive market as MCUs is. In addition, the fabrication process of MCUs differs in many ways in comparison to SRAM construction. Thus, for semiconductor foundries a complex device fabrication leads to an increased cost. Finally, as stated in section §2.2, the flexibility in the placement of IoT devices is a major concern. SRAM being very energy consuming, it wouldn't be possible to power large SRAM devices using batteries. Therefore, to summarize, the construction of MCUs with huge quantities of memory is not economically and energetically viable.

2.3.2 Communication between smart objects

A smart city includes a wide range of devices that differs, mainly, in their communication capabilities. For instance, networks already deployed in smart buildings make use of particular technologies, as known as BA protocols. This kind of infrastructure is able to communicate using IP protocols, implemented at the automation layer. Most of BA PLCs provide IP implementations out of the box, either as a gateway (KNX/IP, LonWorks/IP, BACNet/IP, etc.) [63] or using proprietary modules available from the constructors. Thanks to this, the upper layers can access, in a *"web fashion"* (*i.e.* using OPC, oBIX, etc.) [83], to all this data in order to perform web-based manipulations. Research done in this area has interesting results [62], but the approaches are often based on gateway developments, which implies additional software and hardware. This also complicates the evolution of this services, since the required data provided by the lower layers depends on the configurations and embedded software provided from the first deployment. Moreover, leveraging web capabilities is the main goal of the communication

layer, thus an Internet based solution must be proposed at the lower layer.

2.3.2.1 Internet based approaches

Several communication protocols are used in the classical Internet. Most of them are based on Ethernet [3], in which a standard cable is needed for each participant in the network. A wireless Ethernet protocol was also standardized, known as IEEE 802.11 [2], providing the same features without the need of cables. Other ways to reach the current Internet also exists, such as optical fibre, satellite and different radio standards, just to name a few.

Since the IoT aims to be ubiquitous, communication using cables would complicate the physical installation of such objects. Indeed, the main advantage of MCU based objects is their very low power consumption. Thus, a very low power communication interface would allow these objects to be powered using batteries. Therefore, wireless communication protocols seem to be the best choice. However, communication interfaces implementing standards *i.e.* IEEE 802.11, were not built having low power features in mind, making them too inefficient if batteries are used as power source. Several wireless devices manufacturers and research institutes worked together to create new energy-aware protocols to provide wireless communication for smart objects. One of the standards that came from these efforts is the IEEE 802.15.4 [1]. The latter was created specifically for low-power transceivers, allowing communication ranges near to 802.11, but offering a very low bandwidth. This limitation avoids the exchange of large data packets, which also limits the protocols that can be managed by the interface, before having considerable fragmentation. Moreover, the protocol supports communications in various topologies, such as star, ring, and mesh, while the latter is the most used, since every device can act as a router, overcoming the range limitations. However, using a complex topology reduces the reliability of the network, thus the implementation of acknowledge messages in almost all layers is necessary.

On the other hand, the memory constraints present in MCUs avoid the use of classical approaches to develop software for web based applications, often written with high level programming languages. These approaches make use of common Internet protocol's implementations, which are not aware of this low resources. Thus, a new way to provide Internet functionalities must be developed.

In 2003, Adam Dunkels developed a very lightweight TCP/IP protocol stack, uIP [27], for 8-bit microcontrollers, enabling smart objects to communicate using a standard Internet protocol. With this contribution, several services could be developed to create a first IoT infrastructure. However, most of the services provided in classical Internet does not use a simple way to communicate, such as TCP/IP. Therefore, a more complete framework for web services development should be created, to establish a more transparent and easy to use approach for resource constrained devices.

2.3.2.2 6loWPAN, or how to give an IP address to any object

With the arrival of the IPv6 standard [24], which enabled a wider range of IP addresses than the previous IPv4, the possibility to assign an address to each smart object became a reality. This brings new challenges in the implementation for this new network protocol in smart objects, since the RFC2460 [24] specification was intended for high resources machines.

Montenegro et al. proposed a new specification [76] for constrained resources devices in order to communicate using IPv6 addresses. This specification, called 6loWPAN (IPv6 for low-power Wireless Personal Area Networks) was successfully implemented [35] and was able to provide interoperability with IPv6 ready devices, as it fulfilled all the requirements to have an IPv6 label[†1]. The main feature of the loWPAN adaptation layer is the compression of the IPv6 header, along with fragmentation and mesh addressing features.

Thanks to 6loWPAN compression, we have a first *"Internet of Things"* approach where every smart object can be reached from anywhere on Internet. This enables the development of web services, since the objects themselves can already be part of a network where they can offer their embedded services, hosted in their small amount of memory. A new way to represent this services is then required, that could meet the current web services specifications and approaches.

2.3.2.3 Application layer for smart objects

In the classical Internet, web services are very often represented using the HTTP application protocol [39], in an architectural style called REST (REpresentational State Transfer), defined by Fielding and Taylor [40]. Technically, REST consists of a coordinated set of components, connectors, and data elements within a distributed hypermedia system, where the focus is on component roles and a specific set of interactions between data elements rather than implementation details. Indeed, a REST API aims to provide a way to access services with high performance, scalability, simplicity, modifiability, visibility, portability, and reliability. RESTful services are very useful in the web, since they have a very easy way to operate with HTTP using simple verbs: GET, POST, PUT, DELETE, just to name the common ones. This representation could be also useful for smart objects, allowing the user, or other smart objects, to access its services in a standard way, providing interoperability with a good abstraction level.

While an implementation of HTTP is conceivable for smart objects, the huge amount of memory in both RAM and ROM turns it too heavy and inefficient to run on constrained, battery-powered devices [92]. To solve this problem, Shelby et al. standardized the "HTTP for resource constrained devices" [91], called CoAP (Constrained Application Protocol). This new application protocol is able to manage the methods described above, fulfilling the requirements for a RESTful API.

[†1] https://www.ipv6ready.org/

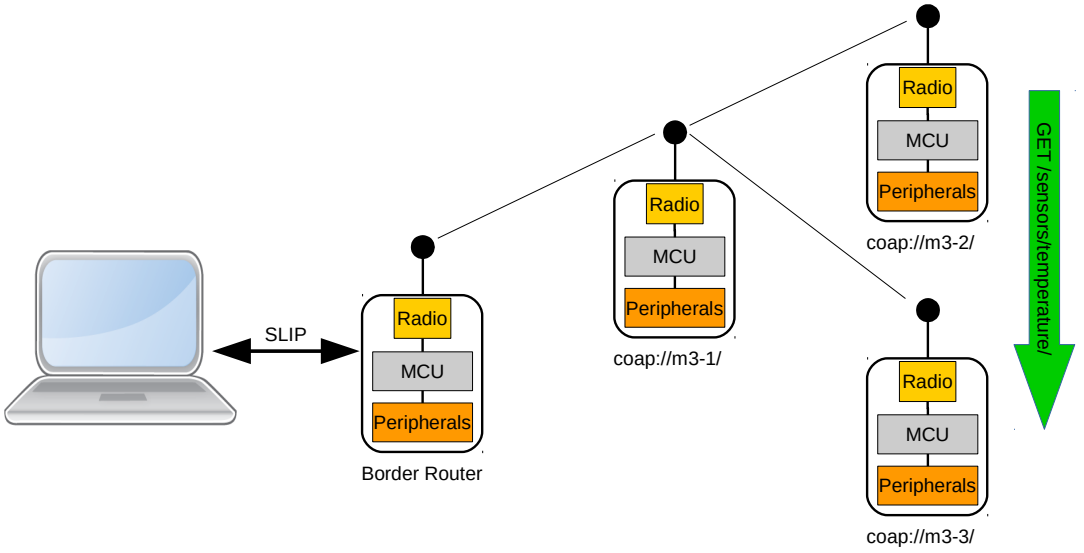


Figure 2.4: *CoAP Request from a node to another.*

Figure §2.4 shows a typical CoAP request, on which only 13 bytes are needed to format this message, leaving around 90 bytes for payload in a single 802.15.4 frame. Moreover, we can observe that a "SLIP" connection exists between the *Border Router* and a computer. This SLIP connection, acronym of *Serial Line Internet Protocol*, serve as a bridge between the radio communications and a wired network connection to the Internet. By these means, nodes behind a Border Router can be accessed from anywhere on the Internet, and more specifically, by using the CoAP protocol.

The services provided by the smart objects can be represented in a HTTP like form, *i.e.* on figure §2.4 the node with the address `coap://m3-2/` is asking for the temperature resource `/sensors/temperature` to node `coap://m3-3/`. By using the CoAP command GET for this resource, we should have a response similar to "22.5", as an example of resource representation. On the other hand, the PUT and POST commands are used to change the state of the actuators that can be present in a smart object. For instance, if we want to change the state of a LED, we access the resource `/leds` with a query `?color=red` for a red LED, and a payload `mode=on` in order to turn it on. This API facilitates the exchange of information between other smart objects, as known as "Machine to Machine (M2M)" communication.

CoAP was also developed having in mind the interoperability issue that comes when we need to communicate with other machines, which do not use CoAP as application protocol, providing an easy way to translate the messages to HTTP.

2.3.3 Overview of full IP stacks for Smart Objects

Having described a smart object capacities and capabilities, we can now analyse a layered comparison between the classical Internet and the IoT protocols stacks, built from the protocols already mentioned above.

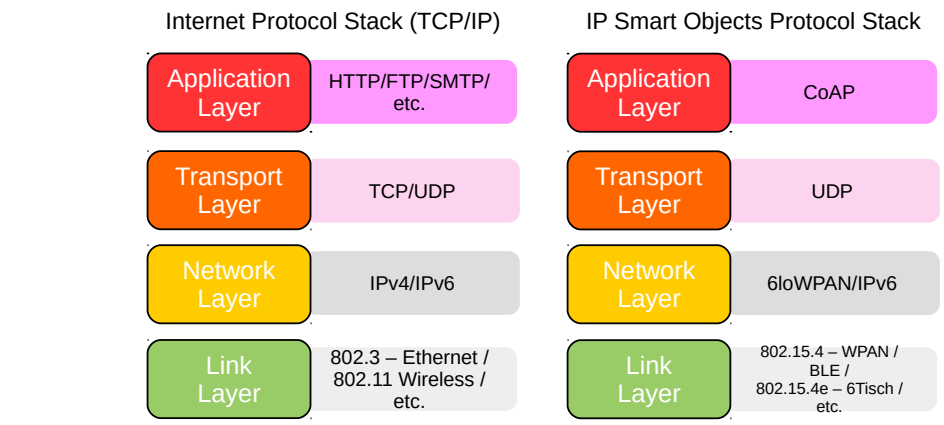


Figure 2.5: *Comparison between Internet protocols stacks.*

Figure §2.5 illustrates a layered representation of both stacks, the classical Internet and the IoT. This view allows a general understanding of some of the protocols that can enable the Internet of Things infrastructure. Both stacks are able to communicate between them, allowing users and developers to mix different types of smart objects, as well as other types of devices that can participate in the same network, such as PCs, servers, cloud and so on.

In section §2.2.2 the representation of the building services shows that, above the IP layer, the data access can be done through cloud services, which provide web APIs for further use by any web application. The presented protocols stack is the proposed way to directly communicate with cloud services, using a standard way to ease the development, test, and continuous integration of web services. Moreover, the use of a standard web API enables a straightforward method to provide communication between devices from very different entities. Taking again the example of a car driver looking for a parking, a simple CoAP GET method from the car asking to the parking if there is a place and where is its location would be enough to do the task.

An implementation of this stack by itself would be complex for a single device, since the goal is to provide easy deployment of different web services being unaware of the hardware platform. Thus, abstractions at hardware level are needed, in order to develop the applications more quickly. Operating systems (OS) for the IoT already exists, which

have made such abstractions. The next section presents two widely used IoT OS and their main advantages/drawbacks.

2.3.4 All-in-one: IoT operating systems

In the classical Internet, several operating systems (OS) are used to take advantage of the services provided on the web. An operating system provides all the necessary software to manipulate web content, based on user needs. We can find, for instance, web browsers, mail clients, file managers, audio streamers, and so on, which use several Internet protocols already provided by the OS.

A similar need comes with the IoT. However, web content in the IoT is not intended to be used directly by humans, but for other machines that gather such content and process it to offer the needed services. Therefore, an operating system for the IoT should integrate all the necessary tools to provide a full communication stack, as well as a friendly environment to ease the development of applications. In this chapter, two operating systems that provides such functionalities are described in the following Subsections.

2.3.5 Contiki

Developed at the beginning for Wireless Sensor Networks (WSN) management, the Contiki OS [32] provides a full network stack composed of several communication protocols, from the network to the application layers. Built as a monolithic event-driven kernel, as well as modular, Contiki offers a standard way to develop applications, following a structured architecture of *processes*.

Written in C, this OS and their applications can be compiled for several infrastructures, as Contiki provides enough abstractions to be platform independent. Moreover, several common MCU architectures are compatible with this OS, such as msp430, AVR, ARM, among others. Indeed, the amount of memory needed in a 16-bit microprocessor (*i.e.* msp430) is about 2KB of RAM and 40KB of ROM, which lefts a considerable amount of free memory to implement the business code.

2.3.5.1 IoT protocol stack

After several years of development, Contiki became an IoT OS by adding an IP Smart Object stack, which is described in figure §2.2. Moreover, Contiki provides the following additional features:

- Duty Cycling. ContikiMAC [30] is a radio duty cycling protocol that uses periodical wake-ups to listen for packet transmissions from neighbours Since radio communication is the most energy consuming task, this feature reduces considerably the power consumption.

Layer	Contiki Implementation
Application	websocket.c, http-socket.c, coap.c
Transport	udp-socket.c, tcp-socket.c
Network, routing	uip6.c, rpl.c
Adaptation	sicslowpan.c
MAC	csma.c
Duty Cycling	nullrdc.c, contikimac.c
Radio	cc2420.c

Table 2.2: *Contiki general network stack with the corresponding netstack source codes according to [19].*

- MAC. An implementation of CSMA/CA that avoids collisions before sending a radio packet.
- Adaptation. SICSloWPAN is the Contiki implementation of the 6loWPAN RFC4944 [76].
- Routing. ContikiRPL [103] implements the RPL [111] protocol, which is an IPv6 Routing Protocol for Low-Power and Lossy Networks. RPL organizes a topology as a Directed Acyclic Graph (DAG) that is partitioned into one or more Destination Oriented DAGs (DODAGs), one DODAG per sink.

2.3.5.2 Main OS features

In addition to a very complete IP stack, Contiki integrates other useful features:

- Multitasking kernel.
- Pre-emptive multi-threading.
- Proto-threads [33].
- The RIME communication stack [29].
- The Coffee file system [102].
- A dynamic loader of new modules [31].

The multitasking kernel, in addition to proto-threads, are the core of this OS. Protothreads are an extremely lightweight, stackless type of threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks.

The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. Protothreads provides conditional blocking inside C functions [33]. These features make the organization of applications very simple, since adding a new process creates automatically a new proto-thread, which is accessible through event-passing messages. A list of events is also available, giving to the user the possibility to create custom events, in addition to the existing ones. Depending on the needs, events are dispatched following a First In First Out (FIFO) approach when asynchronous events are used, and direct passing for synchronous events. This is the main way to communicate between process, allowing to pass memory pointers between each poll.

2.3.5.3 The Coffee file system

While constrained smart objects have often a persistent data storage, such as flash memory or EEPROM, a versatile and easy to use approach to access the saved data is always useful. The coffee file system, using default settings, needs only 5KB of ROM and 0.5KB of RAM at runtime. This is a decent amount of memory regarding the benefits that a file system brings to an OS. Moreover, the design was intended for flash memory, which has a limited write/erase cycles. Coffee supplies a garbage collection system that erase memory only when there are no more available pages to write, which reduces considerably the write/erase cycles compared to direct flash access.

Furthermore, easy storage and access to program data in a binary file form can enable firmware replacement, either fully or partially. This is achieved by Contiki through the dynamic loader, which is described below.

2.3.5.4 The dynamic loader

One of the most appreciable features in an OS is its possibility to update existing capabilities, or adding new ones. Contiki provides a dynamic linker and loader at runtime, which is able to load binary code using the standard ELF format [18]. Leveraging the abstractions already done by the Coffee File System (CFS), this dynamic loader does not need to know whether the code is located in RAM or ROM, since the CFS manages this low-level details. Since Contiki was designed to be modular, only the core contains important program code. This means that any other module can be replaceable by a new one (*i.e.* network stack, sensing applications, etc.), thus allowing bug fixing.

Figure §2.6 shows the minimal modules embedded in Contiki, leaving free space for new ones.

One of the main drawbacks of Contiki is its programming model. Even if the language is plain C, the implementation of processes and proto-threads make use of specific macros, diminishing flexibility of development. Moreover, to implement complex algorithms or big pieces of software, high level languages are the most evident solution

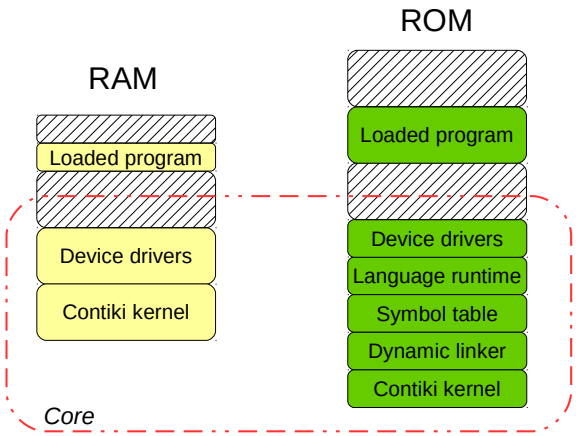


Figure 2.6: *Partitioning in Contiki: The core and loadable programs in RAM and ROM [31].*

to ease the task. Since Contiki allows only C code, the development of such complexity takes a considerable amount of extra time, compared to C++ for instance.

2.3.6 RIOT

RIOT is an IoT operating system [6], built as a microkernel architecture. It supports a full IP stack for smart objects, offering interoperability with another OS like Contiki. Its very low memory footprint make it ideal for IoT, since it can be compiled for several embedded architectures, from 8-bit to modern 32-bit microcontrollers.

Its main features can be described as follows:

- Microkernel (for robustness)
- Modular structure to deal with varying requirements
- Tickless scheduler
- Deterministic kernel behaviour
- Low latency interrupt handling
- POSIX like API
- Native port for testing and debugging

As we can see, RIOT has a very different architecture compared to Contiki. However, they share the same IP stack although the implementations follow different approaches.

Indeed, RIOT implements a generic packet buffer which can send packets either for IPv6 or IPv4. Up to beginning 2014, RIOT had a network stack more or less complete, with most of the implementations already working. Given the IoT infrastructure proposed in section §2.2, RIOT fulfil all requirements, including implementations of RPL and CoAP to provide interoperability.

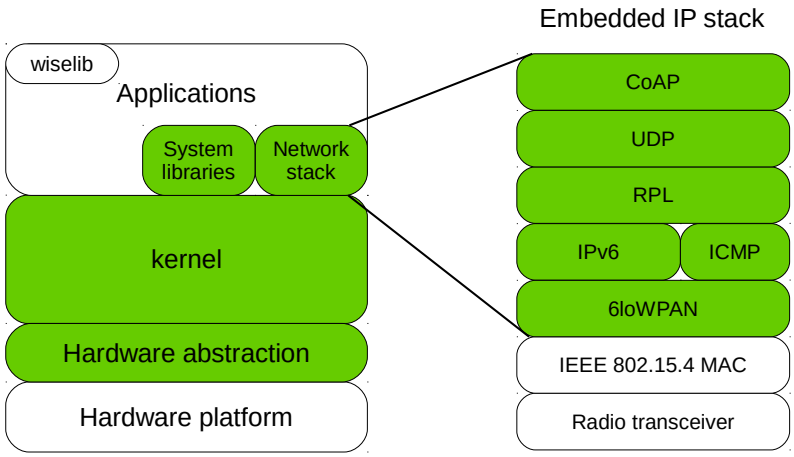


Figure 2.7: *RIOT IP stack (from [54]).*

Figure §2.7 show the modules provided by this OS, as well as the embedded IP stack for networking.

OS	Min RAM	Min ROM	C Support	C++ Support	Multi Threading	MCU w/o MMU	Modularity	Real-time
Contiki	<2KB	<30KB	•	x	•	✓	•	•
TinyOS	<1KB	<4KB	x	x	•	✓	x	x
Linux	~1MB	~1MB	✓	✓	✓	x	•	•
RIOT	~1.5KB	~1.5KB	✓	✓	✓	✓	✓	✓

Table 2.3: *OS comparison (adapted from [6]). In this table,* **x** *represent no support,* • *partial support and* ✓ *fully support..*

In table §2.3 a comparison between OS is made. It shows that RIOT offers several features that other OS cannot, having remarkable advantages such as code size and programming models. Indeed, the very attractive programming model (C++ availability) is very useful to implement self-adaptation and reconfiguration approaches coming from more abstract representations.

However, two main drawbacks were found in this OS:

i. The lack of a flexible and easy to use approach to store data in persistent memory,

and

ii. The lack of a method to replace or add new features at runtime.

Thus, the use of RIOT for current dynamic applications does not seem feasible, since these two missing features are essential to provide any update mechanism, in order to change the OS behaviour.

However, the importance of flexible and efficient OSs for the IoT has been recognized by the research community, as scientist and industry are working together to improve them and making standards to provide better interoperability [53].

2.4 Synthesis

As stated in this chapter, large distributed systems, such as the Smart City example introduced in section §2.2, need a fine management at the software level. This management of the software infrastructure will facilitate the distribution of functionalities between each device, which will result in a value-added service offered to the user. More particularly, in the given example of Smart Building, the provided services for inhabitants will need to provide a collaborative approach including several participants of the Smart City. As shown in section §2.3.4, there are integrated solutions in the form of OSs which provide a ready-to-use environment able to communicate using IoT web standards. However, each device cannot be managed separately, due to the big quantity of nodes that can be present, thus a single-device approach is not worth considering. Seen as a global IoT infrastructure, in which all devices are able to implement IP stacks, this environment should be able to take advantage of the relationships between objects, using the standard IoT web APIs. Moreover, these new devices participating in our IoT infrastructure are no longer simple sensors or actuators, but a more complex object which can provide more than a service to several consumers, as well as to other devices. Services are now provided in a more direct way, which means that the host of such services are the devices themselves. Thus, the collaboration between objects is achieved when a device can talk directly to another using Internet, while these devices are part of very different sectors of a building or Smart City. Indeed, these collaborations are achieved through web services, the evolution and constant changes in the usages must be supported by the software management layer, leveraging the already deployed web interfaces. Thus, deployment of new software components, in order to provide new services in a highly distributed and constrained environment, arises as a serious scientific problem. Indeed, the underlying devices' constraints already discussed in section §2.3.1.1 will difficult the task of continuous deployment, since each device cannot be updated easily. Even if an OS like Contiki offers the possibility to update the base firmware, we cannot use this capability as is to update it for each device, in order to provide a new service for a whole subsystem due the quantity of devices which could be involved in the case of a Smart Building, for instance. However, this functionality is the first step to provide a more abstract way to manage the software layer. Indeed, since every device being part of a

subsystem is not meant to offer exactly the same functionality, a solution able to provide the distribution of functionalities in an efficient way is then needed.

In order to deploy new capabilities, a more abstract vision of the whole subsystem is then necessary. Subsystems of this kind already exist in the classical Internet, and are known as **distributed systems** [20]. This same issue about making changes in the software layer for many nodes at the same time was also detected in distributed systems, leading to new research axes to propose new solutions. Comparing the behaviour of both IoT and classical Internet nodes, we found that they have very few differences, being one of the most important the available computational resources and the network architecture. However, the similarities such as the quantity of nodes, the application layer and the need of distributed functionalities, were enough to motivate our interest on already proposed solutions for classical distributed systems. The main goal of the conducted research in this thesis was to deal with the main architectural (both at the hardware and system level) differences mentioned above, proposing new methods to work with scarce resources and to reduce the inherent energy consumption of the extra network traffic, that can appear with the use of such methods. Thus, the main research question is: ***How can we efficiently manage the software layer of IoT systems?***

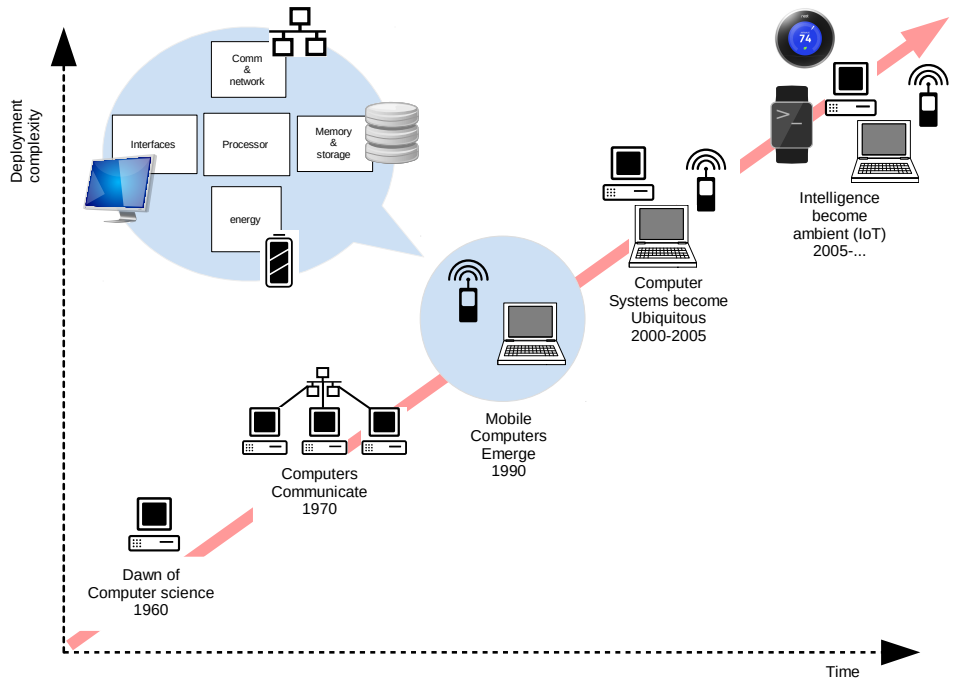


Figure 2.8: *Evolution of computer systems (from [108]).*

The evolution from centralized computing to highly large-scale distributed systems has made difficult to deploy new software into these systems. In figure §2.8 we can ob-

serve that, as the systems become more and more distributed with the time, the architectures vary and thus the deployment approaches must evolve with. Therefore, managing the software layer of heterogeneous and highly distributed systems becomes challenging, which is the case of the IoT.

The next chapter explores the current approaches proposed by the Software Engineering community, in which large distributed systems have been a case of study. Indeed, the big problem of managing software at large scale has already been covered by the research done on the field of Software Engineering, thus the findings on this matter should be considered as a source of inspiration for our concerns.

In the first part a state of the art of the current proposed solutions to deal with big systems is discussed, focusing on their decomposition in small pieces called *software components*. Moreover, existing frameworks leveraging this decomposition are studied, in order to evaluate their possible application into the IoT environment. Afterwards, solutions typical of Model Driven Engineering (MDE) are then analysed, since its integration with software components are used to manage the software layer of classical distributed systems. Indeed, one of the approaches which take advantage of both worlds is Models@runtime [78]. Under this perspective, in this thesis we aim to represent the IoT infrastructure described in this chapter in the form of a model at runtime. We take advantage of its main properties to provide an easy way to manage reconfigurations, new software deployment and updates.

Chapter 3

Managing software deployment in distributed systems

In this chapter, we will first present the general concepts of deployment, which can be diverse according to the different perspectives of software life cycle. Indeed, software deployment is not a new problem, but as shown in the previous chapter, the architectures where we try to apply the typical approaches have evolved, from centralized systems to large-scale distributed systems. In this context, deployment of monolithic, homogeneous targets will be described, followed by its main principles for distributed, non-monolithic and heterogeneous systems, leveraging software engineering approaches. The aim is to provide a general understanding of the issues already covered by software engineering, which are also present in IoT systems. Once these issues are presented, a focus on distributed systems deployment will be performed, in order to make a fine comparison on the deployment methods and its potential application to IoT systems. A state of the art of the deployment methods specially conceived for IoT devices is then discussed.

3.1 Overview

Classical distributed systems involve a high number of participating nodes and a collaborative approach to offer services. Thus, the existing solutions to manage the software layer in classical distributed systems are of high interest for our state of the art. We can start by giving a common definition about distributed systems, followed by a wider definition including software deployment on these large information systems. A definition of a distributed system proposed by Coulouris *et al.* can be found at [20]:

> *" We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages."*

With this definition, we can clearly recognize the main needs of these systems, especially the need to deploy software components for several nodes at the same time.

Indeed, this method allows to separate concerns to different nodes in the network, aiming to provide a specific, global service. Thus, communication and synchronization to achieve the tasks for what the system was designed is essential.

For this thesis, we will give our own definition of software deployment, which covers from the availability of a binary file ready to be executed, to the distribution, dynamic loading and linking on a running platform, either as a new feature or to perform an update.

Deployment definitions

In the last years, software deployment has suffered several changes in its execution environment. As today, it is not a "one-time" process, but rather an iterative operation for the sake of improvement and continuous evolution. As a result, consecutive cycles of software re-design, development and maintenance are carried through the lifespan of an application. Thus, we can define *deployment* as the process of bringing a new information service and maintaining it through the life-cycle for what it was designed, by providing updates and adaptation mechanisms.

To formalize these evolutions and the process itself, several definitions of software deployment have been proposed, every one empathizing different aspects. A more general definition was proposed by Carzaniga in [14]:

> *" Informally, the term software deployment refers to all the activities that make a software system available for use. [...] The delivery, assembly and management at a site of the resources is necessary to use a version of a software system. "*

We can find in this definition the notion of *assembly, delivery* and *management*, as well as an emphasis on a site where those activities are applied. Details of these activities are given by Hall *et al.* [55] in our last definition:

> *" Software deployment is actually a collection of interrelated activities that form the software deployment life cycle. The software deployment life cycle, as we have defined it, is an evolving definition that consists of the following processes: release, retire, install, activate, deactivate, reconfigure, update, adapt, and remove. "*

In the context of component-based systems, Szypersky [97] define it as follows:

> *" Deployment is the process of readying such a component for installation in a specific environment. The degrees of deployment freedom are typically captured in deployment descriptors, where deployment corresponds to filling in parameters of a deployment descriptor. "*

This definition highlights the use of components as a form of deployment, based on the concept of *descriptor*. Moreover, it describes the process as the installation and configuration of these components in an environment. Another definition coming from the Object Management Group Deployment and Configuration of Component-based Distributed Applications Specification (OMG D+C) is broadly used as a reference. In the specification [94], it states the following:

> *" Deployment is defined as the processes between acquisition of software and its execution. [...] In order to instantiate, or deploy, a component-based application, instances of each subcomponent must first be created, then interconnected and configured. "*

The previously cited definitions in this Section are a good guidance to establish a more complete and proper definition in our context of an IoT environment. The process of deployment is described as a development of a software product until its distribution and execution. Indeed, this process is known as *system's life-cycle*. In the case of IoT systems, the process would be the same, although the methods are very different. Componentization, distribution and installation of an IoT application have platform-specific constraints which avoid a generalization of the process.

In order to take into account these constraints, a definition of software deployment in IoT environments can be proposed as:

> *" The process between the production and execution of platform-dependent software systems, following similar activities as in classical deployment consisting in making configurations and bringing the software to its desired execution state. The process can continue along the lifetime of the software system in order to bring it to a new state via reconfigurations and updates, which are subject to the platform resources constraints. "*

It is then necessary to deal with these platform-specific constraints, using abstractions such as component-based development and deployment management, which are already proposed by software engineering tools for classical distributed systems.

We need then to discuss about the current methods to deal with these challenges, taking into account the potential changes and evolutions that will come with new requirements.

Indeed, as the system evolves, new dynamic deployments will be required to offer adapted functionalities. This is challenging in single software deployments, and even more in a distributed environment. Kramer and Magee [69] provided, albeit in an informal way, in their article that introduces quiescence, a definition for what we consider to be dynamic evolution:

> *" [Evolutionary change] may involve modifications or extensions to the system which were not envisaged at design time. Furthermore, in many application*

domains there is a requirement that the system accommodate such change dynamically, without stopping or disturbing the operation of those parts of the system unaffected by the change. "

As software complexity has increased over the years from old computing systems managed only by experts to today's personal computers managed by end users, new methods for software development and deployment appeared. Moreover, the need of reliable, robust, and fixed production costs of software motivated various research fields on software engineering, which helped to build large projects fulfilling such characteristics. In this way, approaches from software engineering have been developed and improved, such as code complexity analysis, testing tools, shared libraries, code interpreters, requirement analysis tools, compilers, dependency management tools, deployment and monitoring tools, and so forth.

In order to give a better understanding of an application's life cycle, we can divide the previously mentioned tools into three families, which aim to ease the execution and deployment of such applications:

- **Runtime tools.** With the emergence of distributed computing, *middlewares* appeared to cope with the problem of interoperability between networked machines. This was caused by the use of different communication protocols. An intermediate layer that abstracts the differences in architecture and protocols was put in place, to be used as a translator. Moreover, middlewares are also used to provide other functionalities, such as **runtime management, data persistence** or **monitoring** of applications.

- **Management tools.** In large-scale computing systems, such as distributed environments, management helps system administrators and operations teams to have control on the supervision and installation of applications. The management domain can be divided into three categories: **deployment, monitoring** and **administration.** The deployment process involves the sequence of actions that brings software from development to execution, while ensuring the adaptability of the software according to the changes in the context. Monitoring is needed to follow the evolutions of the system, in order to find problems on software or hardware.

- **Administration tools.** Finally, administration encompasses the configuration of hardware and low-level software stack, mainly dealing with the continuous growth of types of actions and configurations when using multiple machines for distributed applications.

We highlight two kinds of environments where software is usually deployed using the previously defined tools:

i. Monolithic, homogeneous systems.

ii. Non-monolithic heterogeneous systems

Indeed, these two kind of environments differ considerably one from each other, thus we need to discuss where the differences are, in order to place our research work on one of these fields. We will discuss the current approaches to manage deployment on these scenarios in the next sections. First, by defining these scenarios, a state of the art will be presented to review the current approaches dealing with software deployment challenges, followed by a specific approach for the latter environment, on which our research work has been based.

3.2 Deployment of monolithic, homogeneous systems

As the cycle of software development, release and delivery speed increase very quickly, automation of these processes is needed. One of the most common deployment process is the one which is carried in single, homogeneous machines. In this case, deployment is performed without any issues of heterogeneity, planning and coordination, present in typical distributed systems. Thus, automation of this process by developing specific tools for monolithic, homogeneous systems, results in a less challenging task. However, it is important to study these approaches as a foundational effort for deployment automation in general. We can divide these technologies into three main principles:

- **Package managers.** Mainly used in Linux and UNIX-like OSs, these tools have as goal the deployment of software previously packaged in a standard format. RPM package manager [7] and dpkg [81] are examples of standard package managers. In order to manage multiple packages, usually required as dependencies, an application can be modelled as a graph of interdependent packages. High level tools such as Yellowdog Updater Modified (YUM) [107] and Advanced Packaging Tool(APT) [93] are used to automate the deployment process, which consists in retrieving, installing, updating and uninstalling applications, by calculating the tree of dependencies.

- **Application installers.** They are based on an application-centric deployment model, in contrast to package managers which are based on dependencies. Windows Installer [64] and InstallShield [8] are examples of tools using this approach, on the basis of features and components. Features are the functionalities that can be or not installed according to the user, and components are the parts to compose such features. The composing mechanisms such as the needed components and the order to install features, which are hidden to the user, are determined from the installer.

- **Web-centric deployers.** In order to transfer software in a controlled, secured way, web-centric deployers appeared with the growth of the Internet. The purpose of this approach is to transfer executable software artefacts from a remote (web) server to an end-user's computer. With a view to secure the application deployment due to the web-based approach, only trusted applications can run within a protective environment, which is well isolated from the local resources. Implementations of this method are, for instance, Java Applets, ActiveX components,

Java Web Start (a reference implementation of Java Network Launching Protocol (JNLP) standard), .Net ClickOnce[†1] and ZeroInstall[†2].

As we can see from the above principles of deployment, the high dependency of the execution environment forces the development of automated tools targeting only a specific platform. However, making some abstractions from the platform such as web-centric and Virtual Machine (VM) execution environments, it is possible to deploy artefacts independently of the running OS. Indeed, we can also point out the fact that component-based principles are independent of the deployment platform and execution environment, thus it results interesting to explore this approach to study its main features.

3.3 Deployment of non-monolithic, heterogeneous and distributed systems

The second environment presented on this state of the art encompasses deployment of non-monolithic, heterogeneous and distributed systems. Indeed, the previously presented deployment mechanisms cannot be applied into this kind of scenario. This is due to the resources needed by complex software which in turn needs to be distributed into different calculation units (nodes), in order to achieve and accelerate the delivery of a specific service. Indeed, decoupling software into components distributed in several nodes is one of the most used techniques, which will deal with the complexity through the separation of concerns.

Therefore, the high acceptance of Component Based Software Engineering (CBSE) approaches [21] to deal with the large size of a distributed software architecture, led our investigations to explore its main features, in order to find reliable methods to manage the life cycle of these large information systems. The next Subsection will present it, followed by a common implementation.

3.3.1 Component Based Software Engineering (CBSE)

As discussed in Section §3.1, we can observe that highly dynamic and distributed systems are hard to deploy and maintain through the time. Thus, the use of CBSE approaches to support such evolutions is worth considering, since decoupling the large distributed system into small components results in an easier software life-cycle management. However, managing the deployment of these components is a known issue, which has been addressed by several investigations on this specific domain.

We will first introduce some definitions of CBSE that have been proposed in the literature, introducing the notion of *component* as the main principle. In a general way,

[†1] MS .Net ClickOnce http://msdn.microsoft.com/en-us/library/t71a733d(v=vs.80).ASPX
[†2] Zero Install: http://0install.net/

CBSE aims to leverage the main benefits of SE in terms of development, integration, maintenance, re-usability and separation of concerns, among others. However, a more specific definition was proposed by Szyperski [96], being one of the most used:

> *" A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. "*

In another definition, Heineman *et al.* [58] define a component as:

> *" A component model defines a set of standards for component implementation, naming interoperability, customization, composition, evolution and deployment. "*

We can note that this definition puts more emphasis in the development model, which not only leverages the abstract system composition, but also covers the global system deployment from the underlying pieces.

The main characteristics of a component, shown in §3.1 can be summarized as follows:

- **Interfaces specification.** The available functionalities of a component.
- **Explicit dependencies.** A component could require other component's functionalities or native libraries to work correctly. If so, such requirements should be exposed.
- **Instantiation.** Multiple instances of the given type can exist.
- **Deployment independence.** A deploy unit represent the whole component, which can be reused. However, this feature can be discussed.

Moreover, a component-based approach can define an *Architecture Description Language (ADL)*. This is useful to describe the structure of a software, in a formal way [100] [70] [75]. ADLs are declarative languages that describe a system's architecture as a set of components, connectors, bindings and configurations. Such a language can be used to assemble components, based in two elements:

- **Instances.** They are the main elements which actually embody the required application's functionalities, constituting the business logic.
- **Connectors.** A link between component's instances, determined by the exposed provided and required functionalities of the used types.

In an ADL, the specified interfaces can be bound between component instances through connectors, which are identified as "required" and "provided" interfaces. The

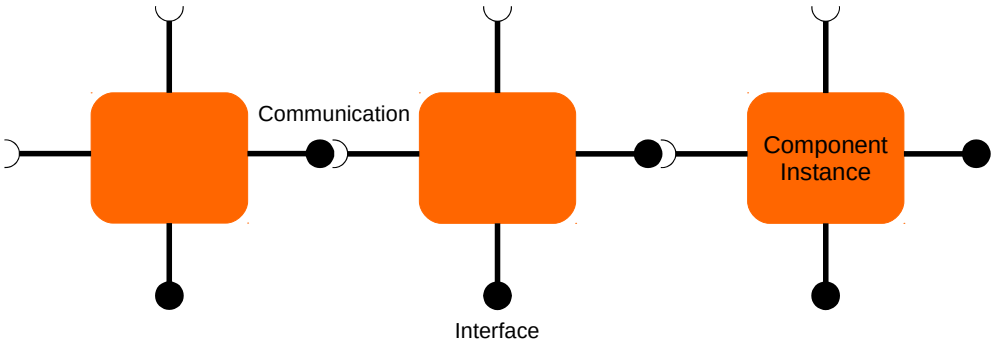


Figure 3.1: *Components and their main characteristics.*

complexity management can be achieved using a descriptive hierarchical composition, allowing scalability. For instance, Fractal [13] provides a structural description of software architecture. However, the needs can differ between systems, in order to associate functional, behavioural and system properties with the architecture.

In the case of big architectures, their division into small pieces can be very useful from a development and maintenance point of view, leveraging the component decoupling and the use of bindings between components to provide services. Indeed, an ADL can ease the task of composing this layer, giving a complete management of the component's life cycle, from the deployment to the instantiation. Moreover, we can associate to the defined characteristics of components and its composition to an execution environment, which is in charge of the exploitation. However, the component's implementation is not defined at this level. The next Subsection will describe some of the most common resources used to develop and deploy component-based applications.

3.3.1.1 Resources for CBSE artefacts deployment

The deployment facilities already introduced in section §3.2 are intended for deploying traditional applications. Indeed, the necessary deployment information is managed by these deployment systems, such as dependencies, geographical distribution on target sites, availability of required resources on these sites etc. Thus, programming of component-based software follows modular design principles, providing a broader view of the system. Afterwards, the objective is to predict the deployment state during development, which is facilitated using component-based programming. Moreover, the models present in component execution platforms provide explicit means to describe components and their dependencies. One typical example of a component based execution platform is presented in this section, in which a modular development is used in order to fit the execution environment that host the components.

OSGi (Open Service Gateway initiative)

It was originally created to provide a general component model for Java platforms, running on top of domotic residential gateways. This model aims to implement a dynamic module deployment based in *Bundles*. Moreover, the OSGi specification defines a life-cycle manager of these bundles, which are a set of encapsulated components. A Bundle designates a specific package from a JAR, which is mandatory for the deployment. Also, dependency contracts can be declared using manifest files, leveraging the notion of Java packages pointing to other bundles or parts of them. An *Activator* class represents the internal code of a component, which defines the life-cycle of the Java module including code to start and stop the application. Moreover, the OSGi framework defines the Java modules that can be deployed at runtime whose granularity dependency is represented using either the JAR or the Java package of the Activator class. This framework also defines the notion of internal service, using a central services registry as shown in figure §3.2, allowing dynamic inscription of new services. Thus, OSGi component contracts are dependency oriented. Two main implementations of this framework are Apache Felix[†3] and Eclipse Equinox[†4], being the first used in several Enterprise Service Bus and the latter the architecture of the Eclipse development environment.

A project called OpenTheBox[†5] is the result of a OSGi platform implementation dedicated to domotics. It is based on the OSGi platform Knopflerfish[†6]. Indeed, the core of this project relies in a central manager called Apam [22] running in a home automation box, which provides an isolated collaboration environment between applications [36] and controls the conflicting accesses to the shared devices [37].

The specification allows interactions anticipated by the services architecture, but they must be managed manually by the developer. This task is very complex and requires a deep knowledge of the OSGi mechanisms in order to finely handle all the possible cases to avoid errors.

Since our goal in this state of the art is to discuss the management of a large set of software services using components bound to each other, runtime control on the deployment of new components is mandatory. Thus, managing software deployment in highly distributed environments, should be defined, highlighting the current challenges for non-monolithic, heterogeneous systems software deployment.

3.3.1.2 Towards highly distributed environments

Distributed systems cannot be managed as centralized systems or single desktop machines, for instance as described in section §3.2. This is due to the very different application domain in which a distributed approach is needed. Indeed, while desktop applications can be easily deployed through local package and update managers (either

[†3] http://felix.apache.org
[†4] http://www.eclipse.org/equinox
[†5] http://openthebox.org/
[†6] http://www.knopflerfish.org/

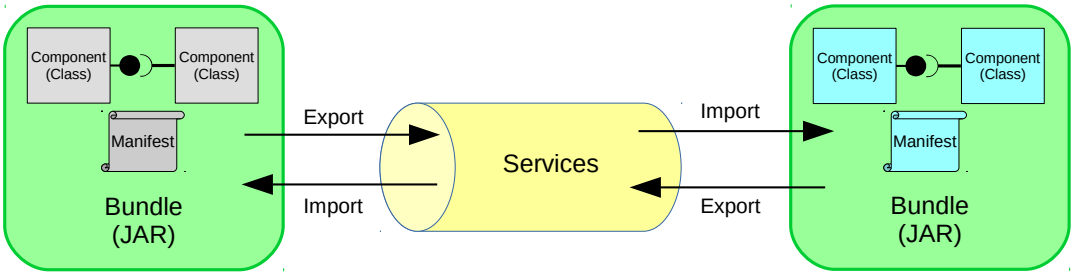


Figure 3.2: *OSGi Architecture.*

at application or OS levels), distributed applications cannot be deployed nor updated using the same methods. Let's take the example of a set of desktop machines running a typical OS, in which basic functionalities are the same, as well as provided services. When a new feature is released for such OS, a single binary including the new feature is deployed on all the machines running the OS, regardless of the underlying hardware. On the other hand, a distributed application cannot be spread among all nodes through the same binary, since each one can offer a different service, or can be in charge of only a part of a bigger system, on which the nodes' hardware capabilities are not enough to run such a big application. This is true for most of the distributed applications, since the goal is to share the resources of several machines (servers) to provide more and better services. Thus, software complexity can be divided among several machines, deploying different parts of the application in different nodes. A big challenge appears when it is necessary to deploy new features or update the current ones in this distributed environments, since it is not possible to manually add or update this features for each machine: first due to the quantity and often the physical location of the equipment and second to the uninterrupted use of the application, which cannot be stopped. The need of a fine management of this software layer is then justified.

Another issue comes with the networking layer availability, which is mandatory for distributed applications. Since communication is the main activity in a distributed environment, network robustness is then crucial. However, it is complicated to estimate the network usage for a given application. As an example, web applications are often exposed to this problem, since they offer their services to an undetermined number of clients. It is known that as the client requests increase, the application is more susceptible to crash, due to the complex network management of all client's connections. Usually this problem is solved by deploying more servers in order to increase the number of maximum connections, increasing also the cost of the infrastructure.

The difficulties presented in this section must be taken into account while developing a solution charged of the distributed applications' deployment. This solution requires support by some kind of automation tool that should cover as much of deployment ac-

tivities as possible. We can then explore the state of the art approaches for deployment in highly distributed environments, for heterogeneous and non-monolithic systems, which are the focus of our study. This is carried through the next Subsection.

3.3.2 Current deployment techniques

Most of the critical systems currently deployed in highly distributed environments need to be accessed without any interruption. Thus, stopping it to make changes such as updates, add new features or other improvements is not allowed. Such services may include life-critical systems, financial systems, telecommunications, and air traffic control, among many others. Therefore, techniques are needed to change software while it is running. This is a very challenging problem and is known under a variety of terms, such as, but not limited to, runtime evolution, runtime reconfiguration, dynamic adaptation, dynamic upgrading, hot updating, dynamic evolution and so on, sharing the common issue of dynamic deployment.

The deployment process usually starts when code is written, or generated, in a programming language, then compiled into binary code to be executed. Each module of the application is then produced in a form of object file, which a linker can then use to construct a final executable binary or a library (i.e., .dll, .so, .a) if desired. Also, a symbol table is embedded with information that defines its dependencies (i.e., shared libraries). When this code is executed, a process of dynamic linking takes place. This step is different from the linking at compile time, taking into account the information of shared libraries included in the executable file, in order to bind them dynamically to the running process. A dynamic loader should be provided with the OS, and different loaders exists for different OS which offer such functionality. The same step is performed while updating such a process, with the difference that the previous process must be stopped to be replaced by the new one. Another approach is proposed for interpreted languages, in which the compilation phase does not take place. The code in this case is directly executed through an interpreter, which can also use a hybrid approach mixing compilation and interpretation (i.e. Java and .NET). In this case, the source code is compiled to an intermediate bytecode format, which can be interpreted by a Virtual Machine (VM). Some optimizations are done in the case of Java, where classes are loaded only when needed. A graphical description of a deployment process as defined on Section §3.1 is shown on figure §3.3, on which we can observe the sequence of the steps, and the possible transitions that can take place during the software life cycle.

Thus far, the proposed deployment concept was explained, which can be represented in figure §3.3, including the creation, distribution and maintenance of a given application. This process can be valid for any application in classical distributed systems. Moreover, three categories of solutions are studied [99]: script based, language-based and model-based deployment. The trade-offs presented in this Section for these different approaches are shown in figure §3.4. As depicted, required time to establish a language-based and model-based approaches is clearly higher than manual and script-based, but can scale easily and handle deployment of complex systems.

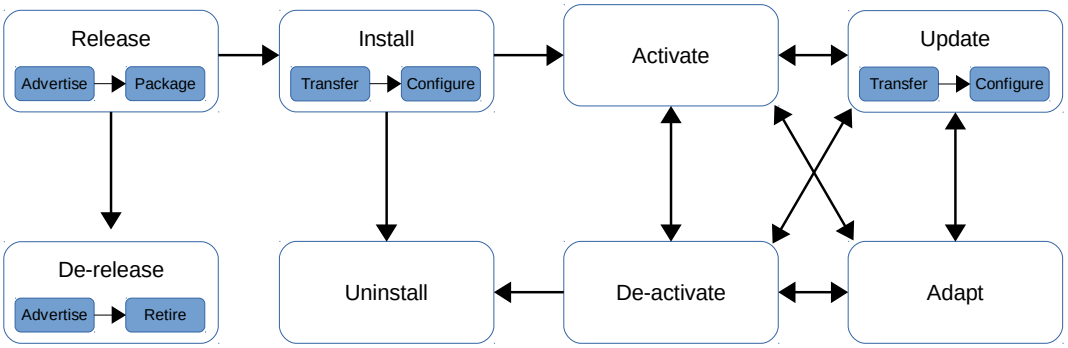


Figure 3.3: *Software Deployment Activities (from [52]).*

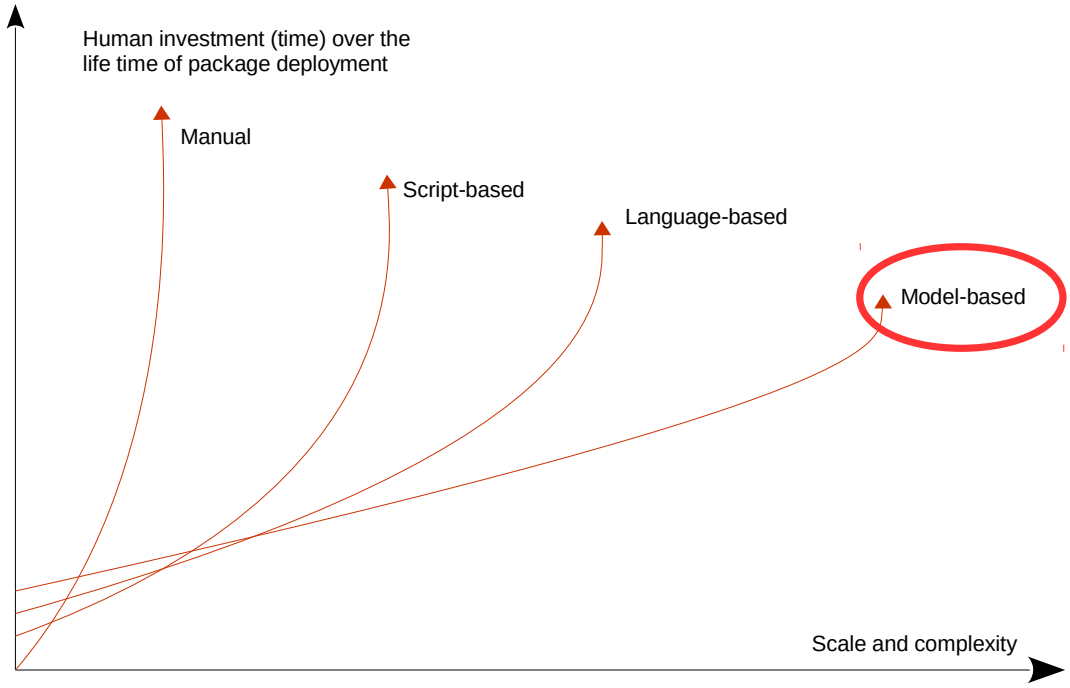


Figure 3.4: *Trade-offs between distributed deployment approaches (from [99]).*

- **Scripted Deployment.** With the aid of scripts (i.e. bash scripts), existing tools are coordinated for conducting common deployment activities on distributed environments. Remote request for files using tools such as scp over ssh are often used to copy files, described in configuration files. Execution of package managers to in-

stall software packages is also commonly used.

Usually system administrators are familiar with these tools, thus it can be very convenient at first glance, as a fairly straightforward and fully customizable approach. On the other hand, it can be complicated to maintain, and very time consuming when more complex use-cases are targeted. Moreover, models of products and site are often limited to ad-hoc models or simply inexistent. In order to achieve automation, it can be necessary a high level of expressiveness for resource description, which is also limited in this approach. One of the biggest problems using this method is the lack of traceability while leaving system administrators to enact deployment via scripts. Indeed, human errors are more susceptible to appear since it is not possible to simulate or verify the script before running, which can result in a downtime of the distributed system.

- **Language-based Deployment.** One of the improvements of script-based deployment is the use of language-based approaches. The deployment tasks are performed leveraging a configuration language, parsers and other tools. Deployers such as SmartFrog [49] and the one proposed in [109] are examples of the utility of this approach. Specialized deployment languages offer an easier usage of these tools. Nevertheless, execution of this method and scripted deployment are very similar, apart from the specialized language.

  A management runtime is often included with this language-based deployment frameworks, while the deployment workflow and the system configuration are described by the proposed language. Moreover, an abstraction layer is also defined for managing the configurations of deployed software. Indeed, a dedicated agent can then coordinate the deployment tasks according to the provided workflow, which is then executed by the distributed deployment engine achieving the maintenance of a desired application state. A higher level of abstraction is then provided by this language-based approaches, describing the actions of the deployment process, in contrast with script-based approaches. However, language-based deployment modelling does not allow for full deployment automation. Indeed, association between several custom automation policies seems to be difficult, even if the language facilitates it by specifying the deployment, as the system grows on complexity. Moreover, heterogeneity of resources and components is not well handled by the language-based approach, as the engine that executes the language should still cope with heterogeneous products and site models. These final issues are addressed by model-based deployment techniques.

- **Model-based Deployment.** An architectural model is used by model-based deployment for modelling structure of a software application together with the target execution environment. Two sides of the architectural model can be highlighted, one including components, connectors, component configurations and their requirements, while the other side targets execution nodes, network connections and resources. One of the key advantages of this approach is the decoupling of software and environment models. Moreover, the relationship between applications and the target environment are also represented. The requirements for composing components are declared on the software model, and target environment descriptions including features and resources are exposed by the runtime model. A high automa-

tion level of the process is achieved by using these models, while the re-usability is improved. When the software is deployed in different execution environments, the re-usability of the model is of importance. In the same way, the model of the execution environment may be reused for deployment of many different applications. Based on the architectural model created during the development phase, component-based systems are helpful to define the software deployment model. Thus, model-based approaches and component-based approaches are especially suitable to conceive an automated tool for distributed software deployment.

3.3.3 Docker

Following the idea of automated deployment, Docker, a client-server engine which provides deployment of applications in form of containers, has recently caught the attention of the software engineering community. Indeed, it allows rapid deployment of containers which provides a complete environment isolated from the host machine, on which a preloaded application can run from a pre-packaged image. This allows to distribute software packages using any of the methods cited previously (scripted, language or model based), since it is possible to organize the self-contained packages and deploy them as needed.

Docker is intended to provide the following services [104]:

An easy and lightweight way to model reality. It's minimalistic *copy-on-write* model allows to create and modify Docker applications very quickly. Indeed, the application built on top of Docker will use only the strictly necessary dependencies to run, and since no hypervisor is running, the use of resources is improved.

A logical segregation of duties. Operations and Developers are now separated, since the code will run in a specific container, which was specifically designed to run a specific application.

Fast, efficient development life cycle. Code maintenance is simpler, since portability is one of the main advantages of Docker.

Encourages service orientated architectures. Docker also encourages service-oriented and microservices architectures. Docker recommends that each container run a single application or process. This promotes a distributed application model where an application or service is represented by a series of interconnected containers. This makes it easy to distribute, scale, debug and provides introspection for the applications.

After a review of different deployment methods from manual to model-based using components or containers, we can highlight that, even if the initial cost as well as in time as in resources can be very high, the use of model-based approaches provides better handling of a distributed system. Thus, exploration of these methods is of high interest in our study. The next sections aim to provide a good understanding of these model-based approaches.

3.3.4 M@R in Dynamic Adaptive Systems (DAS)

With a view to ease software development and deployment for very large and complex information systems, Model Driven Engineering (MDE) focus on, at first, giving simple, abstract and different points of view of information systems, without modifying the actual system. In a second place, a branch of MDE, called Model Driven Architecture (MDA) [65] aims to provide, through Domain Specific Languages (DSLs) coupled with code generators, software development tools and methods. This approach is able to generate executable code from the abstract model, that can be generated for different hardware architectures.

Indeed, information systems have become, in only two decades, an indispensable tool for multiple human organisations. The gravity on the economic and social impacts when a dysfunction or stop occurs make them evolve into ”eternal systems”, virtually permanently available, which on practice can be translated into a per year availability rate of 99,9% [105]. Strongly correlated with the usability evolution, and the very short duration of the functionalities, these constraints impose to let these systems open to non-predictable evolutions, from the time of the first design. For instance, it is now very complicated to anticipate the next services to be offered by an airport to its users. It is even more difficult to design the connectors or interfaces which will allow future modules to be connected and leverage the airport data in order to provide such a new service. However, in 5 years this system should integrate it for the needs of a non-stop market without any interruption or alteration of the running services.

We can then consider these critical systems as *Dynamic Adaptive Systems* (DAS) [74], [79]. Continuous update mechanisms are then carried into the target platforms, in order to change at runtime the software already deployed. Taking into account this dynamic behaviour, DAS are defined using a paradigm based on components, on which the management of these components' deployment is considered as an evolution of the approach.

DAS were typically deployed on critical platforms such as airports or banks, which had no tolerance to downtimes. However, in the last years the pervasiveness of software in all domains demand a downtime rate near to zero, including phones, domestic Internet gateways and domotic services [82]. These requirements need to be supported by continuous updates, even for this non-critical systems.

The development model was also changed drastically, in order to follow the software plasticity present in these systems. When V cycle was widely used and preferred over other development methods in the 80's, coming from the design and initial specification to the code generation or implementation, nowadays Agile methods [95], which aim to bring in shorter development cycles, are more recommended and used, in order to respond more quickly to specification evolutions thanks to users' feedback. Combined to this, the new approach of Continuous Integration (CI) introduces a new test system able to be updated with new artefacts of continuous development. By adopting such a methodology, non-critical systems and DAS specified constraints are brought more closely, converging in the idea of moving together abstractions of these two domains.

Therefore, the V approach from the 80's is becoming obsolete, while agile methods tend to expand it at every development cycle. It is then possible to develop and deploy software for critical and non-critical systems in a continuous manner. MDE approaches to generate code, and especially the MDA unidirectional approach which uses a model to produce code, must take into account the inherent bidirectional development model of this continuous cycle. Moreover, code generators must provide reverse operations to allow cyclic code, which can be present at design time. This can be useful to deal with legacy code as much as at design time, which is one of the main concerns of MDE, as in the tooling, beyond an approach of *design-to-code*.

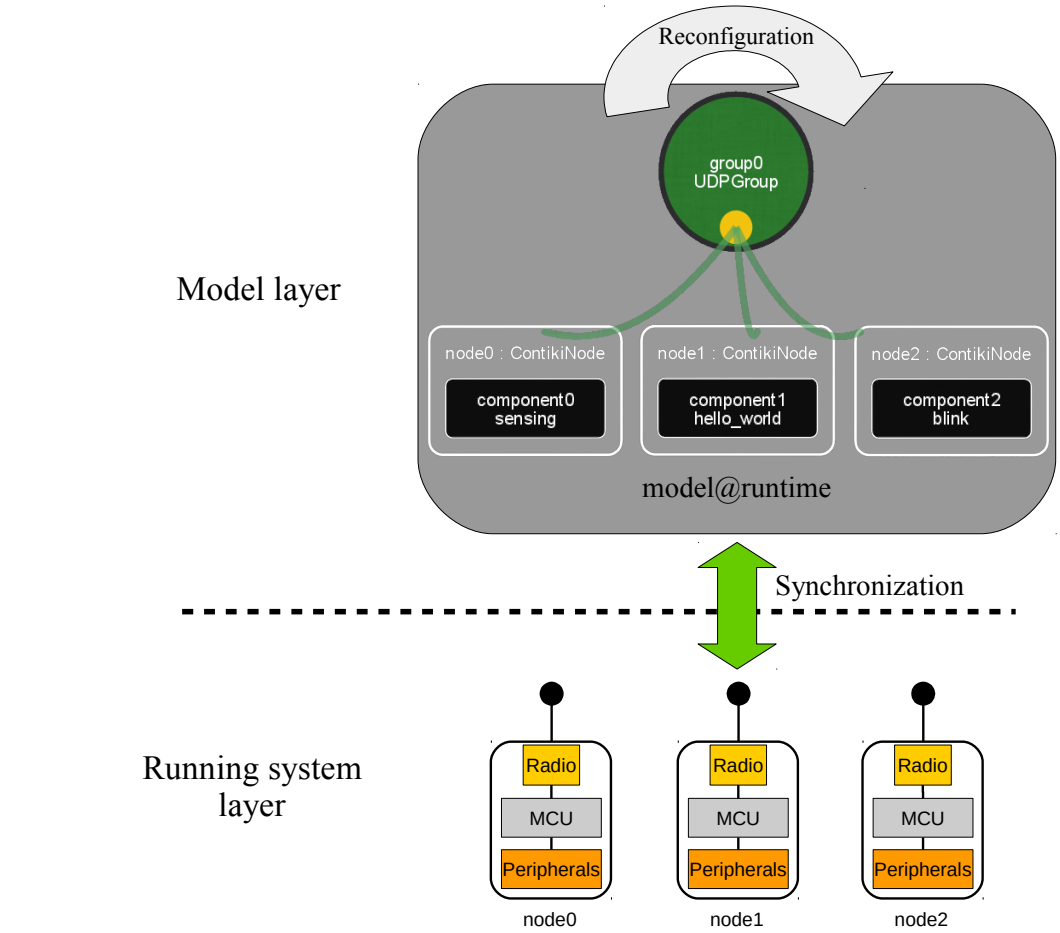


Figure 3.5: *Model@runtime principle .*

As a response to the problem of cyclic design stated previously, a paradigm called *Models@Runtime* (M@R) appeared, aiming to combine MDE techniques with tangible systems. As MDE, M@R were useful at first as a thoughtful visualization of a system, for simulation purposes [85], [10], [114]. A *permanent updated* model is then used to represent abstractly a DAS at runtime. Every different element composing this model can be represented in a schema, providing an easy navigation and an introspective analysis

through the model layer. Reasoning about the state of the system is also possible using the same layer.

Leveraging not only the introspection capability of the model, but also the intersection between computational reflection and models, Brice Morin [77] worked with this Model@Runtime layer aiming to modify it, through the reflexive model representation. Using this intersection allows to modify the internal state of a system [86]. Morin's M@R approach aim to build systems with reflexive capacities, having also intersection and introspection features present in the same layer. As an abstract layer, the proposed reflexivity can be asynchronous, allowing modifications before affecting the real system, i.e. for testing purposes. The principle of model@runtime is illustrated on figure §3.5.

These asynchronous properties separate strongly the actual system and the model, giving to MDE techniques the possibility to manipulate the reflexive layer without affecting at all the running platform. Moreover, a schematic representation leveraging the reflexive layer can be extracted from the actual system, in order to modify it. This modification can be in the form of component's adding or removal in the extracted model, then thanks to a version comparison between both the actual and the modified model, it is possible to actually trigger updates on the platform. A bidirectional connexion also exists, since any modification to the external platform will be reflected in the M@R layer. Modifications in the model can be manipulated before the deployment allowing verifications, in addition to a more flexible way to test different configurations. For instance, if we want to deploy components with dependencies, the platform itself will avoid the deployment if any of the dependencies are not met. At the model level, the same changes can be executed regardless of the order, if the adaptation execution is done after the adding/removal of the components, on which the model is less constrained than the platform. While the restrictions of the platform avoid the direct use of MDE approaches to manage the adaptations, the model can be manipulated to delay these restrictions in the application of the reflexive model, allowing MDE approaches to manipulate the model without following any order.

Approaches such as feature models or aspects [79] coupled with composition algorithms, can take advantage of the asynchronous capabilities of M@R in order to compose a model from the DAS architecture. Since all operations can be done *offline*, no constraints are imposed by the tangible platform before the deployment, while the system can decide when to synchronize. Conceiving and composing models is then essential to assemble a whole DAS. Indeed, several paradigms of composition are also needed, based on previous works [78], [66], [89], that encourage the use of software components to encapsulate the life cycle and composition operators. Moreover, these paradigms are also needed to explore the exploitation viability at this granularity level, in order to manage the different parts of DAS application layer. In the next Section, we will explore a concrete approach of the M@R paradigm, called Kevoree, which aims to provide a complete development and deployment framework for DAS.

3.3.5 Kevoree as a flexible models@runtime approach

Kevoree is a component-based development framework for applications running on DAS, based on the paradigm of Models@Runtime. This approach proposes an abstract model through which it is possible to manipulate the different concepts that characterizes a distributed system. It provides the following concepts to design a distributed system featuring dynamic adaptations:

- **The Node concept** is used to model the infrastructure topology.
- **The Group concept** is used to model the semantics of inter-node communication, particularly when synchronizing the reflection model among nodes.
- **A Channel concept** is included in Kevoree to allow for different communication semantics between remote Components deployed on heterogeneous nodes.

All Kevoree concepts *(Component, Channel, Node, Group)* obey the object type design pattern [112] in order to separate deployment artefacts from running artefacts

Kevoree aims to provide an abstraction able to manipulate the main concepts of a distributed system, in order to ease the adaptations management for this system. To do that, Kevoree proposes several features: synchronization and de-synchronization between the reflexive model and the actual system at runtime, separation of concerns between the business logic and its interactions and finally the dissemination of reconfigurations and resource heterogeneity on which the system is being executed.

Separation of concerns. A distributed application is composed of specific business logic but also communication means (code). Unlike business logic, communication means does not have necessarily specific code due to the application. Thus, it is interesting to separate these two entities, which allows to reuse the different software blocks. This simplifies the component's business logic development, since the communication concerns are separated. Moreover, the adaptation of the communication means between components regarding the context is required for a distributed application.

Distribution management. In order to distribute different functionalities to the different nodes in a system, an abstract representation is provided, in which these characteristics are modelled and can be manipulated at runtime.

De-synchronization. A process of validation carried in the reflexive model before its application is one of the main advantages of this feature. This de-synchronization is possible thanks to the concept of models@runtime that is the base of Kevoree, in which an adaptation is defined through a model that can be validated. Coherence of the configurations is then validated to be sure that no unstable behaviour can be reached or a complete breakdown can happen. In a distributed context, this is of high importance, in order to avoid an adaptation that cannot be executed by all nodes.

Adaptations dissemination. Once an adaptation is executed, every node must be notified of this change, so it can be taken into account by the whole system. However, in a distributed system it is not possible to guarantee an uninterrupted communication between nodes, since networks are subject to communication errors or disconnections. These constraints are considered in the dissemination of the adaptations, to provide a coherent evolution. Different synchronization methods are then used regarding the communication means between nodes.

Execution platforms heterogeneity. Distributed systems are composed of several execution platforms, such as mobile nodes (smart phones), PC, servers or embedded systems. It is then necessary to represent in the Kevoree model the differences between platforms and their specific characteristics.

Kevoree supports multiple execution platforms (e.g., Java, Android, MiniCloud, FreeBSD, Arduino). For each target platform, it provides a specific runtime container. As a result, Kevoree offers an environment to facilitate the implementation of dynamically reconfigurable applications in the context of distributed systems based on different execution platforms.

Moreover, the principle of a model@runtime lies in the general knowledge of the entire system (the reflected model of the running system), which is present in every participant. Indeed, the reflected model should be available in memory for its rapid manipulation, thus we can imagine the big quantity of memory needed to represent large models featuring a vast quantity of nodes. In addition, each node can have instances of one or more components and its parameters, which increase even more the size of the model in memory.

Once we have discussed how software deployment on a highly distributed and heterogeneous system can be managed, we need to discuss about the existing approaches to deploy software on very constrained environments which were already introduced in the previous chapter. The IoT is a part of this very constrained systems, thus we will present in the next Section how software can be deployed onto these systems.

3.4 Towards software deployment on the IoT

Once a state of the art discussing the current approaches to provide dynamical behaviour at the application level for highly distributed systems has been conducted, a summary of current deployment techniques used on constrained environments will be presented. Indeed, several research works propose component models coupled with code distribution approaches, in order to manage the software layer in a constrained environment. However, these existing works are rather intended for WSN. While this represents smaller networks without IP connectivity, WSNs are composed of devices typical of the IoT, thus the applicability of these approaches on larger, IP enabled networks such as the IoT is worth considering. We will then introduce two aspects of software deployment on IoT systems: a static and a dynamic approach.

3.4.1 Static deployment

This is a straightforward solution to deploy new features or bug fixes in embedded systems, an essential part of the IoT infrastructure presented in this thesis. It consists in changing the entire kernel image for a new one, which must be either flashed physically or transmitted through the network, followed by a complete reboot of the system. Indeed, sharing a large kernel image can be very energy consuming for wireless battery powered devices. Advantages of this technique are the possibility of deep changes into the kernel or applications, as well as deployment of a complete different OS.

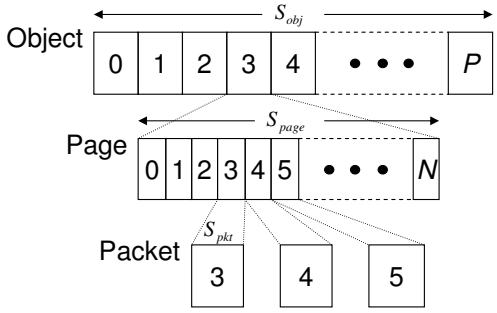


Figure 3.6: *Deluge's Data Management Hierarchy (from [60])* .

Through a full kernel replacement, *Deluge* [60], a very popular protocol using this approach, supports distribution and flashing of a binary image for constrained devices, through the TOSBoot bootloader. It was mainly developed for TinyOS [72] a WSN OS. Indeed, this protocol divides large objects into *pages*, which is the basic unit of transfer and provides three advantages: (i) it limits the amount of states a receiver must maintain while receiving data, (ii) it enables efficient incremental upgrades from prior versions and (iii) allows for spatial multiplexing. Moreover, each page is divided into smaller packets of a fixed size, as we can observe on figure §3.6. Each packed is checked with a 16-bit Cyclic Redundancy Check, in order to ensure consistency. The protocol establishes three operation modes: *MAINTAIN*, *RX* or *TX*. The *MAINTAIN* state provides the information about the pages and version of them, while putting them available and ready to be shared. Indeed, when a page is correctly received, it can be shared again to another node requesting it, thus having the entire object is not necessary to share parts of it. The *RX* state will be enabled when a node is requesting pages of an object. Moreover, it ensures the reception by choosing actively a trusted source, which can be changed if it is not reachable anymore. Finally, during *TX*, a node is responsible to broadcast any requested packets for a given page until it finishes to send all packets. Then it comes back to the *MAINTAIN* state.

As stated previously, this technique incurs in a high energy overhead while distributing the entire image through the network. Moreover, re-installation of a new firmware requires a system reboot, disrupting any running applications which can lead to data

loss of those applications. In order to solve the first problem, some approaches [61], [88] supply a code image comparison in order to provide incremental updates, reducing the size of the update. Indeed, the main drawback of such approach is the significant increase of CPU energy consumption, due to the complexity of the difference calculation mechanism.

We will then discuss, through the next Subsection, the current possibilities in IoT devices to actually allow dynamic behaviour by deploying new software at runtime, while focusing on the decomposition of these software artefacts in smaller deploy units to cope with the main drawbacks of the static deployment.

3.4.2 Dynamic deployment

The process of software deployment can differ significantly from classical distributed systems to IoT systems. This is due to the underlying hardware differences discussed throughout this state of the art. While non-constrained nodes present in classical distributed systems are able to run VMs to execute precompiled bytecode, written in high level languages such as Java, IoT devices are not able to run a complete JVM, but rather very limited ones such as Maté [71] and Darjeeling [12]. Moreover, related research [84] found JVMs very resource consuming in small devices typical of IoT, since they add a considerable CPU overhead while running, which avoids a long-term use for battery powered nodes. Thus, we will focus on software deployment for applications compiled as binary code, which can be executed directly by the native platform. Indeed, in the embedded systems domain, which embodies most of the devices used in IoT, bare metal applications are the most common procedure to provide services or functionalities. Physical flashing of the binary image is the usual deployment method, since it can be done at the manufacturing step of the embedded system, without considering any further firmware changes. With this limitation, it is even more complicated to distribute applications among several nodes which are physically separated, since manual flashing of every node in a typical IoT system cannot be worth considering due to the huge number of nodes. Thus, we need to study the current alternatives to deal with this problem.

Operating systems used in embedded systems and Wireless Sensor Networks (WSN) provide several methods to deploy new applications at runtime, such as:

i. Scripting languages [28] [68].

ii. Virtual Machines.

iii. Kernel replacement.

iv. Position Independent Code.

v. Relocatable code.

As we discussed in this section, research shows that only non-interpreted code is worth considering for long term applications. Thus, scripting languages and VM approaches

will be discarded as a method to deploy new features, in the context of this thesis. Moreover, kernel replacement has been discarded due to its very high energy consumption while transmitting large kernel images.

As presented in this section, applications deployment for IoT systems differs in several ways from classical distributed systems. In order to cope with this differences, which are mostly hardware and network related, approaches coming from the embedded systems and WSN domains have been proposed. We will discuss the two last methods already introduced above, aiming to provide dynamic deployment:

- **Position Independent Code (PIC).** As mentioned previously in this chapter, the executable code usually follows the process of dynamic relocation to find the needed symbol's addresses, then execute the code. PIC follow a different approach to run applications. It consists in the use of relative jumps rather than absolute addresses to find symbols, mitigating the need of run-time relocation. Several overheads can be introduced by the use of this technique, such as pre-loading of addresses before making jumps, more needed jumps while calling kernel or another module, functions registration and de-registration and the necessity of a jump table. Furthermore, a PIC compatible compiler should be used, in order to produce a loadable module using this method, which can be unavailable for certain CPU architectures. For instance, AVR microcontrollers supports this type of compilation, but it is limited to a 4KB program size, while it is not known a compiler supporting PIC for MSP430 CPUs. Applications running non-PIC methods to deploy new modules have shown a 13% better performance compared to PIC [26]. However, the global efficiency while running most of PIC modules is the same as if they were flashed directly on the device.

- **Relocatable code.** This technique is the one used by classical OSs such as Unix based and Windows, also implemented for IoT devices [31]. It uses run-time dynamic linking, relocation and loading of modules compiled using the standard Executable and Linkable Format (ELF). This format includes the program code and data, as well as detailed information about unresolved symbols. To resolve them, the OS must adjust properly the absolute addresses included in the file, depending on the module's location in memory. A relocation type also embedded in the ELF file specifies how the data or code addresses should be updated. These types depend on the CPU architecture, for instance, an MSP430 CPU counts with only one type of relocation, while the AVR architecture has 19. Both architectures are widely used for embedded systems and IoT devices. Once each unresolved symbol is updated with the new address, the `.data` and `.bss` sections of the ELF file can be loaded into RAM, and the `.text` is copied to ROM, then the program can be executed following specific OS functions. Overheads of this method include the transmission over the network of a large ELF file depending on the 32 or 64 bits architecture, a symbol table in which names are used to represent each unresolved symbol, also increasing the size of the ELF file, and finally a CPU overhead is incurred while resolving symbols. As in the PIC method, no reboot is required to run the new application or to apply an update.

	Mechanism						
	VM	**PIC**	**Reloc.**	**OS Protection**	**Kernel Modif.**	**Kernel Replacement**	**Loose Coupling**
Maté [71]	•			•			•
TOSBoot [60]						•	
SOS [56]		•			•		•
Contiki [32]			•			•	•
RETOS [15]			•	•	•		•
Darjeeling [12]	•			•			•
SenSpire [26]			•		•		•
Enix [16]		•					

Table 3.1: *Feature Comparison (adapted from [84]).*

Other features of the proposed approaches can be considered, such as OS protection, low level kernel modification and even a whole kernel replacement. The loose coupling between new added modules and the underlying kernel is also important, since added flexibility of development allows scalability and fast upgrade of components. Several operating systems using the discussed approaches are shown in table §3.1. VMs are included in this table since they offer high loose coupling and OS protection, and can be suitable for very short life-cycle applications running on IoT devices. Moreover, SOS, RETOS, SensSpire and Enix, are included as a matter of comparison, but the implementation and design details are out of the scope of this state of the art.

Component based approaches

As for Component-based module dissemination and deployment, an effort to provide component models for WSN were already done by the FiGaRo approach [80], implemented on top of the Contiki OS. It aims at providing both a component model for modules development and a distribution mechanism using a new dissemination protocol. A very complex component model relying on the use of C macros handled at compile time is proposed for modules development, which are directly used by the developer. Indeed, this programming model demands a deep knowledge of the set of features supported by the approach, increasing development complexity. Moreover, node's updates and reconfigurations are determined by the programmer using a Domain Specific Language (DSL). Dependencies in other components should be explicitly declared also using such DSL, which are not managed by the node itself. Thus, a dependency graph is built by the node to find if a dependency is not met, but the approach does not explicitly explain if any action will be performed in case of a non-satisfied dependency. It rather performs a recursive graph parsing, until the needed dependency is met, otherwise it excludes the new component or update. In order to achieve non-monolithic deployment in an heterogeneous network, the approach provides a mechanism of rules that are declared using the same DSL, in order to filter the targeted nodes, instead of a fine selection of the nodes to be updated. Moreover, even if the approach relies on the Contiki's ELF

loader, the extensive use of this DSL incurs in an overhead since system function calls are not done directly, but rather using the macros provided by the component model.

As for the component's distribution mechanisms, a routing protocol is proposed to find the best path to distribute a component. Thus, it relies in a mesh topology which is built by piggybacking the current value of a node's attributes on every outgoing message. Moreover, the proposed protocol makes use of complex message passing between nodes in order to calculate the best path for the component's chunks to be transmitted. Indeed, this can result in lots of redundant paths, although some efforts are made to reduce them. The presented protocol strongly depends on other application's traffic to exploit enough messages in order to determine a representative topology of the mesh. Therefore, if there is no enough traffic, the protocol can ignore the whole network topology, resulting in an inaccurate components distribution.

The main found drawbacks can be summarized as follows:

- Complex component model relying on C macros, which must be well known by the developer,
- Use of pre-defined rules to achieve fine selection of deployment targets,
- High CPU overhead caused by the use of non-direct function calls,
- Component distribution relies on complex routing dependent on network traffic.

After the analysis on the static and dynamic approaches, we can observe that they succeed on the goal of deploying new software on very constrained nodes, although they have several drawbacks. Indeed, most of them are related to the energy they need to succeed, and lacking of efficient ways to distribute and manage whole firmwares or parts of it. Moreover, the programming model of the proposed approaches can also be difficult to learn, compared to traditional C programming. We need then to state what are the main challenges while designing a new approach to cope with these drawbacks.

As presented on Section §3.3.4, the Models@Runtime approach can deal with most of these drawbacks. Indeed, it proposes a way to model the network and the software layer, without being dependent on the programming model and language used to develop components. Moreover, the network information is also modelled, thus there is no dependency on the topology to localize a specific node. Therefore, it is worth considering to analyse if the existing M@R implementations can be adapted to provide a way to manage and enact software deployment on IoT systems.

3.5 Conclusion

The focus of this work is to propose IoT solutions based on the three tools discussed on Section §3.1: runtime, management, and administration of the software layer on top of this constrained systems. More specifically, a management tool is proposed to deal with

the deployment issues typical of IoT systems, leveraging software engineering existing approaches to solve very similar issues.

Indeed, decoupling IoT environments into software components and managing their deployment is not an easy task, since memory constraints, processing power, energy autonomy and network topology prevents the direct implementation of the same approaches used in classical distributed systems, which is one of the motivations to conduct this research.

Since there are no widely used IoT OS or execution environments for constrained devices, it is very complicated to develop automatic deployment methods using the same abstractions. Moreover, the use of high level execution environments such as VMs is not worth considering, due to the scarce resources found on these devices face to the high resourced needed by a VM.

In conclusion, even if deployment mechanisms for IoT devices exist and can be improved to reach a good reliability level and ease to use, manual firmware flashing for updates and new features should not be worth considering, as it was explained in the previous section. Moreover, figure §3.4 already shows that in classical distributed systems manual deployment is not suitable for large-scale systems.

Given the available methods of deployment for new applications or feature updates, it is then necessary to evaluate which is the most advisable approach for its use and adaptation in an automated deployment manager. Taking into account the already described model-based techniques, the criterion to evaluate the previously presented approaches will be based on the capability of such approach to be easily decoupled into components, that can be disseminated over the network using a minimum of transfers. Moreover, it is very important to have a good implementation of the local deployment procedures, as well as the necessary tools to share deploy units to the entire network. We also highlight the need of specific code distribution, since each node can perform a different task, thus needs different software components to be deployed.

The presented CBSE and Models@Runtime concepts are essential to build our IoT software architecture. Indeed, our research efforts were led to the design and implementation of a software deployment manager dedicated to IoT systems, in order to enable automatic deployment and dynamic adaptations. Thus, the introspection and the dynamic reconfiguration facilities offered by the models@runtime paradigm, and more specifically the Kevoree approach, are an interesting source of inspiration. Indeed, the need of a unified tool managing the software layer of an IoT environment comes with the design of such IoT architecture. However, at this point the main differences regarding IoT systems and classical distributed system become more relevant, since the already investigated approaches are intended for the latter.

Taking this state of the art as a source of inspiration, the purpose of this thesis is to propose a novel way to manage software deployment in IoT systems, facing at the same time issues typical of distributed systems and the new ones raised by the constraints of IoT devices.

The next chapters are intended to explain the main contributions of this thesis, focusing on the already discussed topics on this chapter, to finally propose an automated deployment engine for IoT applications.

Part III

Contributions

Chapter 4

Models@Runtime for the IoT

Looking at our state of the art, various techniques have been proposed to enable software updates, which were presented in table §3.1. As for the full kernel replacement mechanism, which can be managed by an automatic tool [60], the excessive amount of power needed by this approach does not seem appropriated for a highly dynamic infrastructure such as the IoT. The approaches making use of virtual machines were already analysed and it was found that they are not suitable for long-living applications [84]. Even if some component models [80], [98] were proposed to provide an abstraction layer for the life-cycle maintenance, it results in a very complex programming model and routing protocols to distribute components, in addition to a finely tuned memory manager for specific platforms and hardware architectures, which reduces scalability. Thus, the lack of a deployment manager which provides kernel modularization using relocation mechanisms, which seems the best way to provide new features and updates, motivates our research to find an automatic and scalable approach to provide such a manager.

In this chapter, we describe the main challenges while designing a new middleware dedicated to IoT devices, in order to enable the management of software deployment and the dynamic reconfiguration of IoT systems. Indeed, our middleware is inspired from the Component Based Systems and the model@runtime paradigm which have been already described in the previous chapter.

4.1 IoT specific requirements for model@runtime

As mentioned in the previous chapter, the model@runtime approach has been proposed and designed for distributed systems where nodes are powerful computers interconnected through a very high-speed network. Some inherent characteristics of this approach were thus designed without taking into consideration the very specific and constrained nature of IoT systems. In this section, we present the specific characteristics of IoT systems which make them incompatible with the current design of Model@Runtime, and we elicit a set of requirements to design a model@runtime approach for IoT systems.

Compared to a classical distributed system, an IoT system mainly differs through the three following characteristics:

- an **energy constraint**: most nodes included in an IoT system are battery powered with limited capacity. This particular way of powering the computing nodes has a direct impact on the development of software. Indeed, if the software running on those systems have not been designed with the energy constraint in mind, it will drastically reduce the life time of the system.

- a **very limited computing and memory resource**: most nodes included in an IoT system presents a very limited amount of computing and memory resources compared to a more classical computing node. These resource constraints have an impact on software development since classical algorithm and design may not fit into such constraints.

- a **multi-hop routing infrastructure**: in an IoT system, most nodes are interconnected through a multi-hop routing infrastructure. This specific way of routing packets through other nodes in the network directly impact the way of designing inter nodes communication since fulfilling any specific network communication may drain energy and computational resources from several nodes in the network.

Therefore, designing a specific model@runtime approach for IoT system will require to cope with these three main characteristics which differs from classical distributed systems. Typically, a model@runtime approach for distributed system is designed with two main components:

- a **model**: which represents the current state of the whole distributed system. This model typically represents three layers: (i) **the hardware level** with all nodes included in the system; (ii) **the network level** with all communication path between the nodes, and (iii) **the software level** with all software components, their configurations and their communications paths. This model is used as the corner stone of the approach to deploy and perform dynamic evolution of the software system.

- a **software agent**: running on all nodes. This software agent is in charge of all activities to transform this declarative model into a running system. These activities include model interpretation (loading, comparison, and so on), software component downloading, and dynamic software loading. All software agents in the system are independent and performs independently the specific tasks needed on each node.

In an IoT system, the software agent has to be redesigned and implemented in a different way to take more carefully into account the three inherent characteristics described above. The model has to remain the same conceptually, in order to interoperate with hybrid systems which includes IoT nodes together with computing nodes with less limitation of computing power. We divided the problem of designing a model@runtime approach for IoT into the following two main challenges:

- the **intra-node challenge**. This challenge is related to designing the software agent part together with a typical IoT node in order to fit all activities related to local model interpretation and dynamic software loading into the resource constraints IoT node. This challenge also relates to the evaluation and minimization of the performance and energy overhead of the model@runtime approach.
- the **inter-node challenge**. This challenge is related to the design of new communication schemes to take into account the underlying routing topology, in order to minimize the energy consumption of the required model@runtime communication. This includes in particular a communication scheme to minimize the energy consumption related to software component downloading and distribution.

4.2 Kevoree for the IoT

Focusing on our first challenge, the intra-node case, we aim to develop a software agent able to provide a set of tools to leverage the models@runtime advantages and properties, in order to provide an automatic platform for IoT software deployment and maintenance. This section will present the design of such an agent, taking into account the previously mentioned characteristics.

One of the models@runtime approach implementations already discussed was the Kevoree framework. As presented in section §3.3.5, the base of this framework is the Kevoree meta-model, which was designed to follow a distributed architecture and also offers a minimalistic component model. Indeed, the implementation and design of Kevoree did not take into account any restriction on the underlying system, thus relying on high memory and storage capabilities with fast processing features. Thus, we cannot envision the direct mapping of this implementation on very constrained nodes such as IoT nodes.

However, some of the Kevoree activities are interesting for their use on a distributed environment such as the IoT. Indeed, Kevoree is able to perform reconfiguration and deployment tasks, which are the main functionalities we are searching for. To be precise, we are interested on the following features:

- **The meta-model to represent the whole distributed system.** The representation proposed by Kevoree through the meta-model is very close to our IoT environment, since it focuses on independent nodes and provides abstractions for the communication means, which is the main activity of an IoT node.
- **Principles for model manipulation.** Several tools are needed while changes on the system are reflected on the model. It is then needed to change parameters, add or remove components, check for changes between old and new models and so on. Thus, tools like model serializer/deserializer, model comparing, and model visitors, are essential for a functional Kevoree implementation.
- **Kevoree editor** A web editor is available for model checking/editing. This is very useful while triggering adaptations by hand.

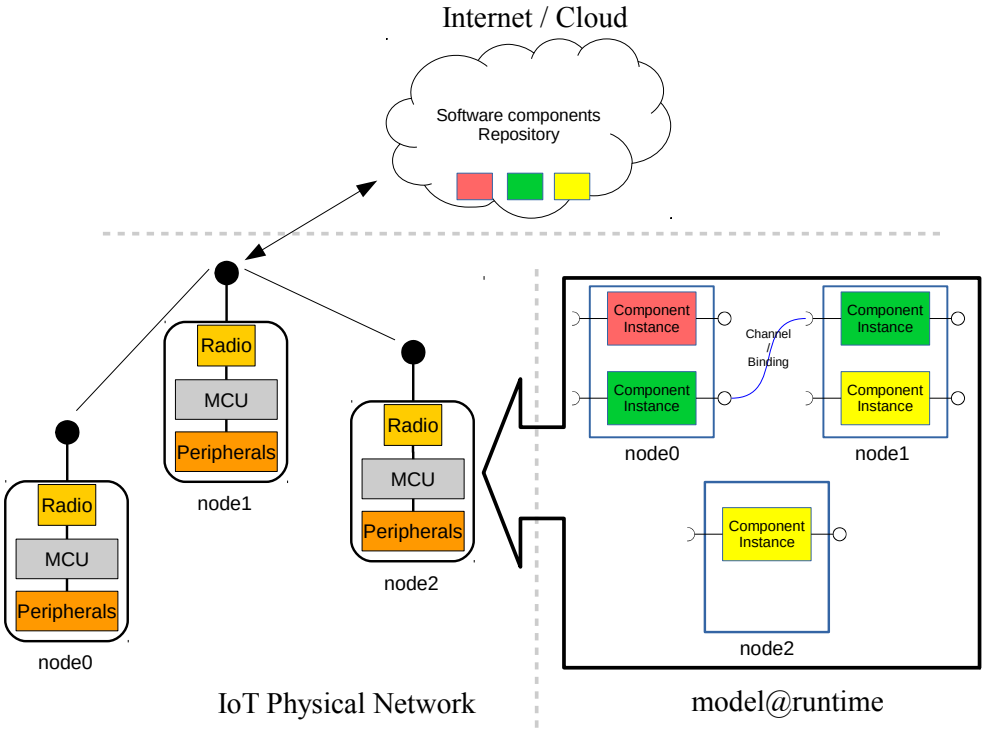


Figure 4.1: *A M@R representation for IoT devices.*

Our goal is then to provide a middleware that will be present on each node of the IoT system, and will take care of the various tasks imposed by the model@runtime paradigm. In figure §4.1, we can see a representation in which three IoT nodes are part of a small network. This example shows how the model is present on the three nodes, each node having the knowledge about three available components on the repository, the instances present in the other two nodes and a binding through a channel between two components of different nodes. Moreover, any change of this configuration should be reflected on the model, followed by the dissemination of these changes to the other nodes, and vice-versa, any change on the model will affect the actual node. Figure §4.3 describes the actions taken when a reconfiguration or adaptation is triggered.

Thus, we need to use only the previously described mechanisms provided by this approach, and adapt the selected features to address the node's hardware constraints. The next Subsection will take into consideration the minimal requirements towards the design of a new M@R implementation for IoT nodes, inspired from the Kevoree approach.

4.2.1 Minimal Kevoree properties needed on the IoT

As already discussed in section §3.3.5, the Kevoree meta-model can be easily transformed into Java code using a modelling framework. However, in our case this approach cannot be used. This is due to two reasons: first, we cannot use high-level languages such as Java and second, we are limited to the C language which can be compiled for the IoT node's architecture already presented in §2.3.1. Moreover, meta-models follow very often an object-oriented approach, which is not defined for procedural languages such as C. Thus, a direct transformation taking into account these constraints is not worth considering. However, a similar approach for code generation and modelling tools can be proposed to meet the requirements of a model@runtime approach. Indeed, the Kevoree Modelling Framework [46] can be adapted to support the generation of C code, in order to provide a fully Kevoree-like middleware for IoT devices. Even so, the efforts to adapt such a framework raise more and different challenges which are out of the scope for this thesis. Since our goal is, first of all, to investigate the limits of an IoT node in terms of memory, a manual implementation (transformation) of the meta-model and its modelling tools is then needed, by adapting the concepts to the limited resources of the node. Indeed, this allows a rapid prototype which can be finely tuned to meet the memory constraints present in IoT devices, in contrast to a more generic code generation approach.

On the typical implementation of Kevoree, a *core* application is embedded in every node. This application provides to each system element (node, component, communication channel, groups) an access to the current model, allowing to submit new configurations through new models. If this *Kevoree Core* receives a new model, it is in charge of the following actions:

- **Model validation.** Verify if the serialized model actually corresponds to a valid Kevoree model, by parsing it and proceeding to deserialization.
- **Adaptation planning.** Once the model is deserialized and loaded in memory a model comparison is performed. A list of differences is then generated, describing the actual adaptations to be performed, which may contain adding/removing components and modifications to parameters (reconfiguration).
- **Adaptations execution.** Following the adaptation plan, the adaptations (component deployment, reconfiguration) will take place, in the following order:
  i. **Stop Instance**. A running instance will be stopped.
  ii. **Remove Binding.** If a binding between 2 channels of a component exists, it will be removed.
  iii. **Remove Instance.** A component instance will be removed.
  iv. **Remove Deploy Unit.** A deploy unit will be removed.
  v. **Update Binding.** When a binding between channels exists, it will be changed to the new channels.
  vi. **Update Deploy Unit.** An existing deploy unit gets replaced by a new one (most recent version).

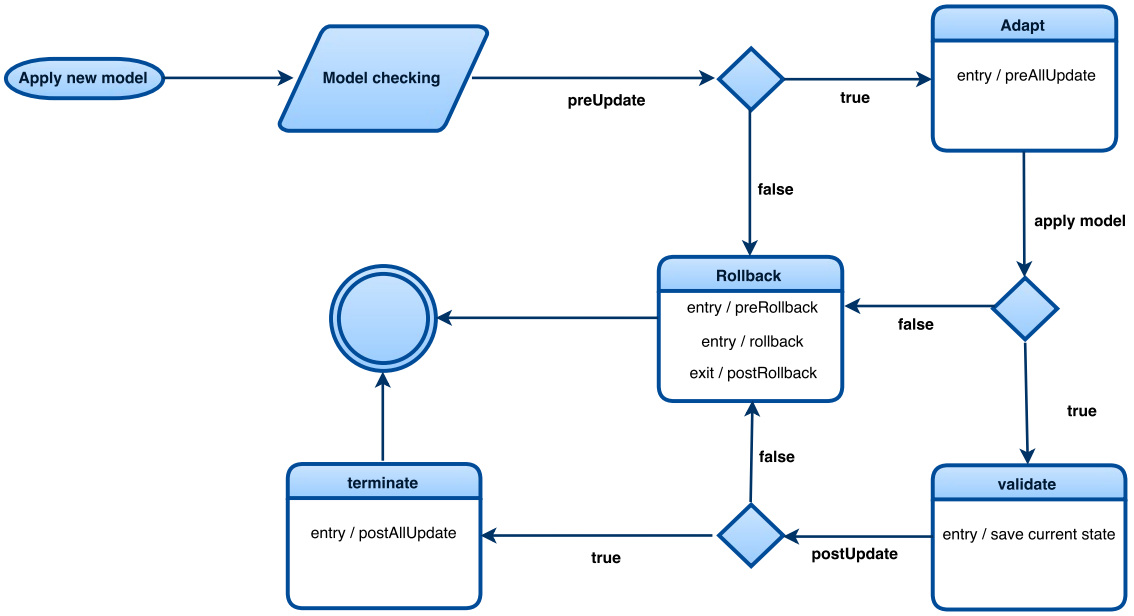


Figure 4.2: *State-transition diagram showing the adaptation process in a typical Kevoree implementation.*

vii. **Add Deploy Unit.** If a new deploy unit is needed, it will be downloaded and made it available for instantiation.

viii. **Add Instance.** An instance will be created from an existing Deploy Unit.

ix. **Add Binding.** A new binding between two channels will be created.

x. **Update Dictionary Instance.** If a dictionary entry exists for an instance, it will be updated with a new value.

xi. **Update Instance.** An existing instance will be updated with a new version.

xii. **Start Instance.** An existing instance which was stopped will be started.

This order will avoid any inconsistency while performing the adaptations.

The model validation is delegated to any system element registered as a *listener*. Each component, communication channel or node can make use of an interface and be registered on the *core* in charge of the model management. Once registered, the instance is notified with regard to the different reconfiguration stages. Figure §4.2 represents in a state-transition diagram the integration of *listeners* with the process of a typical implementation of Kevoree.

In contrast, our design for IoT devices cannot follow the same algorithm. Indeed, the model checking and rollback mechanisms require to save the entire model in memory for checking and, if something goes wrong, to bring back the previous model. This is very memory consuming for our application, thus a typical IoT device would not be able

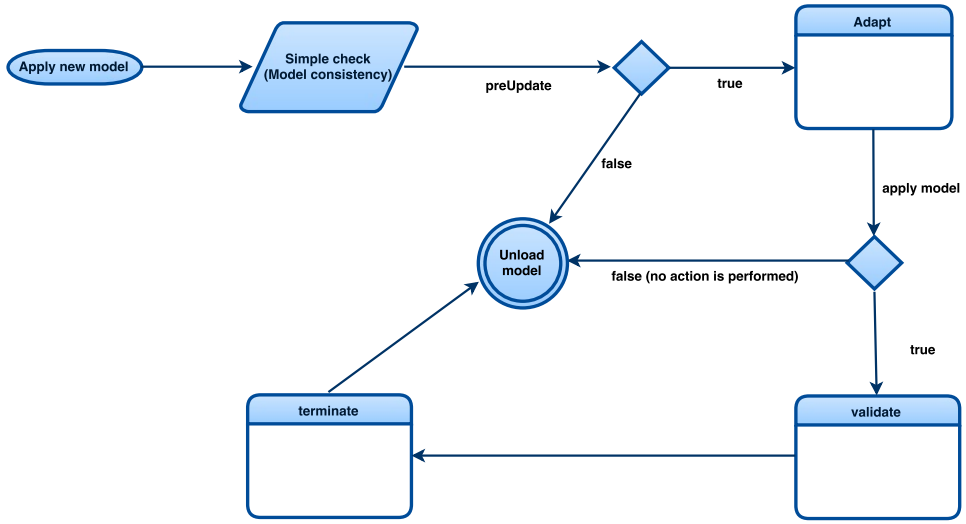


Figure 4.3: *State-transition diagram showing the adaptation process for IoT devices.*

to keep several models in RAM for the rollback mechanism. Therefore, only a simple model checking followed by the adaptations execution has been proposed. Indeed, in our proposition the node is in charge of the adaptation mechanisms, since for our concerns (an IoT system) nodes are the main component and it can only execute an instance of itself. This vision contrast with a typical Kevoree implementation, where various *Kevoree Core* can run on a single machine, resulting in several nodes reflected in the model.

In figure §4.3 the proposed reconfiguration algorithm is presented, which is able to modify the model retrieved from a node. The set of adaptation stages mentioned above are then carried through a transactional manner.

After describing the main design issues and challenges, we can highlight these in figure §4.4. Indeed, a models@runtime implementation is not straightforward, and needs special attention to meet the constraints already discussed previously. In summary, as depicted on figure §4.4, we have two aspects which result in a complete M@R implementation: the Kevoree meta-model, which describes the model representation of the current system, as well as the component model, and finally the M@R engine which manipulates this model. This engine should meet all the constraints present in an IoT device, and at the same time to provide the main functionalities to receive, compare and generate a list of found differences (traces) between the current model and a (modified) new one. Moreover, in order to receive and send models through the network, a serializer and de-serializer is needed, using a JSON format. In addition, a model compressor and an adaptation of Deluge [60] as a dissemination tool, already discussed in Subsection §4.3 are also needed. In conclusion, our middleware would be able to provide a list of needed adaptations, based on predefined adaptation primitives, such as add/remove components, change dictionary entries (parameters) and stop or start component

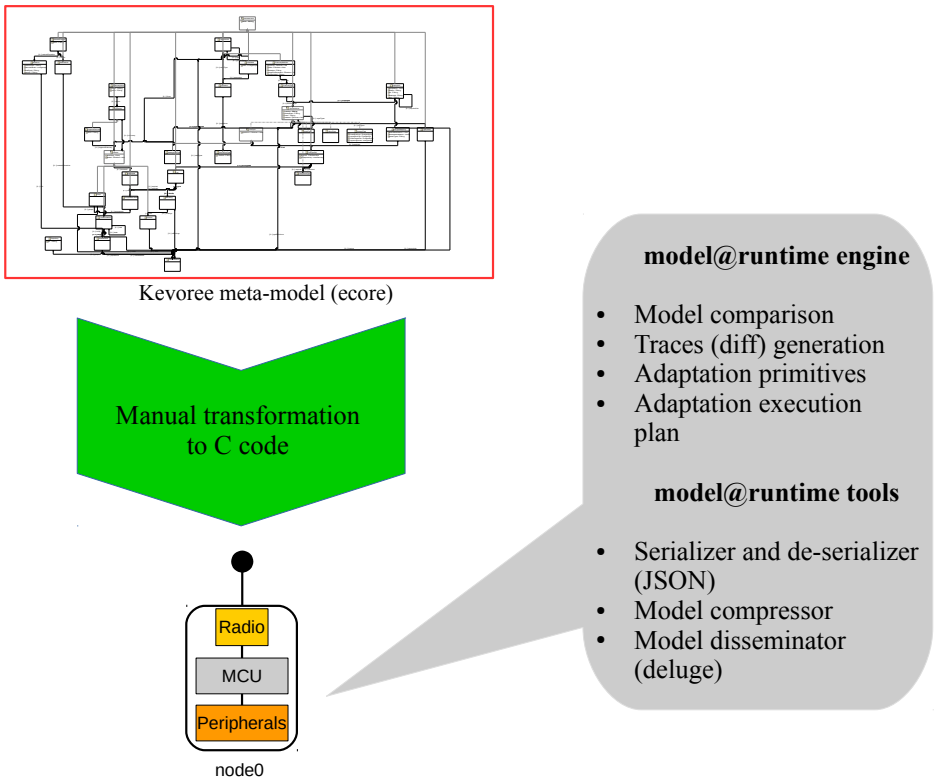


Figure 4.4: *The two different needs for a complete IoT models@runtime engine.*

instances. Finally, an ordered plan to execute such adaptations is required for the actual system, which will perform the actions. Therefore, we can now separate our approach into two different challenges: the first one, which is the meta-model and modelling engine implementations, and the second one, the system facilities development to enact the adaptations provided by the M@R engine.

We need thus to have a clear view of the requirements while implementing such middleware, especially at the system level. Indeed, the underlying system should provide all the needed features on which the M@R design relies. The next Subsection will discuss these requirements in order to find an existing software platform that fits for our design.

4.2.2 Kevoree software implementation requirements

Our middleware approach will need several features from the underlying system, in order to match with the high-level description provided by the model@runtime. As we discussed on our state of the art in Section §3.4.2, the most common approach used to run applications on IoT devices is bare-metal development followed by firmware flashing into the ROM memory. Even if this method allows a fine control of the underlying

hardware, the development time can be very long and difficult to debug, since abstractions are mostly done only at the hardware level, and does not come to the system level. Moreover, since complexity grows, applications for IoT should be developed without special attention to hardware and system concerns. Thus, the use of an IoT operating system seems to be the way to go, in order to take advantage of its system-level abstractions.

As presented in the state of the art, several IoT operating systems exist, but according to the needed features some of them are more convenient. We can thus establish the requirements as following, according to OS properties:

i. **Basic OS functionalities such as timers, task scheduler and Inter Process Communication (IPC).** Basic functionalities that are the core of any embedded OS.

ii. **Dynamic linker and loader, following the third approach presented in section §3.4.2.** A dynamic linker and loader is essential to our approach, since changes in the model containing new software components will trigger the download and instantiation of an artefact, which should be linked and loaded at runtime by the underlying system.

iii. **Network stack implementing basic IP functions (TCP, UDP, HTTP, CoAP).** In order to share a model and download software artefacts, IP communication is mandatory, since the goal is to use the Internet to reach component repositories from different sources.

iv. **Persistent data handling, preferably a file system.** The method used to avoid RAM storage is to serialize the model@runtime in a file on the flash memory, usually in the JSON format. Thus, a way to store and access this JSON file for reading and writing should be provided.

v. **Abstractions for attached devices (LEDs, sensors, actuators, etc.).** Although not mandatory, an OS usually carry basic hardware abstractions, providing an easy way to develop applications which need a physical interaction with the external world.

In order to make a rapid functional prototype of our middleware, we will make use of the Contiki OS which, given the features presented above, seems to fit our minimal requirements. Despite the programming model already described as a drawback, Contiki offers all the needed functionalities, as well as a wide community which collaborate very actively in the development and debug of it. Moreover, Contiki includes an implementation of 6loWPAN [76], an adaptation layer for an IPv6 compressed stack, which let us assign directly an IPv6 address to the device. This enables a ready to use IoT environment. Afterwards, an UDP transport layer is provided, allowing a standard way to reach UDP servers to download the needed deploy units to perform system adaptations, according to the model@runtime engine. Indeed, one of the most important features provided by this OS is the dynamic code loading mechanisms. These are based on a dynamic linker and loader that use the standard ELF object file format [31].

4.3 Networking aspects on IoT environments

Regarding the *inter-node* challenge, another concern while adapting the selected features of the Kevoree M@R implementation is the amount of energy needed to run it. Indeed, an IoT device running on batteries should be able to embed this middleware without a high overhead in terms of energy consumption. Since the most processor consuming tasks are model checking and adaptations execution, a smaller overhead is induced, compared to the traditional approach, thanks to the simplification of such process. However, it was discussed that the most energy consuming task for an IoT device is the radio communication. This rises new and different challenges while interconnecting our nodes running the adapted Kevoree implementation, since the traditional Kevoree implementation was done having in mind no restrictions for network usage and bandwidth, thus algorithms such as Gossip [44] were used for model dissemination. The implementation of such a protocol in an IoT device could be very memory consuming, in addition to a high-energy consumption while using the network.

Given the energy and memory constraints, a dissemination protocol adapted for IoT devices is then needed. It results interesting that the already described Deluge protocol [60] seems to be a good option, since it was developed having in mind the constraints of an IoT device. Even if it was presented as a bad choice while performing entire firmware transmissions, the model information to be shared is not as big as an entire firmware, but rather a small serialized file, thus the needed energy to disseminate a model is actually very low.

A last concern should be taken into account while performing adaptations. We observed that the adaptation process would need to download new deploy units to be instantiated according to the new requirements, or when an update is needed. Indeed, the traditional Kevoree approach will download all needed software artefacts from a registered repository, without any care about the network topology. However, since the nodes on the IoT can form very different and complex network topologies (multi-hop mesh, stars), we need to analyse several strategies to reduce network traffic as much as possible. Therefore, the need of a more complex technique to download software artefacts appears, in order to reduce the inherent energy consumption while performing this task.

4.4 Summary

At this point, we can highlight the main requirements and functionalities needed from a typical IoT device, in order to run a models@runtime implementation. We can now observe that the challenges are concentrated on the internal behaviour of an IoT node, thus focusing only in providing an *intra-node* implementation. This intra-node view will guide us to put our efforts into a first proposal of the Kevoree-IoT middleware, taking into account all the design requirements exposed throughout this chapter.

It is important to notice and remember that our environment is very constrained,

first in the intra-node perspective but also in the inter-node mechanisms. Indeed, we can remember our challenges as following:

i. **Intra-node challenges:** Memory, processing, storing and programming environment constraints.

ii. **Inter-node challenges:** Networking, thus communication, which impacts primary the energy consumption.

The next chapter will provide the implementation details of our intra-node design specifications, followed by an evaluation of the possibilities and limitations of our approach. Since our goal is to provide a middleware that works on real platforms, our evaluations were conducted, first, on an especially designed hardware platform. Moreover, once our first tests are analysed, we followed our research goals by testing our implementation on a large-scale testbed. Indeed, this last evaluation will trigger the actual challenges for the inter-node needed mechanisms, which will be presented in the chapter afterwards.

Chapter 5

Intra-node challenges: middleware implementation and evaluation

As we discussed on the previous chapter, the presented design of our middleware should be able to represent a running system in the form of a model@runtime, according to the Kevoree meta-model. Indeed, this representation can be manipulated by the existing Kevoree editor, on which we can modify parameters, add or remove nodes and components or bind component's ports using channels. Therefore, we focus on the intra-node challenges first, by implementing our middleware on real hardware, leveraging our designed platform and existing OS facilities.

In this chapter, we will first explain how the model@runtime can be represented on the IoT device limited memory, as well as the model analysis and manipulation. Indeed, as we stated previously, our middleware should also be able to interoperate with the existing Kevoree implementations, in order to achieve interaction with other participants on the internet, such as cloud services, existing distributed applications or simply data mining on remote servers and clients. Therefore, the standard Kevoree model representation, in the form of a serialized JSON format, is a must-have for our implementation. This rises several challenges in how a *"big"* (to be saved RAM) representation of the model can be stored and accessed on our constrained nodes. We will present how we cope with this challenge, from the representation of objects in a procedural language such as C, to the manipulation techniques used to compare and execute the actions described on the models.

Experiments were conducted to evaluate the functionalities and the scalability of our approach, and the results are presented at the end of the chapter. First, a generic implementation is tested on a single *"big"* node, followed by scalability experiments on a typical node present in a large-scale testbed. Our main goal is to state the minimal functionalities at the model level, from which we can then implement the needed mechanisms to achieve real reconfigurations and adaptations described in the new upcoming models.

5.1 An empirical study on constrained IoT devices

The model@runtime paradigm has been mainly investigated in the context of distributed systems. These research efforts have been focused on the provision of a comprehensive set of tools to easily deploy, dynamically reconfigure, and disseminate software on a set of distributed computing units. The current model@runtime tools have been implemented regardless of the specific characteristics and constraints of IoT devices. In particular, the network topology and the resource constraints of the nodes forming the distributed system have not been taken into consideration. As a result, state of the art model@runtime tools are not suitable to be used in the context of IoT Systems.

In [45] μ-Kevoree, the closest effort to port the model@runtime paradigm on the constraints of a Cyber Physical System (CPS) was presented, in which the underlying device is comparable to an IoT device. Despite the particular attention given to the specific constraints of a Cyber Physical System, this work heavily relies on over the air firmware flashing to support the deployment and reconfiguration of software. We consider that relying on firmware flashing to support software deployment constitutes a flaw in the approach because of its energy cost (the complete firmware has to be sent, and if any error occurs, the whole process is restarted). A second limitation of this approach lies in the fact that each resource constrained node relies on a more powerful node to perform most of its tasks related to the dynamic reconfiguration (firmware synthesis, reconfiguration decision and so on). This second limitation is not suitable in the context of a system mainly composed by resource constrained nodes since all these nodes have to be managed by bigger nodes. Pushing this idea further, the management of a CPS composed of a wide number of resource constrained devices and a bigger node, the latter will have to manage all the smaller devices in a centralized management scheme.

The next section will describe a more complete M@R engine implementation based on the Kevoree meta-model, together with some tools which allow model manipulation.

5.1.1 Kevoree-IoT: Estimating needed resources

In contrast with most of the current implementations of the M@R paradigm, which are intended for high resources machines thus they make use of high-level programming languages, our first implementation should follow a procedural language such as C. We can justify this by the fact that most of the open source compilers for IoT devices support only this programming language, in addition to C++. However, C++ applications are difficult to integrate in OSs like Contiki or RIOT, which are convenient OS able to run our middleware. Thus, a first approach has been developed in plain C[†1] following the directions presented in Section §4.2.

Indeed, our efforts were put into a first manual implementation of the Kevoree meta-model. The size of the test application which contains a group and a node without components is of 181997 bytes, while the needed RAM before the execution is of 1616 bytes.

[†1] https://github.com/kYc0o/kevoree-c

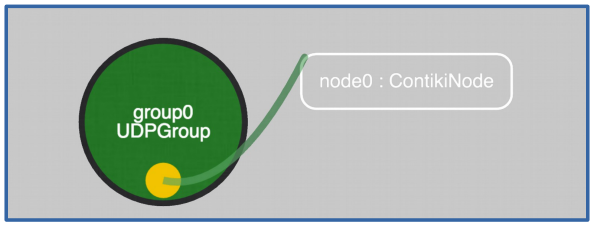


Figure 5.1: *Graphical representation of a minimal M@R.*

A graphical representation of the model can be obtained through the Kevoree Web Editor[†2], in which a serialized JSON file containing the model can be loaded for edition. This representation is shown in figure §5.1.

Then, it is necessary to characterize the minimum system requirements for an IoT device to implement a full model@runtime middleware, as in hardware capabilities as in execution environment. This will avoid the use of third party nodes to perform the high-level tasks described above. However, even if a proposed device meet the characteristics to perform such tasks, the trade-offs between energy consumption and the middleware execution should be taken into account. A general description of needed features to execute the proposed middleware is given in the next subsection.

5.1.2 Handling hardware constraints

Considering the challenges presented in Section §4.2, we must test our implementation on real hardware platforms in order to measure the limitation of our approach, as well as its compatibility with the other implementations (Java, Cloud, Android). Moreover, the energy constraint of each IoT node, together with the mesh topology of the network makes communication in these environments fairly reliable. This implies the necessity to optimize the way the model and the software are disseminated on the network.

At first glance, our implementation was compiled for three different platforms (*class 1 [11]*): the ***zolertia Z1***[†3]***, redbee econotag***[†4] ***and wismote***[†5], which were widely used for research purposes in the IoT domain. Unfortunately, none of these platforms was able to meet the space requirements in ROM and RAM to run a minimal model at runtime as shown in figure §5.1. However, the existence of more powerful microcontrollers, an essential part of an IoT device, shows that typical IoT applications have not exceeded the resources of existing experimentation platforms. Indeed, only some development kits

[†2] http://editor.kevoree.org/v4/
[†3] http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf
[†4] http://redwire.myshopify.com/products/econotag-ii
[†5] http://wismote.org/doku.php

such as TI's CC2538DK[†6] were available at that time, but its high cost, low availability and poor support gave low acceptance for research purposes. Thus, the design and construction of a new experimentation platform seemed to be the fastest way to obtain the first results of the implementation, by integrating the latest microcontrollers and peripherals available on the market. It is worth to consider that this platform was conceived for research and experimentation purposes, and not as an end-user device.

An exhaustive search was conducted to find the latest technologies to build an experimentation IoT device, which fits the requirements already estimated in section §5.1.1. The first step was to find the basic components of such a device. Taking into account the requirements from section §4.2.2, an IoT device should, more specifically, embed:

i. **Microcontroller.** A low-power microcontroller unit is needed to perform the computational processing tasks as well as to store the program code. A huge amount of ROM and RAM is needed at the scale of these devices; thus, it is necessary to find a microcontroller with a good trade-off between energy consumption and memory size.

ii. **Communication interface.** Communication is the main task that will perform our device, since it is mandatory for IoT environments. A radio interface implementing the widely-used IEEE 802.1.5.4 standard [1] is then necessary to communicate in an interoperable way with other devices. An ultra-low energy consumption is also important, since communication is the most energy consuming task.

iii. **External flash memory.** Storage for data is a very useful feature for an IoT device. It can serve as persistent data storage for logging and data collection from sensors, as well as updates and eventually new firmwares.

iv. **Sensors and actuators.** In order to collect experimentation data, some sensors should be embedded on the device, the most common being temperature, humidity and position. Indicator LEDs are also commonly present in these devices, to provide visual signs such as function status (ON/OFF) or current communication.

v. **Ports for external devices.** An easy way to connect third party devices allows a dynamic behaviour, since we can connect other sensors, actuators and devices which cannot be present internally on the device. Such devices should be connected using standard interfaces, such as I2C and SPI. General Purpose Inputs/Outputs (GPIO) are also widely used, either as digital or analogue interfaces.

vi. **Battery capabilities.** An IoT device is very often used in environments where a constant power source is not present. Thus, work on batteries is a very useful feature to test energy consumption while adding flexibility of placement.

Given these features, it was then necessary to integrate several components in order to build the required IoT device. We put special attention to the most critical components, which are the microcontroller, the external flash memory and the battery controller. Regarding the radio transceiver and the other peripherals, there are no huge differences

[†6] http://www.ti.com/tool/cc2538dk

Microcontroller	Speed (MHz)	RAM (KB)	ROM (KB)	Consumption at max. speed (mA)
STM32F0	48	32	256	22
STM32F1	72	96	1024	68
STM32F3	72	80	512	61.5
STM32F2	120	128	1024	49
STM32F4	180	256	2048	98

Table 5.1: *Comparison between STM32F microcontrollers.*

between the most commonly used, for instance, the ones used in the previously analysed devices. The specific conception of the platform will be described in the next subsection.

5.1.3 Towards a new IoT device

Several low-power microcontroller architectures were available at the time of our first M@R implementation, such as TI MSP430, Atmel AVR, Microchip PIC and ARM Cortex-M, just to mention the most common ones. Since our middleware minimum memory requirements were tested on 16-bit MSP430 microcontrollers (Zolertia's Z1 and Arago's WiseMote), finding a huge scarcity of memory, 32-bit architectures were the target of our scope. Indeed, ARM Cortex-M microcontrollers offer only 32-bit RISC architectures, which are widely used both in research and industry. Thus, the first step was to select an ARM Cortex-M microcontroller fitting our memory and energy consumption requirements. The ST Microelectronics STM32 family of microcontrollers offers a wide range of devices with different processor speeds and memory sizes.

Since our goal is to build a device on which our experimentations could be executed on a more flexible way (without caring about memory requirements), it is preferable to use a microcontroller featuring the highest memory capabilities, both in ROM and RAM, over energy consumption. Moreover, this kind of devices are able to change the processor speed as needed, thus reducing the energy consumption. Indeed, our choice was a STM32F4 microcontroller, which was used as the core of our IoT device. Table §5.1 shows a comparison between the available microcontrollers, featuring the maximum processor speed, RAM and ROM, followed by the current consumption in such configuration.

As for the power scheme, given the capabilities of the selected microcontroller, a power source between 1.8V and 3.6V is required. Indeed, a first approach to power the device is the use of an USB port, which delivers around 5V. Thus, a 5V to 3.3V (the recommended tension) is required. In contrast, when the device is needed to run on batteries, the supplied voltage will change. A couple of AA batteries connected in series is the most standard array to power IoT devices. However, a wide set of peripherals

	Speed (MHz)	RAM (KB)	ROM (KB)	External flash (MB)	Radio transceiver	Peripherals	Embedded Sensors
Zolertia Z1	16	8	96	2	CC2420	UART, I2C Phidgets (USB only), GPIO	3 axis acc. and temperature
Arago's WisMote	25	16	256	8	CC2520	UART, I2C, Phidgets, GPIO	3 axis acc. light and temperature
Redbee econotag	26	96	N/A (device run from RAM)	128 (KB)	Integrated (SoC)	GPIO	N/A
DiverSE Board	180	256	2048	16	CC2520	UART (x2), I2C Phidgets, GPIO	N/A

Table 5.2: *IoT Platforms comparison.*

such as sensors and actuators work very often at 5V. Therefore, it was necessary to add a DC to DC converter, coupled with an automatic power selector which detects when the device is powered either by USB or batteries. This allows to use any power scheme without decreasing performance or peripherals compatibility.

Thus far, the main required features for our device are met. A comparison between the previously tested platforms and ours is given in table §5.2, showing the most common useful features. Moreover, figure §5.2 shows the layout of the new board, on which we can observe the different components and the size of the PCB. Indeed, its measures are 3.5cm x 4.5cm.

Given the features of our new IoT device, the next step is to provide the hardware abstraction layer and a network stack to integrate it into an IoT network. Indeed, several IoT operating systems existed at the time this device was developed, thus leveraging one of these available OS was the most recommended procedure. It is important to notice that our middleware implementation it's independent of the underlying OS, since it only provides abstractions for the representation of the running system in the form of a model@runtime. Thus, it can be adapted to any other OS able to integrate modules written on C, as well as networking facilities typical of the IoT to disseminate such model.

5.2 The new Kevoree-IoT Contiki implementation

In this section, we present our initial results towards the design of a middleware which will offer the functionalities of model@runtime over the previously conceived IoT

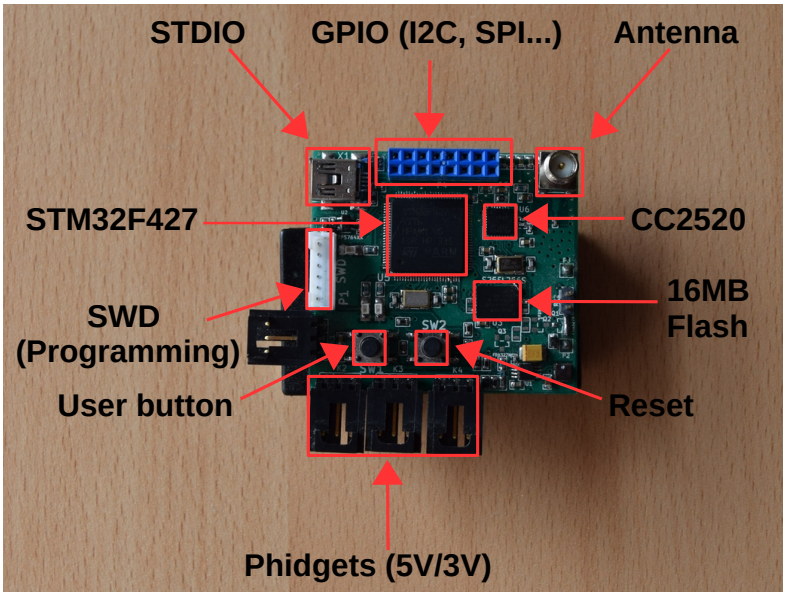


Figure 5.2: *The new designed board to meet our needs..*

device. This middleware, which we called Kevoree-IoT (in reference to the Kevoree-like M@R implementation of our middleware), will provide a development framework for applications on top of these systems, and will provide the runtime infrastructure to support the deployment and dynamic reconfiguration of these applications.

We based our implementation on an already ported Contiki version[†7] for a very similar device than ours, which provided a basic hardware abstraction layer for the radio interface and some I/O drivers. The rest of the drivers were developed according to our platform hardware design. Once the porting completed the next step was to test our middleware on such platform. To do that, a Kevoree-IoT application should be developed, using this new Contiki port.

Based on the Kevoree-C implementation introduced in subsection §5.1.1, the needed Contiki application was developed and tested in our platform. The first implementation[†8] was done using the version 4 of the Kevoree meta-model. This meta-model has been mapped into C code as a Contiki application. We were able to add nodes, components, groups and channels, and bind them through the Kevoree editor by loading a generated JSON file.

[†7] https://github.com/vedderb/contiki
[†8] https://github.com/kYc0o/kevoree-contiki

5.2.1 An object-oriented representation using C

As we discussed previously on Subsection §4.2.1, special attention must be put while transforming a meta-model into a procedural language such C. Indeed, this programming language does not have an object-oriented design, thus a representation of classes, their properties and relationships from the Kevoree meta-model should be proposed.

We took as a source of inspiration the existing Kevoree implementation for C++[†9], which is the closest language that match with C. In this implementation, the classes and its relationships are transformed into C++ code directly, since it supports object-oriented representations. We can observe four components in the Kevoree meta-model:

- **Classes.** This is the abstract representation of an object type. It can be instantiated and contains properties and relationships with other classes.
- **Properties.** Values that can have the form of a primitive type (integer, character, boolean, etc.), depending on the types supported by the language. Special properties called methods are also part of a class, which provide a specific functionality when the method is called.
- **Relationships.** These are the connections between two or more classes. We can note two types: a reference and a containment reference.
- **Inheritance.** A class can inherit the previously described components from another one, besides adding its own.

Thus, implementing this representation lead our efforts to optimize the resulting code, while respecting the previously described composition. Indeed, the first step was to represent a class as a C data structure. While doing it, some decisions were made in order to cope with the intra-node constraints:

- Primitive types such as integers and boolean keep the same size as in the meta-model.
- String types are of a fixed size, to avoid dynamic memory allocation thus eventual fragmentation.
- Methods are represented as function pointers, in a separated structure called *Virtual Table*, allowing easy inheritance.
- Properties inheritance is achieved by copying the parent's properties (including and respecting all the inheritance hierarchy), followed by a pointer to the virtual table. This way we can reuse code and ensure polymorphism.
- References are represented as a pointer to the referenced structure, while containment is achieved using a minimalistic implementation of a hashmap.

[†9] https://github.com/kevoree/kevoree-cpp

```
typedef char* (*fptrKMFMetaClassName)(void*);
typedef char* (*fptrKMFInternalGetKey)(void*);
typedef char* (*fptrKMFGetPath)(void*);
typedef void (*fptrVisit)(void*, char*, fptrVisitAction, fptrVisitActionRef, bool);
typedef void* (*fptrFindByPath)(void*, char*);
typedef void (*fptrDelete)(void*);

typedef struct _KMFContainer_VT {
void *super;
/*
* KMFContainer_VT
*/
fptrKMFMetaClassName metaClassName;
fptrKMFInternalGetKey internalGetKey;
fptrKMFGetPath getPath;
fptrVisit visit;
fptrFindByPath findByPath;
fptrDelete delete;
} KMFContainer_VT;

typedef struct _KMFContainer {
    KMFContainer_VT *VT;
    /*
    * KMFContainer
    */
    KMFContainer *eContainer;
} KMFContainer;
```

Listing 5.1: KMFContainer: the main container on Kevoree

```
typedef struct _NamedElement_VT {
    KMFContainer_VT *super;
    /*
    * KMFContainer
    * NamedElement
    */
    fptrKMFMetaClassName metaClassName;
    fptrKMFInternalGetKey internalGetKey;
    fptrKMFGetPath getPath;
    fptrVisit visit;
    fptrFindByPath findByPath;
    fptrDelete delete;
} NamedElement_VT;

typedef struct _NamedElement {
    NamedElement_VT *VT;
    /*
    * KMFContainer
    */
    KMFContainer *eContainer;
    /*
    * NamedElement
    */
    char name[16];
} NamedElement;
```

Listing 5.2: NamedElement class representation inheriting from KMFContainer

We can observe on listing §5.1 the representation of the main Kevoree class *KMFContainer* which includes only a pointer to its container and a pointer to the Virtual Table containing the function pointers to its methods. As an example, the class *NamedElement* in listing §5.2 shows how inheritance is achieved, by copying the property of the parent class. On the other hand, method inheritance is achieved by copying all the function

pointers into its own virtual table. Moreover, to access parent's method implementation, the virtual table contains also a pointer to its parent (super class).

Once a representation of the basic requirements is implemented, we need to provide mechanisms to represent an instantiated model using the data structures described previously. The next Subsection describes briefly how the process of serialization and de-serialization takes place.

5.2.2 Model manipulation challenges

While representing an instantiated model in memory, we need to allocate enough place for the required nodes, components and connections that are described in a serialized form. This starts by a model de-serialization, which consists in the transfer of a JSON file among the nodes in the network, followed by the model parsing and execution of the adaptations and reconfigurations described on it. Due to memory limitations, a model cannot be loaded entirely, thus mechanisms to partial loading and parsing should be provided. Indeed, a very efficient JSON loader was developed to fit the memory constraints. This loader takes the JSON elements one by one directly from the external flash, thus avoiding the allocation of big memory slots, and freeing the already parsed elements. Once an element is loaded, it is compared with the current status of the system (the running model being de-serialized at the same time), generating a list of differences (traces about changes) which are analysed to create a list of adaptations and reconfigurations.

The execution of the adaptations and reconfigurations will use the features provided by the underlying system, which were already presented in section §4.2.2.

5.3 Firsts evaluations of the approach

Our firsts results showed that our implementation was able to run on our experimentation device. This first firmware contains the Contiki OS kernel, device drivers, a CoAP web server and the Kevoree-IoT middleware. The memory size of this firmware is of 215228 bytes in ROM and only 18404 bytes in RAM.

Once our middleware was running in our experimental platform, we needed to test it at large scale in order to test the model dissemination. This needs a large infrastructure with several devices where a basic experiment using our middleware can be executed. Indeed, the manufacturing of several of our devices was our first option. However, since the manufacturing of the firsts devices was made by hand, to build some other a huge amount of time was needed, and could be very costly. Therefore, a search for large-scale testbeds for the IoT was done. The next section will describe one of the platforms available in a large-scale testbed, showing its main characteristics, which were analysed in order to find if they met our requirements.

5.3.1 Requirements for large-scale evaluation

At the time of our firsts experiments, a testbed existed at the INRIA Rennes centre, where our research team is based. A Wireless Sensor Network called Senslab [25], formed by 256 devices was deployed in a centre's cellar. This WSN offered wireless sensors of similar characteristics as the Z1 platform, which was already described. Thus, the minimum requirements to run our middleware were not met by these WSN's devices. However, an extension of such testbed was done recently, adding new experimentation platforms. This new platform, called FIT IoT-Lab [41], featured new devices of a similar architecture as ours. Indeed, two more powerful nodes were added to the testbed: the M3 node and the A8 node, in order to provide more powerful IoT capabilities.

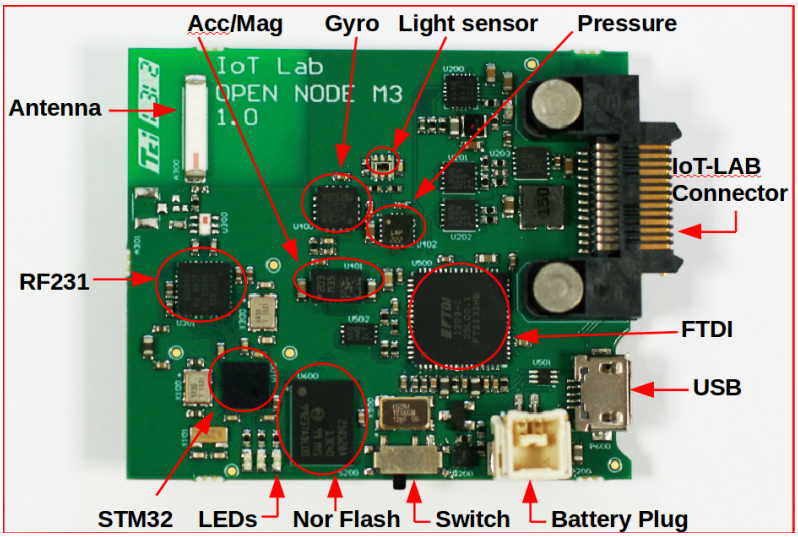


Figure 5.3: *The FIT-IoT Lab M3 Open Node.*

Figure §5.3 presents the M3 Open Node, an IoT device with very similar characteristics of the one from our design. This device features a STM32F1 Cortex-M3 microcontroller, with 512KB of ROM and 64KB of RAM. External sensors are available, as well as 3 LEDs. The communication is handled by an AT86RF231 radio, implementing the standard IEEE 802.15.4, typical of IoT devices. In addition, a Contiki port for this device was already available, facilitating the integration of our existing implementation for Contiki. This combination seems suitable for our experiments, since our firsts results showed that around 200KB of ROM and 16KB of RAM were enough to run a minimal implementation of our middleware, including the kernel, communication stack and peripheral drivers.

The next subsection will give details about the technical contributions provided for the IoT-Lab testbed, in order to have a complete IoT platform able to run our middleware and ready for our large-scale tests. Indeed, testing on a large-scale testbed facilitates the firmware flashing, serial output for debugging and power monitoring, thanks to the

tools provided by IoT-Lab[†10]. Moreover, a Contiki port is maintained and supported by the IoT-Lab team, providing a ready to use environment for our already developed tools. However, two important features lacked in this port: the low-level interface for the Contiki File System (CFS) and the relocation operations needed by the Contiki ELF loader. Therefore, given the importance of these features for the correct implementation of our middleware, it was necessary to extend the testbed capabilities to add support for the CFS and the ELF loader.

5.3.2 Extending the FIT IoT-Lab testbed

As stated previously, two missing features were detected in the Contiki port made for this testbed. The first one was the implementation of a low-level interface necessary for the Contiki File System. This driver make use of low-level functions to read, write and erase the flash memory embedded on the M3 node. Therefore, this functionality is necessary to leverage the high-level abstractions provided by the CFS, which are a very useful approach to manipulate files into the flash memory, also needed by the ELF loader.

5.3.2.1 File system driver implementation

The definitions and porting instructions for CFS can be found on the wiki page of the Contiki main repository[†11] as well as in the article introducing this file system in [102]. A brief description of the file system is given as follows:

> *Contiki provides a set of file systems for using various kinds of storage devices in resource-constrained systems. All of these file systems implement a subset of the Contiki File System (CFS) interface, and two of them provide the full functionality: CFS-POSIX and Coffee. CFS-POSIX is used in Contiki platforms that run in native mode. It uses direct calls to the POSIX file API that is provided by the host operating system. Coffee, on the other hand, is primarily aimed at sensor devices that are equipped with flash memories or EEPROM.*

Thus, the usage of the *Coffee* API is the one that should be implemented for the M3 platform. To do so, information about the flash memory embedded on the platform must be mapped into the configuration headers used by Contiki, in order to provide a connection between the physical device and the OS. This information consists in several parameters, such as:

Total size. The actual size of the memory flash.

Sector size. A flash memory is divided by sectors. This information allows the CFS to organize the way the memory will be written.

[†10] https://www.iot-lab.info/tools/
[†11] https://github.com/contiki-os/contiki.git

Page size. This parameter is given by the manufacturer of the flash memory and allows to organize writing cycles in a more efficient manner, avoiding fragmentation.

Coffee start. It describes the actual memory address where Coffee can start to write. It allows to reserve a memory space that cannot be accessible from the API, providing a secure space where we can store read-only data (*i.e.* initialization data, node ID, etc.).

Coffee size. This is the overall memory available for reading/writing using Coffee. It is calculated by the difference between the total size and the Coffee start size.

Coffee name length. With a view on internal memory savings, a limitation on file names is imposed. This will avoid the use of large filenames which have an impact on the memory footprint of the CFS.

Coffee max. open files In the same way as name length, the number of simultaneous open files can increase the memory usage by the file system. Thus, a limitation on this concern is needed.

Afterwards, parameters to configure the internal behaviour of Coffee are needed, in order to provide more or less flexibility on the usage. Those parameters are set on the maximum optimal levels for our platform, in order to give the maximum flexibility as possible.

Once the flash parameters are given, some functions allowing the actual read-/write/erase onto the flash memory must be also mapped in the interface. Thus, functions from the lower driver implementation are needed. Indeed, these are provided by the Hardware Abstraction Layer, already included in the IoT-Lab low-level driver implementation. This HAL is included in a separated repository called *openlab*[†12]. Within this repository, all the low-level drivers providing functions to manipulate every piece of the embedded system are included.

Once the porting of the file system is done, we can take care about the ELF loader. This second technical contribution aims to provide the relocation functions for the embedded ARM Cortex-M3 CPU, used by the Contiki ELF loader in order to find the correct addresses of the symbols provided by the OS. Indeed, the ELF loader is the main feature used to add new modules to the kernel or update its existing features.

5.3.2.2 Runtime address relocation for ARM Cortex-M3 platforms

As presented in section §3.4.2, we have studied the different methods to add new modules into a running IoT device. The method used by the Contiki ELF loader is the relocatable code. Indeed, this approach needs to relocate the temporary addresses given to the unresolved symbols of a new module, in order to get access to the needed functions provided by the OS.

[†12]https://github.com/iot-lab/openlab.git

Since the use of the Contiki ELF loader involves two aspects, a prepared firmware able to load new modules and the new modules themselves, modifications to the compilation routines should be provided for both artefacts As described in the Contiki wiki[†13], a firmware able to load new modules should be prepared as follows:

> *The firmware must be prepared with a symbol table to able to load ELF modules dynamically. This three-step process ensures that all available symbols in the firmware are also visible in the symbol table, along with a pointer to their address.*
>
> *make <firmware-name>*
> *make CORE=<firmware-name> <firmware-name>*
> *make CORE=<firmware-name> <firmware-name>*

Thus, compilation instructions for the `CORE` variable must be provided. Since they are CPU specific, they must be added in the Makefile for our specific platform.

```
ifdef CORE
.PHONY: symbols.c symbols.h
symbols.c:
$(NM) $(CORE) | awk -f $(CONTIKI)/tools/mknmlist > symbols.c
else
symbols.c symbols.h:
cp ${CONTIKI}/tools/empty-symbols.c symbols.c
cp ${CONTIKI}/tools/empty-symbols.h symbols.h
endif
```

Listing 5.3: Compilation settings to create a proper symbol table

Listing §5.3 shows the instructions to create a symbol table for the M3 platform, including all the functions used in the base firmware. These symbols are created using the `arm-none-eabi-nm` command to extract all the symbol's names from the main firmware, and with the aid of a Contiki tool called `mknmlist` a source C file is generated where all used symbols are declared. Finally, this source code is compiled and linked to the main firmware. On the other hand, if the `CORE` variable is not set, a predefined source file without the symbols is copied to be compiled with the main firmware.

As for the creation of new ELF modules, the Contiki dynamic linker and loader [31] depends strongly on the relocation methods supported by the CPU architectures for which the module is being compiled. Since our implementation is based on an ARM architecture, we need to provide the relocation functions for this platform. Indeed, ARM offers the processor-specific definitions in the *ELF for the Application Binary Interface (ABI) for the ARM architecture*[†14] document. This document specifies the way of an ELF binary should be produced by a compiler, including the relocation types. Thus, this information should be used to implement a dynamic linker which will be in charge of the relocation process for unresolved symbols present in the new module. For the ARM architecture, more than 100 relocation types exist, but just a few operations stills relevant.

[†13] https://github.com/contiki-os/contiki/wiki/The-dynamic-loader#Preparing_a_Firmware_for_ELF_Loading
[†14] http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044e/IHI0044E_aaelf.pdf

We will focus in only 3 types of relocation, which were present in most of the examples we compiled. These relocation types are presented in table §5.3.

Code	Name	Type	Class	Operation
2	R_ARM_ABS32	Static	Data	(S + A) — T
10	R_ARM_THM_CALL	Static	Thumb32	((S + A) — T) – P
30	R_ARM_THM_JUMP24	Static	Thumb32	((S + A) — T) – P

Table 5.3: *Relocation types compatible with our loader.*

These three relocation types are the most commonly found in small component examples, although 10 and 30 share the same operation. Indeed, these binary operations are the most important part of our implementation, and were coded on our platform-specific driver for Contiki.

Afterwards, the requirements of an ELF module handled by the Contiki ELF loader are stated on the Contiki's wiki, as following:

> *An ELF file consists of a header followed by a set of sections which typically include at least a section for binary code (`.text`), a section for statically allocated data with pre-assigned values (`.data`), and a section for zero-initialized data (`.bss`). Additionally, each symbol is represented in a symbol table (`.symtab`), and strings are stored in a string table (`.strtab`). For a file to be accepted by Contiki's ELF loader, it must contain at least the sections listed above.*

In order to produce an ELF file compatible with these characteristics, we need to provide the compilation recipes following the method specified by Contiki. Indeed, the Contiki's documentation provide a generic method to produce a Contiki ELF module (CE), as stated below:

> *Contiki's build system includes an easy method to create modules that can be loaded into a running Contiki system. Simply compile the program with the suffix .ce, as shown below. The suffix instructs the make program to compile an ELF file from a C source file. The object module is stripped from unneeded symbols. The .co suffix works similarly, but does keeps unneeded symbols in the ELF file.*

```
cd example/hello-world
make TARGET=sky hello-world.ce
```

This method makes use of Contiki's predefined compiler flags that should produce a compatible ELF file. However, such flags are intended mostly for MSP430 architectures. We have experimented these default flags for the CE compilation and it was found that

several other sections were present in the ELF file. These "extra" sections contained needed information about the module, and were skipped by the ELF loader. Thus, specific compilation mechanisms shown on listing §5.4 have been proposed to solve this problem.

```
CUSTOM_RULE_C_TO_CE = "defined"
%.ce: %.c
@# Requires '-DAUTOSTART_ENABLE' to be loaded
$(CC) $(CFLAGS) $(OPENLAB_INCLUDE_PATH) -DAUTOSTART_ENABLE -c $< -o $*.o
$(GCCPREFIX)-ld -r -T $(OPENLAB)/merge-segments.ld $*.o -o $@
$(STRIP) --strip-unneeded -g -x $@
```

Listing 5.4: Compilation instructions to generate an ARM ELF file.

We can highlight that a special linking phase (triggered by the `-ld` flag) using a linker script was needed to merge the extra sections generated by the standard compilation phase, followed by the stripping mechanism for unneeded symbols (mostly debug symbols). This script consists in merge all `.text`, `.data`, `.rodata` and `.bss` related sections into common, unified ones. This is needed for the ELF parser provided by Contiki, which is intended to only parse the sections mentioned above.

Once the correctness of the ELF file is validated by the loader, a relocation phase is conducted. This makes use of the ELF loader internals, which should be defined by the platform. These steps are described by Contiki as follows:

> *Each CPU architecture in Contiki that supports ELF loading implements a set of architecture-dependent functions. The table §5.4 shows the API that needs to be implemented in order to support ELF loading. These include functions allocate RAM for data (`elfloader_arch_allocate_ram()`), allocate read-only memory (ROM) for code (`elfloader_arch_allocate_rom()`), write to the allocated ROM (`elfloader_arch_write_rom()`), and relocate addresses in the ROM (`elfloader_arch_relocate()`). The relocation information is stored in objects of type `struct elf32_rela`, which is defined below in listing §5.5. In this structure, `r_offset` indicates the location to be relocated, `r_info` specifies the relocation type and symbol index, and `r_addend` is the value to add when relocating an address in `elfloader_arch_relocate()`.*

elfloader_arch_allocate_ram(int size)	Allocate RAM
elfloader_arch_allocate_rom(int size)	Allocate ROM
elfloader_arch_write_rom(int fd, unsigned short textoff, unsigned int size, char *mem)	Program ROM.
elfloader_arch_allocate_relocate(int fd, unsigned sectionoffset, char *sectionaddress, struct elf32_rela *rela, char *addr)	Relocate addresses in ROM

Table 5.4: *The architecture-dependent functions in the ELF loader.*

```
struct elf32_rela {
elf32_addr      r_offset;
```

```
elf32_word      r_info;
elf32_sword     r_addend;
};
```

Listing 5.5: The 32-bit ELF relocation structure

These architecture-dependent functions were implemented for the common openlab platform. Moreover, for the allocation ROM and RAM functions, a pre-allocated space of both was necessary, which was declared in the configuration header for the IoT-Lab M3 platform. The source code is available in the master branch of the Contiki port for the IoT-Lab testbed.

Finally, once all symbols are located in the right place and the code is copied into its respective memory space, an entry point to the new module is provided as a new Contiki process. Indeed, this new feature is available on the list of processes, and is now possible to access it by using events. The Contiki message passing mechanisms are used to access the new provided functionalities, if needed. In a common module loading behaviour, the new process will start automatically as if it was embedded from the beginning.

The next section will describe a first evaluation of our middleware, using as example an application which will receive a new model, make the comparison and then produce the actual changes (adaptation plan) described on the differences between an old and a new proposed model, on top of this already described IoT device.

5.4 Evaluation of Kevoree-IoT on the IoT-Lab testbed

This section presents the experiments conducted to evaluate our framework described in subsection §5.2. The goal of this evaluation is to assess the feasibility of using a model@runtime implementation on IoT devices. In these experiments, we focus on measuring the overhead induced by our middleware, in order to evaluate its overheads in memory usage and in energy consumption.

5.4.1 Experimental overview

As presented in Subsection §4.2.1, our model@runtime design has been divided into two main aspects: the actual model representation (through the Kevoree meta-model) and the model manipulation engine (Kevoree-IoT). Indeed, our first goal is to assess the main functionalities on the model representation and manipulation, since these are the core of our middleware.

Therefore, this evaluation focuses on answering the two following research questions:

RQ1: Does the overhead induced by our models@runtime implementation fit the resources constraints?

RQ2: Is this resulting overhead small enough to allow scalability?

To answer these questions four experiments are performed using the following metrics:

- **Start-up delay:** time needed to load the current model. To measure this time, we evaluate the time in milliseconds until the application is ready to work.
- **Consumption overhead:** amount of energy in joules drawn by the node while running our firmware. This energy is measured using IoT-Lab tools.
- **Memory:** amount of memory used by our model representation and modelling tools. Such memory is measured by comparing both flash and RAM between various firmwares implementing our middleware, and another one without this implementation.

5.4.2 Experimental setup

We compare the performances of two different firmwares, in order to measure the overheads induced by our M@R layer.

- The first firmware consists in a simple *Blink/COAP* application, with the network stack and a CoAP server initialized. This application features a LED that starts blinking from the beginning of the experiment. The CoAP server represents a standard way to communicate with the node, and it's used to control the LED and get information about the sensors.
- The second firmware includes the same functionalities but is being represented by the model at runtime. For this purpose, we add our model@runtime platform in order to get a model reflecting the current state of the *Blink/COAP* application. To do so, a basic model is built from the current system, representing the *Blink/COAP* application as a component instance.

In this experiment the two firmwares are uploaded on an IoT-Lab node and the applications are executed during one minute. To evaluate the overhead of our middleware the two firmwares are compared with respect to:

- memory consumption both in ROM and RAM,
- the energy consumption,
- and the start-up delay.

The results of this experiment are shown in Table §5.5.

As it can be observed from the table, the usage of a model@runtime has a visible overhead on the memory both in ROM and RAM. This overhead is due to the code of

	Memory used		Energy consumption	Start-up delay
	ROM (in bytes)	RAM (in bytes)	Joules	Msec.
Blink + CoAP	79344	13244	9.6	0
Kevoree + Blink + CoAP	112724	15822	9.606	39.1

Table 5.5: *Memory use for the* Blink/COAP *example.*

our middleware for the ROM part and to the model loaded in memory for the RAM part. We consider that this memory overhead is reasonable compared to the benefit of enabling an abstract representation of the current system.

Our approach also impacts the start-up time and causes a very small delay before the application is ready. This delay is measured using timestamps. It is due to the time used by the processor, in order to load a model@runtime from the current application. This delay is considered reasonable as it is very small and it only impacts the initial loading of the application and has no effect during the normal operation.

As shown on table §5.5, the overhead of our framework on the energy consumption is very low. This consumption has been measured using the data generated by the IoT-Lab platform, which includes voltage and power used by the node in an experiment. The energy consumption overhead, shown in Table §5.5, is obtained as a product of the power in watts used by the node while loading the model@runtime, and the time needed before the application is ready and the LED is blinking. The overhead on energy consumption is only due to the extra computing power needed in the start-up phase to load the model in memory.

This overhead evaluation highlights the feasibility of implementing a complete model@runtime middleware on IoT devices. The memory overhead is reasonable and fits with the resource constraints of the IoT nodes. We consider that the critical overhead for such system is the energy consumption, and our results show that it is marginally impacted by our middleware.

5.4.3 Scalability

In this experiment, we evaluate the scalability of our approach by focusing on the memory needed to represent a large model. To do so, we first measure the memory size

without any model loaded in node's memory. The command `size` of the ARM compiler was used to obtain that measure, since this application does not need dynamic memory allocation.

Our goal is to evaluate the biggest size that the model can reach. We progressively increase the model size by augmenting the number of nodes until running out of memory. Three variants of this experiment are run at last:

- with one component per node,
- with two components per node,
- and with three components per node.

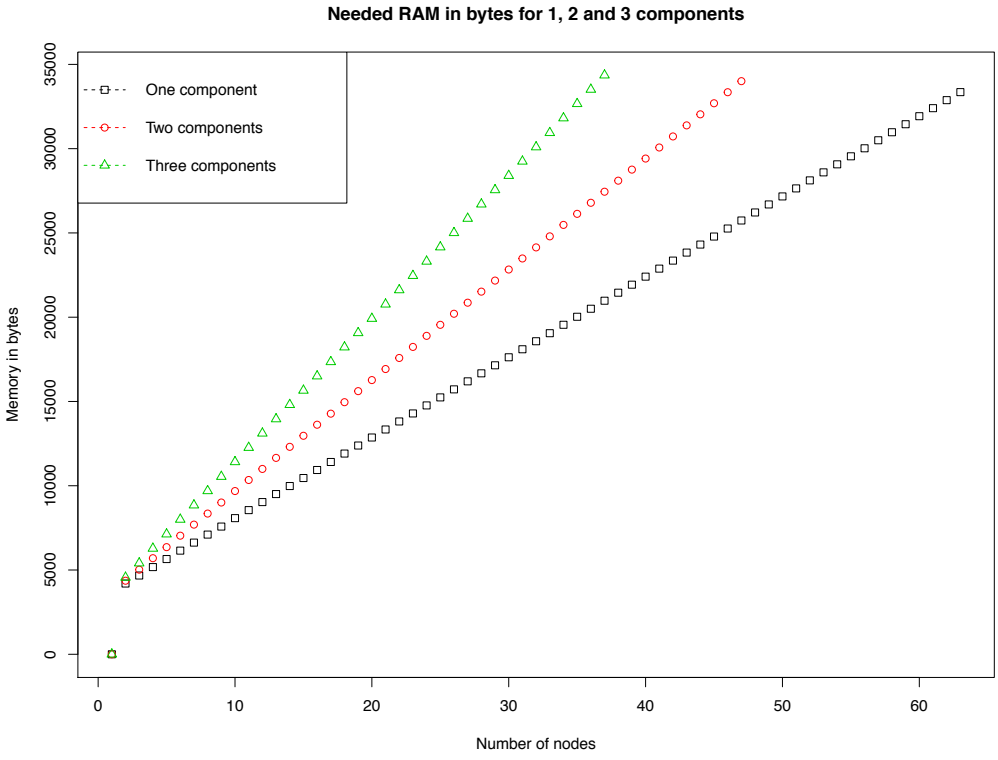


Figure 5.4: *Memory overhead for 1, 2 and 3* Blink/COAP *components.*.

Figure §5.4 shows the memory usage for each model depending on its size. These results show that our current implementation enables to scale the model up to 60 nodes with one component per node, and up to 37 nodes with three components per node. These numbers are encouraging, since some tens of nodes can enable a small IoT local network, such as home, small buildings, cars, factory chains, etc.

5.5 Summary

Our initial results show that the models@runtime implementation is feasible for IoT devices and there are enough resources to deploy several other functional software components. Indeed, we can observe that our overheads are small enough to affect the overall operation of a node, while adding an abstract representation of the running system, in addition to reconfiguration and adaptation possibilities. Since these results are promising, we highlighted that our middleware is able to represent a running application by abstracting it through a component model, without an important overhead, neither in memory nor in energy consumption. The Contiki OS provides most of the functionalities that are mandatory for the good implementation of our models@runtime middleware. As we described in this chapter, two crucial functionalities were not implemented on the IoT-Lab M3 platform: The File System and the ELF Loader. Indeed, without these functionalities the adaptations generated by the model@runtime engine could not be executed. Since the goal of our middleware is to deploy new modules by the means of an ELF file, these contributions are mandatory. Naturally, easy storage and access for this file simplifies the task of loading, by leveraging the abstractions proposed by the Contiki File System. Moreover, by adding the relocation and dynamic linking mechanisms to the M3 platform, real deployment of new features is now possible.

The added support for the two main functionalities needed by our middleware was crucial for the continuation of our research, since it allowed to conduct the evaluation on real nodes. Without them, our experimentations would have stopped at the model level, without enacting the actual changes represented in the model at the system level. The relevancy of our implementations was acknowledged by the FIT IoT-Lab maintainers, who accepted our pull-request[†15] including these technical contributions in their main fork of the Contiki repository.

In order to continue our research goals, we need to review our results and the tools already provided by our middleware. Indeed, several challenges were raised with the results of our first model@runtime implementation. One of these challenges is about the creation of components and their distribution across the network. Indeed, special attention should be put on this inter-node mechanism, since we have argued throughout this thesis that most of the energy consumption is due to radio communication, on battery powered wireless nodes. Therefore, distributing software components on mesh topologies, which are less robust than typical computer networks, raise scientific questions about the best method to distribute them. Thus, a distributed algorithm to download components taking into account the inter-node network topology and energy consumption should be proposed. The next chapter will discuss how these components can be built, and how they can be distributed in an energy-efficient manner.

[†15] https://github.com/iot-lab/contiki/pull/2

Chapter 6

Inter-node challenges: distributing software artefacts

Throughout this thesis, we have noticed three important characteristics of IoT systems: (1) they are distributed, (2) they include resource constrained nodes especially in memory and energy and (3) they are heterogeneous both in hardware infrastructure and role in the network. Indeed, the heterogeneity is an important factor as it encompasses the hardware and software layer: each node composing the network has its own role in it, and thus its specific hardware and software configurations which will evolve over time. As we presented in the introduction of chapter §4, solutions from the domain of Software Engineering can deal with this problem, but cannot be seamlessly adapted to fit with all the characteristics of an IoT system, especially with the resource constrained nodes. Therefore, a first approach is to propose a way to decompose these systems into components, leveraging the benefits argued by CBSE. This was already discussed in the previous chapters, showing that a model@runtime can represent this decomposition in the model. However, decompose this software in components raise the question about their distribution in a typical IoT network.

As a consequence, in this chapter we will address the following scientific problem: **How to *efficiently* distribute, deploy and configure software components for IoT devices?**

Following our initial experiments presented in the chapter §4, our M@R implementation provides an abstraction layer (which is simpler and safer to manipulate than the actual system) to tame the complexity of software adaptation in a distributed system. The software layer is represented as a set of interconnected components which are then deployed and configured on each node. Following this approach, deploying and reconfiguring software requires distributing the code of the specific software components to the targeted nodes. Currently, state of the art approaches to disseminate code over the air are limited in their ability to select specific targets (they disseminate the same binary to every node in the network [60]), and thus waste energy when all nodes do not present a homogeneous software layer. Another approach called FiGaRo [80] propose the use of rules to filter nodes regarding its use or current capacities, in order to select some of

them for an update or new deployment, thus a fine selection of a specific node becomes complicated.

In this chapter, we present an extension on the use of models@runtime to represent both the network and the application layer of the Internet of Things systems. At runtime, we leverage these models through a new algorithm to distribute a software component only to those devices that need it, providing a fine selection by manipulating the model directly. Moreover, this new algorithm aims at minimizing energy consumption during the whole reconfiguration and adaptation step, proposing a distribution mechanism which performs this task in a very efficient manner.

6.1 Inter-node challenges

When changes in the model are disseminated in the IoT system, each node will adapt its local state to the new requirements. Indeed, this adaptation can include the deployment of one or several new components, as depicted in Figure §6.1. Moreover, a given component is not necessarily needed by all nodes in the network. Is in this part where state of the art algorithms to disseminate code over multi-hop WSNs are limited, since their ability to target specific nodes is poor or inexistent. For instance, the Deluge protocol [60] disseminates the same "data pages" of a whole firmware to every node, thus there is no fine selection of the nodes we want to upgrade. In order to deal with this limitation, we propose a new algorithm called *Calpulli*, to efficiently perform the wireless distribution of components to the destination nodes. We take advantage of *(a)* the routing properties of the running system and *(b)* the information provided by the model to find the best strategy of components downloading.

We can summarize the contributions of this chapter as follows:

- **Leverage the already described modelling tools to configure and re-configure IoT systems.** This approach provides a view of a running Internet of Things system on its current state and an efficient way of reconfiguring the software layer of these systems.

- **An energy efficient algorithm to disseminate software components.** This algorithm leverages the network topology and the information contained in the model to decide locally on the best way to disseminate software components.

- **An evaluation of the algorithm on a network of real sensor nodes.** The main experiment aims at evaluating the energy consumption of software reconfigurations using our approach on a network of real sensors using the IoT-lab platform.

The next section will present our proposition to divide an IoT application into components, following the component model proposed on the Kevoree meta-model.

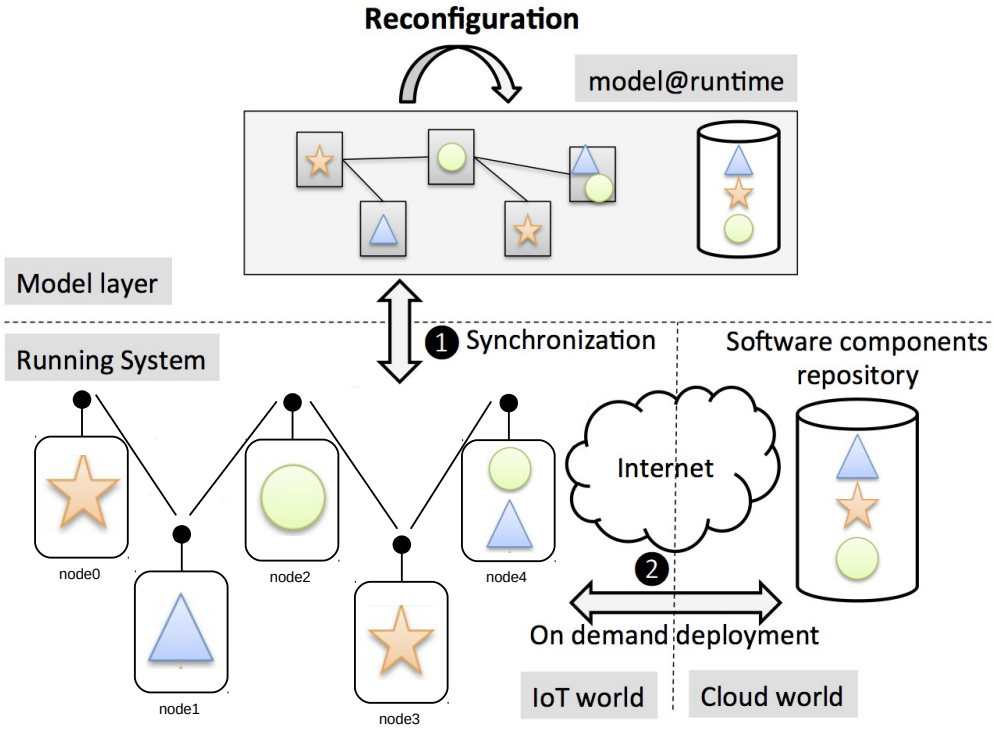


Figure 6.1: *Model@runtime principle.*

6.2 Componentization of applications in the IoT

One of the main challenges discussed on this chapter consists in decomposing IoT applications into smaller pieces in order to ease development and maintenance. Several approaches have already proposed component models for embedded systems [48] and WSN [73], [50], [80], [17], [98], on which the concepts already discussed in §3.3.1 are used to bring reconfiguration facilities onto these architectures. Indeed, many similarities can be found between embedded systems, WSN and the IoT, mostly on the resource constrained nodes being part of these networks. However, the network size, used protocols and applications complexity in IoT systems demand a different approach.

Our proposition based on a Model@runtime [78] presented in section §3.3.5, a paradigm which aims at simplifying the development of distributed dynamically adaptable systems, proposes the use of a model which represents the system state as depicted in Figure §6.1. This model can then be synchronized with the real running system:

- any change in the system state is reflected into the model,
- any change in the model will be sent to the running system which will adapt its

behaviour to reflect these changes.

This two-way synchronization can be done automatically or on demand.

Model@runtime is generally used together with a component based software architecture. In component based software architecture, an application is broken into different software pieces called software components which are linked together through software connectors [23, 75, 106] to from the architecture of the application. Software components can then be deployed independently at remote locations as illustrated in Figure §6.1. One of the benefits is that it facilitates the management of dynamic applications, which is a primary concern for IoT systems deployed in constantly evolving environments.

When using component based software architecture together with the model@runtime paradigm, the model layer represents the software architecture (a set of software components and connectors) mapped on the distributed physical execution environment. Any change made in the software architecture triggers an adaptation on the real system in order to deploy, remove or update software components or connectors.

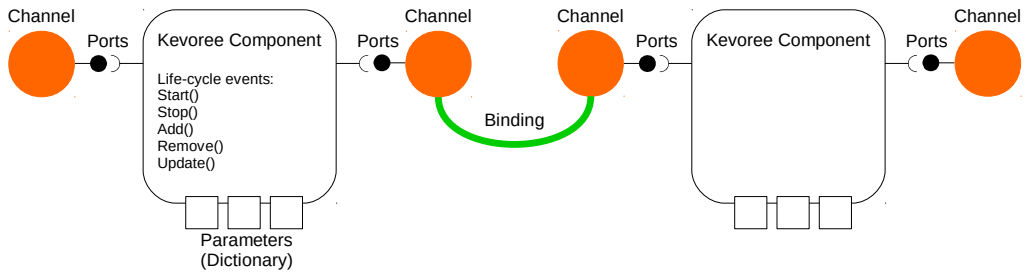


Figure 6.2: *Kevoree component model.*

Our component model can be represented as depicted in figure §6.2, on which the properties of a component are separated as follows:

- *Kevoree Component.* The instance itself containing the life-cycle events triggered either by an external event (through the Kevoree-IoT core) or by an internal adaptation (which will modify the model).
- *Properties.* Internal properties or parameters are the configurable settings that modify the component's behaviour These properties are represented by a dictionary, which can be affected by an *update* event.
- *Ports.* In contrast with other component models, an extra abstraction for component's interaction is provided by our model. It consists in exposing the ports (interfaces) only to another instance called *channel*, on which communication means are implemented in a separated way.

- *Channel.* An independent instance which is very useful to avoid direct function calls between components. Indeed, instead of using direct function calls, a communication channel can apply different and more complex semantics, a valuable feature in distributed environments [9, 43].
- *Binding.* Is the representation of the connections between channels. Several bindings can exist per channel, in a point-to-multipoint schema.

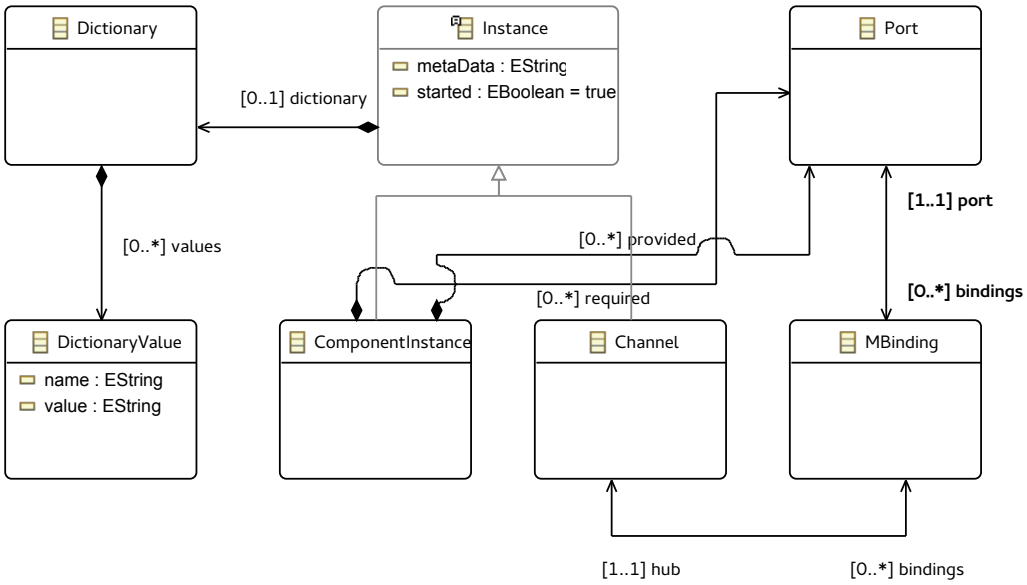


Figure 6.3: *Component model from the Kevoree meta-model.*

As for the Kevoree meta-model representation, we can observe in figure §6.3 that a component instance can use an unlimited number of ports which can also have an unlimited number of bindings. We can clearly identify a channel as a separated instance, thus the implementation is completely independent from the component. Moreover, the dictionary abstraction allows to configure both channels and components, adding flexibility to the communication means.

A component-based implementation for Contiki

The implementation of this component based model was done by mapping the abstract concepts into data structures, as a part of a Contiki loadable module. We use the Contiki's dynamic loader and linker feature described in [31], as a method to load dynamically the components and create its instances. Indeed, a Contiki process is used to register new components to the Kevoree-IoT core, which are then available in the form of deploy units. Afterwards, once the loading is completed, we can affect the life-cycle

of the application by creating new instances of the newly downloaded type. Instance creation is done using a simple memory block allocation scheme, allowing to create as many instances as free memory is available.

This adaptation of the component model, even if it is very oriented to its use in a Contiki environment, was optimized to achieve a behaviour very similar to the original Kevoree implementation, which is very high resource consuming. Indeed, this task was very challenging due to the constrained resources of the nodes on which we aim to run our experiments. Moreover, a more challenging problem raised while we tried to distribute components across the network. While using a straightforward technique to reach the target node of a component, which consisted in a direct deploy unit download from a remote repository, we realized that this method was very energy consuming. Indeed, every time a component was needed a considerable quantity of nodes in the mesh network were used to transport the packets to its final destination, and this was repeated for each node. Thus, the new challenge lies in the way a component is disseminated on the network. Indeed, providing the best path to download it and using local "cache" repositories to distribute a common component for several nodes could reduce energy consumption. Therefore, a new distribution algorithm is needed in order to provide such functionality. Details of this algorithm are discussed in the next section.

6.3 Calpulli: A distributed algorithm for component dissemination

In this section, we present *Calpulli*, an algorithm designed to properly distribute software components in the IoT. It is then possible to provide a better component distribution to the targeted nodes, thanks to the available information, both the execution state of the system (in the M@R) and routing details of the mesh network. The goal is to reduce redundant retransmission of identical information by caching, choosing the best node in the network on which we can store a requested component. Thus, with the use of *Calpulli*, we are able to save energy while distributing components (by reducing the retransmissions). The two needed features by *Calpulli* are described as following:

Routing properties of the running system. Networks used by IoT systems are often referred to as Low power and Lossy Networks (LLNs). The Routing Protocol RPL [111] is a IPv6 routing protocol for LLNs that builds a tree for routing, more precisely a Destination Oriented Directed Acyclic Graph (DODAG). This graph starts at the root/BR (Border Router), a specific node chosen by the system administrator and connected to the rest of the wired network. Each node in the graph has a rank that represents the number of hops to the root. In this hierarchy, each node on the graph has a routing entry towards its parent and it can send a data packet to the BR by forwarding it to its immediate parent. Figure §6.4 gives an example of a simple DODAG within a 9-nodes network.

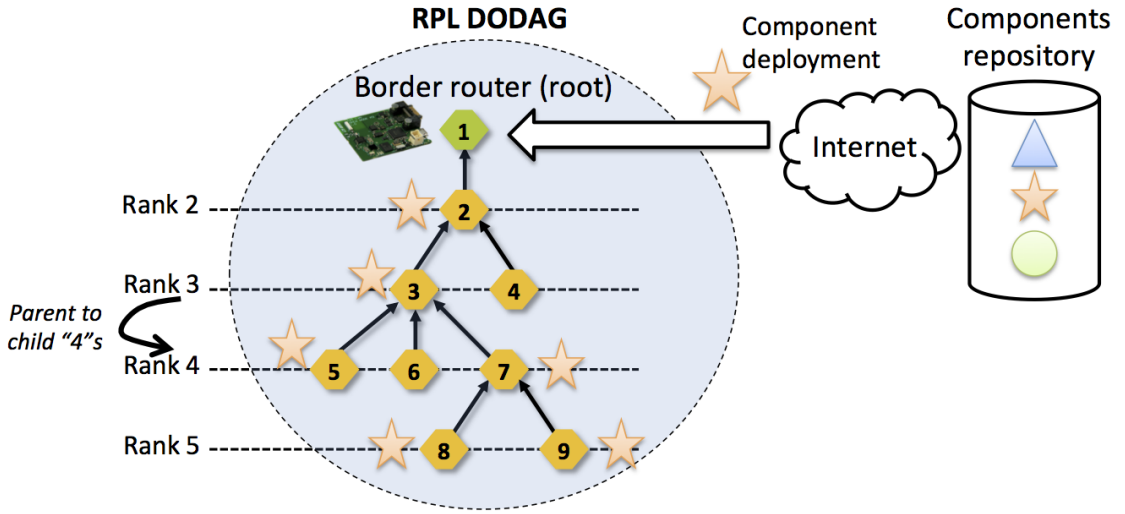


Figure 6.4: *Dissemination of software components using a DODAG.*

Information provided by the model@runtime. The model describes all the components of all the nodes in the network. After a model synchronization, a node knows which components must be downloaded from the remote repository for its own adaptation. Moreover, the information available on the model allows a wide introspection on the current deployment, thus it is possible to know which other nodes are requesting the same component, if any. In the example shown on Figure §6.4, the component represented by the ★ is needed by nodes identified as 2-3-5-7-8-9. As an example, thanks to the model information, node 9 knows that the component ★ is requested by nodes 2-3-5-7-8.

The objective of *Calpulli* can be summarized as follows:

- When the model has changed, the topology is used to identify how the components will be propagated, and where they will be cached, by parsing the model and looking for nodes requesting the same component (★ in Figure §6.4). Once it is located, *Calpulli* will mark it as a temporary caching node.
- This selected node (the node 2 in our example, called the *repository node*) downloads the component and acts as a local repository for other nodes (nodes 3-5-7-8-9).
- All nodes will perform the same steps, and since the model and routing information is the same all of them will select the same node as repository. Thus, they will wait for the repository node until it is ready to provide the component ★.

The selection process of the repository node for the component ★ uses the hierarchical structure of the DODAG, available in the routing information (RPL storing mode). When a node needs the component ★ in our example, two cases can occur:

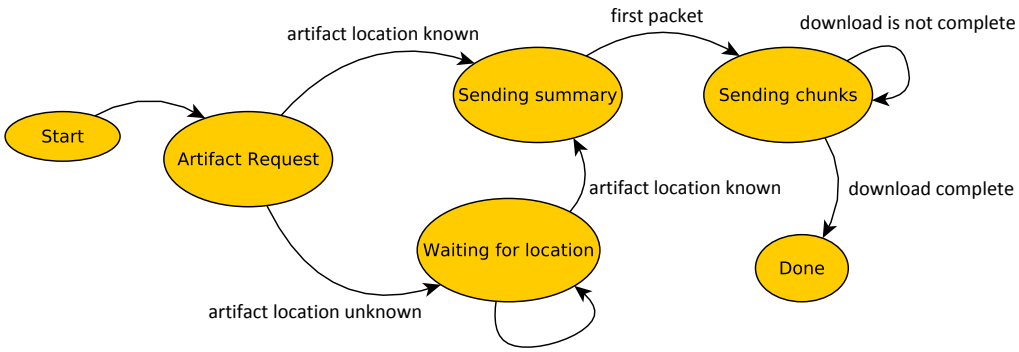


Figure 6.5: *State diagram describing a new component download, acting as a repository.*

i. **A node determines itself as a repository.** This happens by comparing the topology and the model, when it is found that more than 2 children need the same component. Thus, the node will wait for the children component requests.

ii. **A node determines that it needs a component.** If the node does not act as a repository, but find that a component is needed, it will just send a component request to its parent. Its parent, if it is not a repository, will forward the request to its own parent, and so on until reaching the main repository, outside the local network.

Thus, when a node which was not marked as a repository receives a message from a child node asking for a component, it just forwards the message to its immediate parent without modification. In contrast, when the node is marked as a repository node and receives a component request, it will store the address of the requesting node in a queue.

Calpulli can also determine that no node can be a repository candidate. Therefore, the component request will reach the Border Router, which will forward the request to the main repository. This main repository will act as a repository node, and it will also store the requesting node's addresses in a queue.

The implementation of *Calpulli* was integrated as a part of the Kevoree-IoT runtime, which makes use of the available UDP sockets on Contiki to transport a component from a remote repository located on the Internet. Indeed, a small protocol was developed to support *Calpulli*'s downloading mechanisms, in order to establish synchronization between nodes acting either as *"repositories* or *"clients"*. This protocol aim to use the node address determined by *Calpulli* as a repository, and also to transform any node into a repository if *Calpulli* determines it.

In Figure §6.5, a state-transition diagram is used to represent the details of the proposed protocol for components downloading. As we can observe, once *Calpulli* has determined the nearest repository candidate, an artefact request is sent to such node which will first determine if the artefact location is known (if the requested deploy unit

was already downloaded). Afterwards, if the location is unknown, it will ask directly to the main repository (a central repository somewhere on the Internet). When the location is known (deploy unit downloaded), a summary is sent to the *"client"*, indicating details about the size and number of chunks that will be sent. Afterwards, the server will receive a response asking for the chunks composing the deploy unit. When the download is over the node stops the protocol, and will look for the next request in the queue, if any.

In order to evaluate *Calpulli*, a series of experiments was carried on. The goal is to assess the energy savings provided by the algorithm, in comparison with state of the art algorithms for firmware updates, as well as the time needed to perform such updates.

6.4 Empirical evaluation

In this section, we evaluate the performance of the *Calpulli* algorithm with respect to energy consumption and delay to distribute software components. The energy consumption is related to the amount of transmissions on all nodes in the mesh network, since a data packet can be retransmitted by several nodes before reaching its destination. The time to deploy covers the delay between an artefact request and the deployment of the requested component. We have evaluated our algorithms on a representative scenario taken from a real deployment scenario. Indeed, our algorithm was implemented in the IoT-Lab testbed [41] with a specific configuration of 10 nodes, as a starting point. While the availability of a huge number of nodes in a testbed seems to be adequate for a large deployment experiment, some difficulties were encountered during experimentation, regarding the network topology and routing protocols restrictions. This is due to the layout of the physical deployment on the testbed, since the nodes are too close to each other, making difficult to build a mesh with several hops. Therefore, modifications to the transmission power and sensibility of the radio interfaces have been done, in order to have a good topology to run the tests. As for the routing protocols, we used ContikiRPL [103], an implementation of the RPL [111] protocol for LLNs and 6loWPAN. This protocol triggers regularly a "rebuilding" of the topology; thus, parents and children can change even if the nodes do not move. Indeed, this modification can change the behaviour of our algorithm, thus our experiments must be run several times using the same configurations. Even so, the representative results on 10 nodes show that our approach performs better regarding energy efficiency and deployment time in comparison to state of the art algorithms.

6.4.1 Use case

In our scenario, 10 selected nodes on the IoT-Lab testbed are preloaded with a firmware containing *Kevoree-IoT*, *Calpulli* and a UDP client/server which will act as a components repository when needed. Moreover, a set of components were available on a central repository running on a PC, which was interfaced with the testbed through a SSH tunnel to a border router. This PC is representing the main repository available on Internet, thus external to the IoT network.

Our goal for this use case is to answer the following research questions:

RQ1 : Does *Calpulli*, our dedicated software components distribution algorithm, consumes less energy in average than state of the art algorithms for firmware/components distribution?

RQ2 : Is *Calpulli* quicker to distribute software components than state of the art algorithms?

6.4.2 Experimental setup

This evaluation is based on experiments done with a set of physical nodes presented in the previous Subsection. All the experiments described in this section have been carried out on the IoT-Lab testbed [41]. Indeed, our experiments are designed to fit a "class 2" sized node, which is called "IoT-lab M3" on this platform. The choice of this node is done based on precedent analysis of performance in different nodes [101], providing the best trade-off between overall energy consumption and hardware capabilities.

In this evaluation, we consider two base line algorithms to compare our results:

- **Deluge:** Already described in Section §3.4.1, it is an epidemic dissemination protocol [60] which aims to distribute a "large data object" (*i.e.* larger that cannot fit into node's RAM) on WSNs. To manage the data size, an object is divided into elementary pages. A page is the basic unit of dissemination process and allows incremental upgrades. This protocol guarantees the distribution of exactly the same file to all nodes.
- **Kevoree:** the straightforward protocol used by the Kevoree framework [43] in the Java implementation which is agnostic of the network topology. This algorithm considers each node independently and each node will separately download the software components which should be installed from the main repository on Internet.

In the two algorithms, a node will use all necessary hops to reach the main repository, contributing to the overall energy consumption during the adaptation step.

For our evaluation, we are interested in these two different variables:

- **Energy consumption**. It is measured using the tools provided by IoT-Lab. The power used by each node is sampled every 100ms, thanks to a dedicated chip (INA226 current/power monitor) installed on the control board of every node. For a node's power consumption, the instantaneous power P_i is sampled by the control

node n times for a given time t_i. Thus, the total consumption in the interval $[t_0, t_n]$ is given by the following equation:

$$\sum_{i=0}^{n-1} \left((t_{i+1} - t_i) \frac{P_{i+1} + P_i}{2} \right) \tag{6.1}$$

then a mean value for all nodes is calculated to obtain the overall consumption for that experiment.

- **deployment time**. It is considered in the same way as the energy consumption, from the beginning of the adaptations until the system is executing all the component instances described on the new model.

Our experiments consider a fixed number of nodes, but the network topology varies between each experiment. Moreover, we also vary the number of components to distribute, in order to evaluate the impact of this variable on the three algorithms (Deluge, Kevoree and Calpulli). We developed four different components, compiled as Contiki ELF modules as described in §6.2.

As the topology of the RPL tree (DODAG) is hard to control within the IoT-Lab testbed, we have repeated each experiment 5 times to reduce any bias introduced by the random topology. For all the experiments concerning *Calpulli* and the Kevoree algorithm, we have generated 10 different models (configuration including which component has to be installed on which node) for a given number of components (1 to 10). Thus, in the models, the quantity of components will be distributed among the 10 represented nodes, choosing randomly one of the four available on the modelled repository. Moreover, the nodes where the components are distributed have been chosen randomly to reduce any bias in the experiment. A total of 100 models were generated for the experiments

As for Deluge, a simple Contiki ELF file was produced to represent a component to be disseminated in some selected nodes, using the implementation already available in Contiki. Since Deluge does not offer any guidance for the deployment, we selected and configure the same ten nodes to receive the disseminated ELF file. A sink node was also configured to provide the ELF files, acting as a main repository. Indeed, we decided to use a node as a repository, since a UDP server cannot be accessed by deluge, unless the server is also running Deluge. Since Deluge is not intended to run on big machines, where our UDP server is running, the best way to provide the components is to configure a single node as a repository.

The next Subsection will discuss the power needed to deploy artefacts using the described protocols.

6.4.3 Evaluation of power consumption

For our first test, we evaluated Deluge. When we tried to disseminate only one component to only one node, the protocol was very slow, exceeding the time given to our

experiment (20 minutes) before the component was deployed. After several tests, 3 components out of 10 were correctly deployed, when selecting 10 nodes to be part of the deployment. With this behaviour, Deluge seems to be very slow, with a very high energy consumption in the dissemination step. The consumption being too high, and the deployment unfinished for the given experiments, Deluge is not included in the results graphs.

The experiments for the Kevoree basic protocol and Calpulli, from 1 to 10 components deployed, were conducted as follows:

i. A generated model is chosen for a given quantity of components.

ii. That model is deployed in the network and the adaptations take place.

iii. The experiment is over when all the components are installed and running.

iv. This is repeated 5 times, using a different model for the same quantity of components.

A total of 50 experiments were conducted for each algorithm.

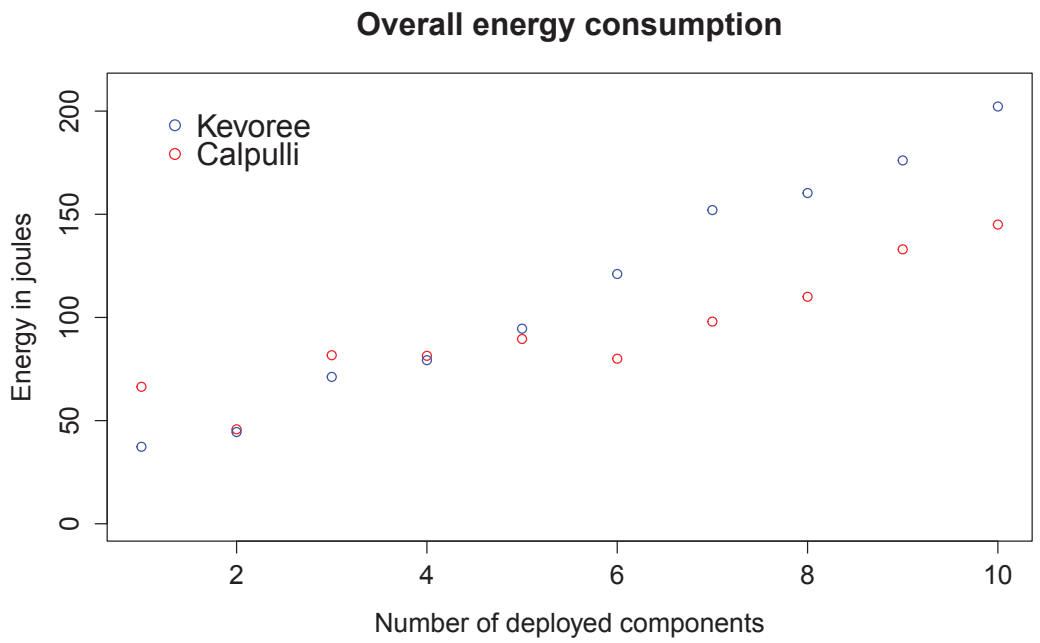


Figure 6.6: *Energy consumption in the whole network for the given components.*

The results of these experiments can be observed in Figure §6.6, where we acknowledge a reduction in the overall consumption when more than 5 components are deployed. Indeed, differences around 50 joules can be highlighted.

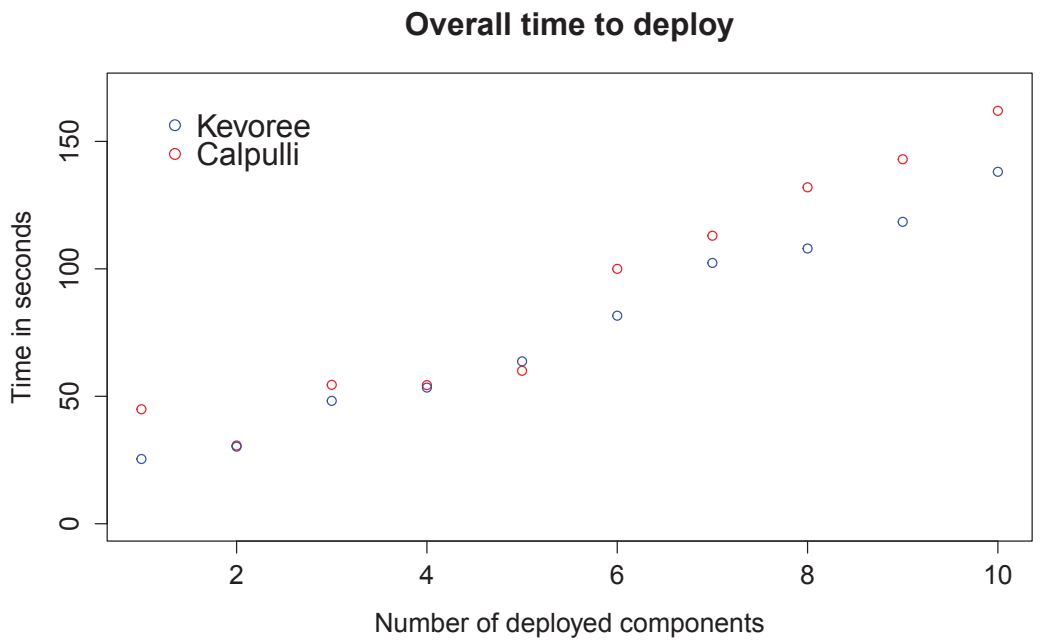


Figure 6.7: *Time to deploy the selected components.*

6.4.4 Evaluation of deployment time

The time to deploy was also evaluated as a part of a trade-off given by the delay of the analysed protocols. In the case of Deluge, as we already state in the previous evaluation, the time to deploy is too high, having only 3 nodes out of 10 working correctly, after the 20 minutes given to the experiment. For the Kevoree and *Calpulli* algorithms, we can observe in Figure §6.7 that the time is approximately the same for 2 and 4 components. After the fifth component, the time to deploy begins to increase. Since *Calpulli* has a delay in time before downloading the components, caused by the time used to resolve whether it is a repository or not, there is a trade-off between this and the gain in energy consumption. We argue that for larger networks the benefits could be more considerable. We discuss this on the next Section.

6.5 Theoretical evaluation

After an empirical evaluation of *Calpulli*, we needed to know if the tendency towards a better performance in energy consumption still there in bigger networks than 10 nodes, the subject of our first evaluation. However, the conducted experiments to get our first results needed a special configuration and set of nodes in order to create a stable RPL network. This is very time consuming while scaling our approach for more nodes, besides the technical challenges when larger models need to be parsed by the constrained nodes. Indeed, even if the M3 node is able to de-serialize big models, the implementation of the current model parser needs to be enhanced, which is a different challenge out of the scope of our main scientific problem. Therefore, this section proposes a theoretical evaluation based on a very simple model of content caching, which is the main purpose of *Calpulli*. The goal is to assess the gain in energy by avoiding retransmissions, which is the most energy consuming task.

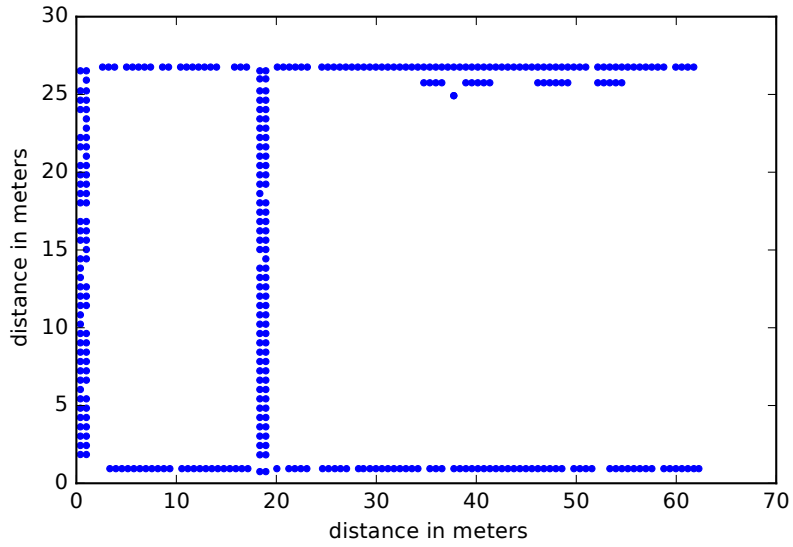


Figure 6.8: *Topology of the IoT-Lab M3 nodes at the Grenoble site.*

6.5.1 Model of the IoT network

One of the big challenges while evaluating *Calpulli*, was to build a representative RPL network by selecting the right nodes depending on the distance and the place of the Border Router. In this theoretical evaluation, we propose a model of the existing topology of the Grenoble IoT-Lab testbed[†1], which is presented on figure §6.8. This model will take into account all the M3 nodes present on the testbed represented by each blue

[†1] https://www.iot-lab.info/wp-content/uploads/2013/10/planMontbonnot.png

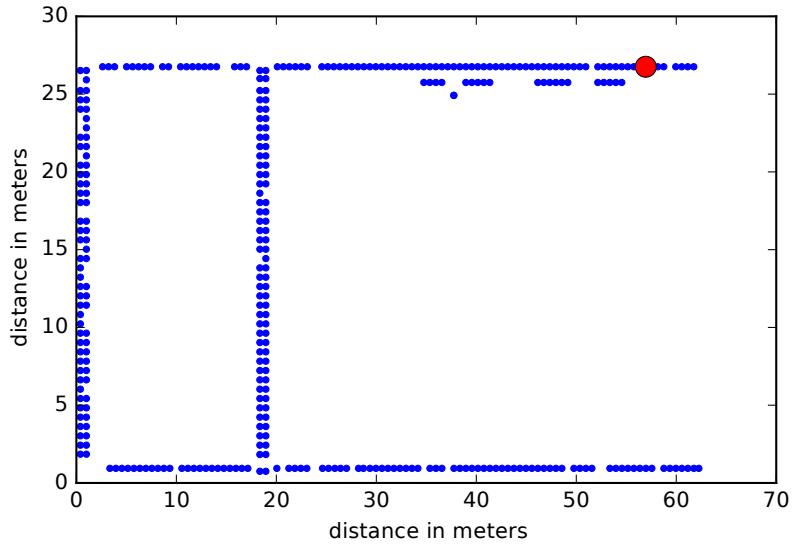


Figure 6.9: *Selected node as a Border Router with a maximum of 9 hops.*

point, including the physical distance between them shown in the axis of our figure (distance in meters), the radio capabilities (channel, range) and the possibility to simulate a RPL tree.

Once we represent our topology, we need to select a Border Router among the available nodes. This was done by selecting a node which its position is the optimal to build the biggest RPL tree, with a maximum quantity of hops to the last connected node. This is important to test an algorithm dedicated to distribute a specific content to a specific node, since our goal is to measure the performance on large multi-hop networks.

Figure §6.9 presents the selected node (node 59, big red point), which according to the model can build at least a 9 hops RPL tree. From this node, we can select a subset of reachable nodes at the maximum quantity of hops (9), as shown in figure §6.10. This subset is a reference to the possible RPL trees that can be formed by choosing nodes randomly, in order to change the topology.

Figure §6.10 illustrates a subset of nodes discussed above, while Figure §6.11 shows the RPL tree built from this subset. The next Subsection will show the results of the simulation with the presented topology and RPL tree.

6.5.2 Experimental setup

A subset of the topology is taken varying the max number of hops (from the sink), then in this sub-topology, the target "number of nodes" is selected: the number goes

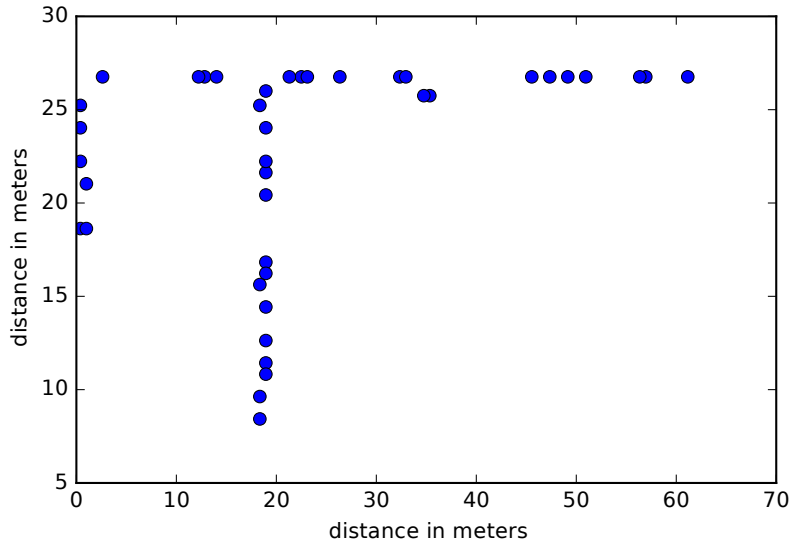


Figure 6.10: *Subset of reachable nodes from the sink.*

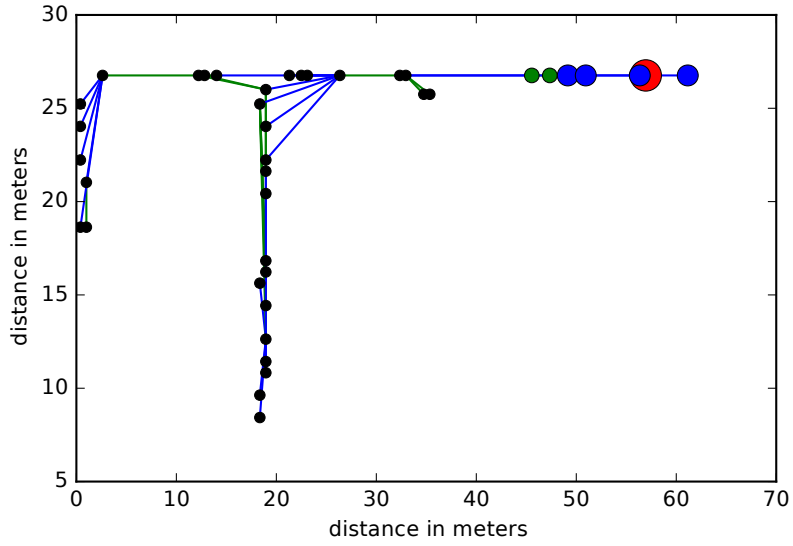


Figure 6.11: *Theoretical RPL tree.*

from 10 to the max number. For each such number of nodes 100 experiments are run where that number of nodes are selected randomly in the sub-topology (for example the sub-topology on figure §6.10). For each experiment, the cost of unicast and the cost with

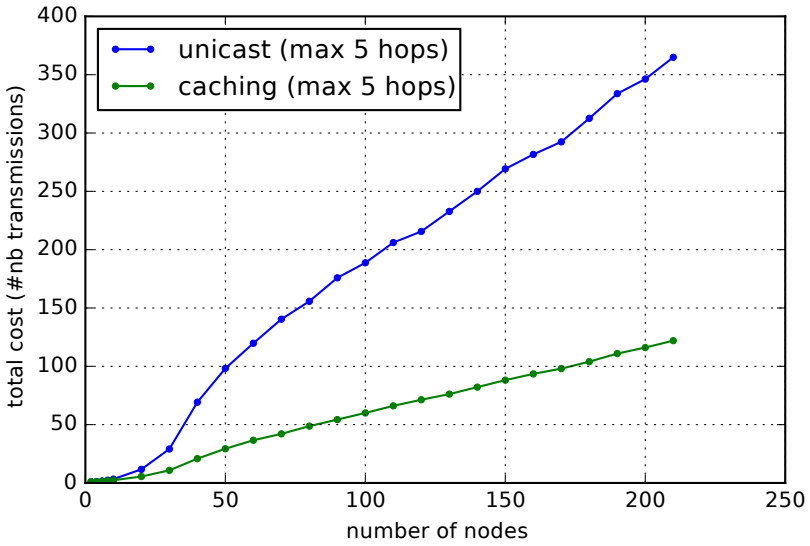


Figure 6.12: *Number of retransmissions for 5 hops maximum.*

caching are computed (= the number of necessary transmissions) with all nodes trying to get same identical content. The distributed content is only one component, which is spread randomly between half of all available nodes. It means that every node has a 50% chances to deploy a component, which allows an approximatively binary distribution of the content. Disconnected nodes are not considered.

6.5.3 Evaluation results

Once our experimental setup is ready, the measurements while distributing a component on the previously modelled network took place.

Figure §6.12 shows how the cost of unicasting the component from a single source grows linearly, since the needed retransmissions will grow as the hops increase. Indeed, the blue curve shows how the nodes need to retransmit a packet to the last possible hop every time the content (a component) is requested. For instance, at the end of the curve we can observe that we can reach around 210 nodes while limiting our tree to 5 hops. Then, the cost to send the content to half of the nodes grows to around 370 transmissions. The variations are due to the fact that several nodes could be attached to the same parent, so the retransmission takes place only once per node, according to the chance of deploying such component. Another example showing the same behaviour is shown in figure §6.13

In contrast, using a simple caching technique, a node can store the content to retransmit it without requesting it again to the source. Indeed, less transmissions are necessary

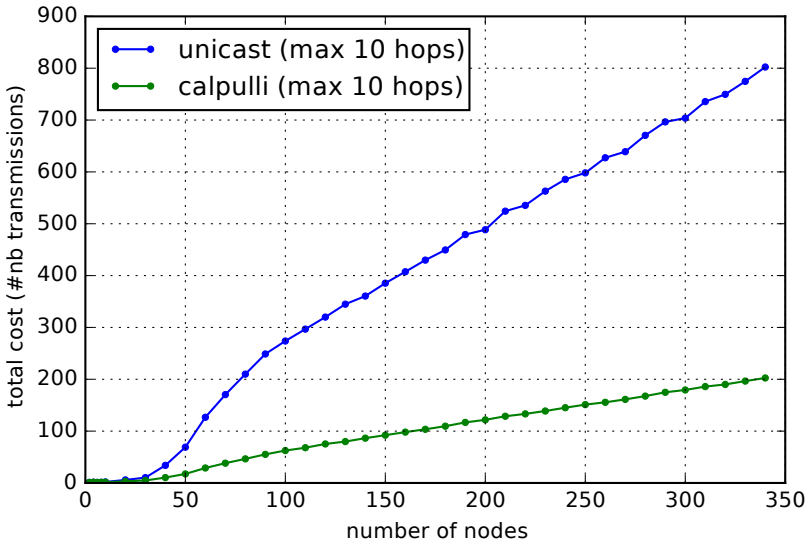


Figure 6.13: *Number of retransmissions for 10 hops maximum.*

to distribute the content to all concerned nodes, since each node which is storing the content can distribute it to its children. We can observe that the number of transmissions is drastically reduced by using this method, and even more for a bigger network as the presented in figure §6.13.

With a view to give a deeper analysis, we can measure the average cost per node while trying to transmit a component. This is done by taking the number of retransmissions divided by the quantity of nodes, in order to get an average cost per node depending on the quantity of hops. Indeed, while transmitting a component to a fixed quantity of hops and a fixed number of targets, the cost will be the division between them. In our example, observable in figures §6.14 and §6.15, we can see that the cost tends to the half of hops, since the target nodes are the half of the maximum reachable nodes (around 340). The cost in unicast, for instance, in a maximum of 10 hops will be near to 5, which is the average distance in hops from the sink to the destinations. But most important, we can observe that the cost with the simple caching technique tends to one. It means that while using this technique a node needs to transmit the content only once, since it's being stored on the next hop, so no need to retransmit it again if it is requested by another node on the same path.

6.5.4 Conclusion

These results show that an algorithm for content caching, in our case components, is still worth considering, since the energy savings can be huge compared to straightfor-

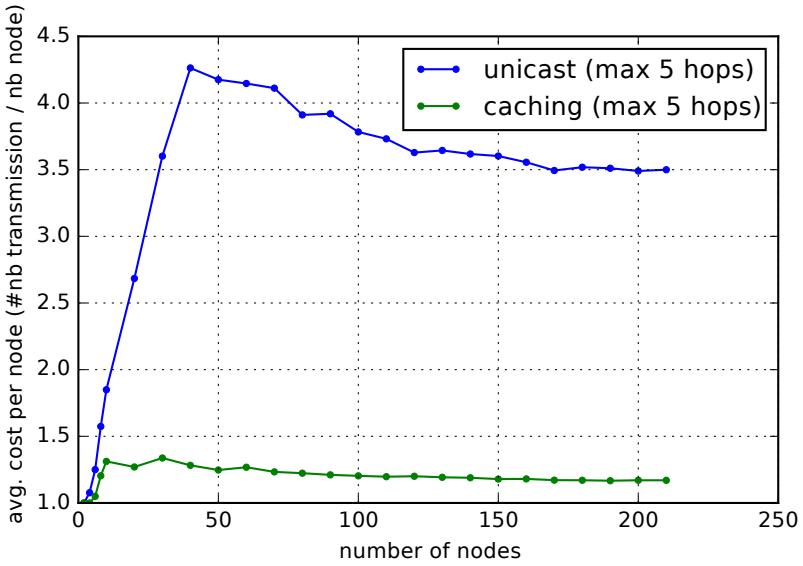


Figure 6.14: *Average cost per node for 5 hops maximum.*

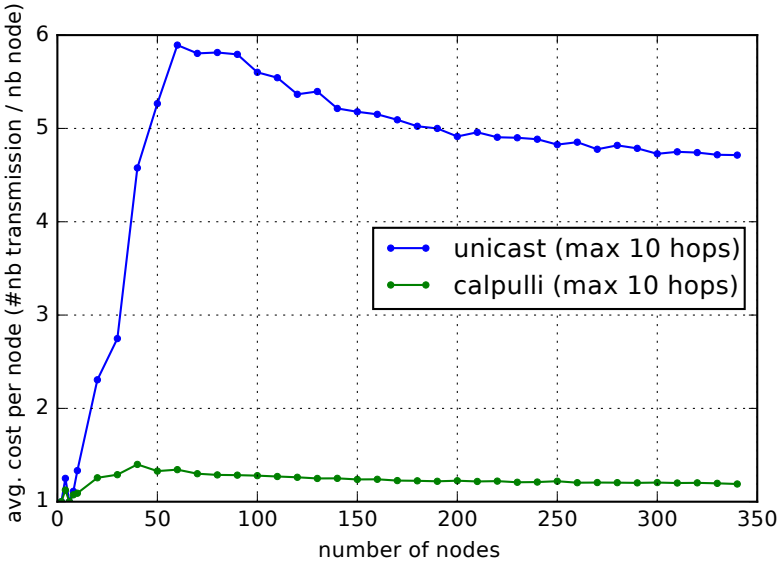


Figure 6.15: *Average cost per node for 10 hops maximum.*

ward unicast protocols. The code of this simulation is publicly available for further ref-

erence[†2]. Indeed, *Calpulli* performs a very similar algorithm using the knowledge of the model which contains the nodes requesting the same component (content). Then, since the topology is also known, *Calpulli* can determine which node is the best candidate to cache the requested component. In this theoretical experiment, we can extrapolate the behaviour of *Calpulli*, which we think will perform better in large networks than 10 nodes, as it was shown in our empirical evaluation, according to the results presented in this chapter.

However, we cannot completely neglect the energy needed by a node to cache a component, which can variate according to the component's size. Indeed, a deeper evaluation can be performed by also modelling the overall energy consumption of a node while transmitting and caching content. This evaluation will be part of our perspectives.

6.6 Conclusion

This chapter presented how Kevoree-IoT, our model@runtime implementation, can be used to deal with the problem of software deployment and configuration in Internet of Things systems. The main contribution presented is *Calpulli*, a decentralized algorithm which takes advantage of model@runtime information together with the network topology to optimize the distribution of software components, regarding energy consumption and deployment time. We highlight that *Calpulli* provides a very good trade-off between deployment time and energy consumption with respect to Deluge and the classical Kevoree algorithm. Indeed, our current Kevoree-IoT implementation along with *Calpulli*, can successfully distribute software components into an IoT network in an efficient manner.

Therefore, two things can be noticed on the execution of our approach:

i. We provided a mechanism to distribute software components to specific nodes

ii. and an algorithm to efficiently distribute such components.

With these contributions, we have added adaptation features to nodes being part of an IoT system. Indeed, each node can perform different types of adaptations and change its behaviour regarding the execution state before a new model is disseminated in the network.

Results

While evaluating *Calpulli*, our results show that a trade-off exists between components distribution speed and the energy consumption, resulting in overall energy savings. Indeed, in the empirical evaluation *Calpulli* was able to save energy while caching

[†2]https://cloud.sagemath.com/projects/269f7823-e948-4bab-8e0b-999ae12c8685/files/TopologyAnalysis.html

content on intermediate nodes before reaching the actual concerned node. Energy saving is very important in networks like the IoT, where the topology coupled with the energy constraints can determine the life-cycle of a node. Moreover, our theoretical results show that algorithms such *Calpulli* are very pertinent in networks with a huge quantity of nodes, especially on multi-hop mesh topologies.

These results are complemented by our theoretical evaluation, on which we argue that in bigger networks, *Calpulli* will save even more energy, according to the number of hops needed to reach the source of components.

Part IV

Conclusions and future work

Chapter 7

Conclusions

Software engineering is a research domain that has reached a very mature status nowadays. While its applications are intended to solve problems for very complex and large software systems, we have highlighted throughout this thesis some aspects that can be leveraged for their adoption to solve problems related to software management on the Internet of Things. More specifically, Model Driven Engineering has been studied on the context of distributed systems, since its similarities with the IoT seem evident. Therefore, the scientific problems raised by this new paradigm are the same than those already covered for distributed applications. However, the very different capacities and network topology of IoT systems make this adoption very challenging. Indeed, we showed that the hardware constraints and energy autonomy typical of IoT nodes are the main obstacle to implement the solutions proposed by the MDE community *"as is"*.

7.1 From IoT to MDE

Our state of the art described these two domains in detail, presenting at first the new paradigm of the Internet of Things and its main characteristics as a typical distributed system, and second the most recent tools aiming to solve a common issue found on these two domains: the management of a distributed software layer for (self)adaptation purposes. This management is very important in complex and large systems such as the presented on this thesis, since maintenance costs and services downtime can be drastically reduced by applying the right techniques. Indeed, the existing approaches proposed to deal with this problem have shown their capacity to increase efficiency by providing several tools for developers and maintainers, leveraging abstract models that can be manipulated in a safer and easier way than the actual system. Moreover, dividing these large systems into small software components easier to maintain has been presented as an excellent complement to provide adaptation mechanisms, by facilitating their replacement and addition of new ones, resulting in new behaviours. Thus, these promising results have been the source of inspiration to us to investigate the main principles for their adoption on very resource constrained environments such as the IoT.

7.2 Models@runtime to manipulate IoT software

Our first contribution make use of these models, known as *"models@runtime"*, to provide an abstract representation of a whole IoT network, including the running applications, their current state of execution and bindings between components. The implementation of this approach is inspired from the Kevoree meta-model, taking into account the very scarce resources found on IoT devices. We first show that the use of models@runtime on IoT devices is possible, with a small overhead compared to the benefits of the system representation through a model. Indeed, our experiments determined the limits of this approach, by calculating an approximate number of maximum nodes and components that can be present on the model before exhausting the memory capacities of a typical IoT device. Afterwards, we highlighted the system facilities necessary to execute the adaptations generated by a model manipulation (changes on the current model). These facilities have been extended for our target IoT nodes, in order to provide a complete experimentation platform which fits our requirements.

Thus far, we provided a mechanism to reflect our system in the form of a model@runtime and synchronize the current execution state to match with its representation. In order to provide the adaptation mechanisms, a component model based on dynamic loading of code has been implemented, relying on the possibility of the underlying system to actually load new code. This component model follows the same Kevoree meta-model representation, which can also be synchronized together with the IoT node and network characteristics.

We can highlight that the use of a model@runtime to manage the software layer in IoT devices is possible, and adds an abstraction layer which facilitates the distribution of tasks among different nodes on the IoT, thanks to the components decoupling of the whole system.

7.3 Towards "content caching" via *Calpulli*

Our second contribution highlights the very specific problem of components distribution, found in IoT environments due to the network topology. Indeed, this problem is not present in typical distributed systems, since they rely its communication means on very robust, fast and reliable networks. We argue that component downloading using epidemic dissemination of components reduces the capacity to finely select the nodes to be updated/extended, since state of the art protocols are not guided by an abstract representation of the system. In addition, individual downloading of components can use several nodes on a mesh topology, in order to reach the component repository. In most of cases, these nodes are used several times to forward the same content requested by different nodes, thus energy is wasted using this approach. Therefore, a new algorithm has been proposed to deal with the high-energy consumption while transmitting software artefacts through a mesh network, typical of the IoT. Our approach, called *Calpulli*, uses the knowledge of the network topology through a typical IoT routing protocol, RPL, coupled with the application information available on the model, to find the best path and

caching mechanisms for artefacts downloading. The evaluation of our approach showed that even in a small representative network, the benefits of using such an algorithm are important regarding the energy consumption.

In summary, our algorithm for component caching results in a very efficient way to distribute software artefacts among the nodes present in an IoT network, while saving a considerable amount of energy. Moreover, this algorithm will perform better as the network grows.

In overall, we can conclude that mechanisms for software management in very large, heterogeneous and non-monolithic distributed systems, as the Internet of Things, are necessary, due to the high efforts required to maintain the life-cycle of software running on these platforms. Moreover, we highlighted that IoT systems require specific mechanisms to provide such management, since the target nodes are very different to typical nodes present in distributed systems. Indeed, the constrained resources and network environment make the development of management tools very challenging. Therefore, the proposed approaches and algorithms to perform this task are the firsts steps towards a more complete framework to support dynamic behaviour and self-adaptation on the Internet of Things, as it was presented throughout this thesis.

Chapter 8

Perspectives

The presented research work has shown that software engineering tools coupled with a fine knowledge of the constraints found in IoT systems are able to enable adaptation features for these large distributed systems. Indeed, several perspectives are open for future work. The next sections will present them in detail.

8.1 Modelling tools for accurate code generation

Our current implementation relies on a manual transformation of the Kevoree meta-model in order to provide the C code that can be compiled for the Contiki OS. Therefore, automatic tools inspired, for instance, from KMF [46], can be used to provide a flexible framework which enables a more versatile meta-model implementation, in addition to a more accurate code generation. Moreover, using high level modelling tools can lead to optimizations on the generated code, since changes on the abstract representation are easier to perform than manual coding.

8.2 A flexible component development framework

The programming model used to implement components for resource constrained devices can have strong dependencies on the underlying OS. Indeed, our approach can be extended to support high level modelling tools to provide the right code depending on the chosen OS, avoiding specific development restrictions, since the component model is already provided.

8.3 Leverage ICN mechanisms for efficient components distribution

Our presented protocol, *Calpulli*, shares several principles that are being investigated by the Information-Centric Networking (ICN) [4]. Indeed, this paradigm explores in-network caching, multi-party communication through replication, and interaction models decoupling senders and receivers. The goal is, as for *Calpulli*, to provide the best mechanisms for highly scalable and efficient distribution of content to better cope with disconnections, disruptions, and flash crowd effects in the communication service. In our particular case, software component distribution, these mechanisms are of high interest, since reductions on retransmissions of the same content lead to energy savings. It is then interesting to investigate such approach, and to explore the possibilities to use our model-based approach to improve its efficiency.

Bibliography

[1] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pages 1–314, Sept 2011

[2] Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, pages 1–2793, March 2012

[3] Iso/iec/ieee international standard for ethernet. *ISO/IEC/IEEE 8802-3:2014(E)*, pages 1–3754, April 2014

[4] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26–36, 2012

[5] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010

[6] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80, April 2013

[7] E. Bailey. Maximum rpm. red hat software. *Inc., February*, 1997

[8] B. Baker. *The Official InstallShield for Windows Installer Developer's Guide*. M & T Books, 2001

[9] O. Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. PhD thesis, Lille 1, 2005

[10] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009

[11] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014

[12] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182. ACM, 2009

[13] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software-Practice and Experience*, 36 (11):1257–1284, 2006

[14] A. Carzaniga. A characterization of the software deployment process and a survey of related technologies. Technical report, Technischer Bericht, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1997

[15] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 148–157. IEEE, 2007

[16] Y.-T. Chen, T.-C. Chien, and P. H. Chou. Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 183–196. ACM, 2010

[17] D. Cid and others. Looci: The loosely-coupled component infrastructure. In *2012 IEEE 11th International Symposium on Network Computing and Applications*, pages 236–243, 2012

[18] T. Committee and others. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee*, 1995

[19] Contiki. Contiki ip stack. https://github.com/contiki-os/contiki/tree/release-3-0, Accessed: 2016-01-13

[20] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005

[21] I. Crnkovic and M. P. H. Larsson. *Building reliable component-based software systems*. Artech House, 2002

[22] E. Damou. *ApAM: un environnement pour le développement et l'exécution d'applications ubiquitaires*. PhD thesis, Université de Grenoble, 2013

[23] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 266–276. IEEE, 2002

[24] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, RFC Editor, December 1998

[25] C. B. Des Rosiers, G. Chelius, E. Fleury, A. Fraboulet, A. Gallais, N. Mitton, and T. Noël. Senslab very large scale open wireless sensor network testbed. In *Proc.*

7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCOM), 2011

[26] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu. Dynamic linking and loading in networked embedded systems. In *Mobile Adhoc and Sensor Systems, 2009. MASS'09. IEEE 6th International Conference on*, pages 554–562. IEEE, 2009

[27] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003. USENIX.

[28] A. Dunkels. A low-overhead script language for tiny networked embedded systems. *SICS Research Report*, 2006

[29] A. Dunkels. Rime—a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands*. Citeseer, 2007

[30] A. Dunkels. The contikimac radio duty cycling protocol. Technical report, 2011

[31] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 15–28, New York, NY, USA, 2006. ACM

[32] A. Dunkels, B. Grönvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004

[33] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006

[34] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The web of things: Interconnecting devices with high usability and performance. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 323–330, May 2009

[35] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session*, Raleigh, North Carolina, USA, November 2008.

[36] J. Estublier and G. Vega. Managing mutiple applications in a service platform. In *Principles of Engineering Service Oriented Systems (PESOS), 2012 ICSE Workshop on*, pages 36–42. IEEE, 2012

[37] J. Estublier, G. Vega, and E. Damou. Resource management for pervasive systems. In *Service-Oriented Computing-ICSOC 2012 Workshops*, pages 368–379. Springer, 2013

[38] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1): 18–28, 2010

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999

[40] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002

[41] E. Fleury, N. Mitton, T. Noel, C. Adjih, V. Loscri, A. M. Vegni, R. Petrolo, V. Loscri, N. Mitton, G. Aloi, and others. Fit iot-lab: The largest iot open experimental testbed. *ERCIM News*, 2015(101), 2015

[42] T. R. Foundation. The raspberry pi. https://www.raspberrypi.org/help/what-is-a-raspberry-pi, Accessed: 2015-08-26

[43] F. Fouquet. *Kevoree: Model@ Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes*. PhD thesis, Université de Rennes 1, 2013

[44] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *Distributed Applications and Interoperable Systems*, pages 16–30. Springer, 2012

[45] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012

[46] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. *An eclipse modelling framework alternative to meet the models@ runtime requirements*. Springer, 2012

[47] FP7-ICT-2009-5-257992. Project smartsantander. http://www.smartsantander.eu, Accessed: 2015-09-06

[48] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. A survey of configurable, component-based operating systems for embedded applications. *Ieee micro*, (3):54–68, 2001

[49] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. Smartfrog: Configuration and automatic ignition of distributed applications. *HP OVUA*, 2003

[50] P. Grace, G. Coulson, G. Blair, L. Mathy, W. K. Yeung, W. Cai, D. Duce, and C. Cooper. Gridkit: Pluggable overlay networks for grid computing. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 1463–1481. Springer, 2004

[51] E. Grassl and F. Schmidt. Energy-autonomous electromechanical wireless switch, March 28 2006.

[52] O. N. Günalp. *Continuous deployment of pervasive applications in dynamic environments*. PhD thesis, Université Grenoble Alpes, 2014

[53] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating systems for low-end devices in the internet of things: a survey. *Internet of Things Journal, IEEE*, PP(99): 1–1, 2015

[54] O. Hahm. Riot os. http://riot-os.org/files/RIOT.pdf, 2014

[55] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proceedings of the 21st international conference on Software engineering*, pages 174–183. ACM, 1999

[56] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176. ACM, 2005

[57] T. Hartmann, F. Fouquet, J. Klein, Y. Le Traon, A. Pelov, L. Toutain, and T. Ropitault. Generating realistic smart grid communication topologies based on real-data. In *Smart Grid Communications (SmartGridComm), 2014 IEEE International Conference on*, pages 428–433, Nov 2014

[58] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001

[59] HiKoB. M3 open node. https://www.iot-lab.info/hardware/m3/, Accessed: 2015-08-26

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

[61] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25–33. IEEE, 2004

[62] M. Jung, J. Weidinger, W. Kastner, and A. Olivieri. Building automation and smart cities: An integration approach based on a service-oriented architecture. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 1361–1367, March 2013

[63] W. Kastner, G. Neugschwandtner, S. Soucek, and H. Newmann. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178–1203, June 2005

[64] M. Kelly. Gain control of application setup and maintenance with the new windows installer-the new windows installer does more than just copy files; it lets your windows 95, windows 98, or windows nt. *Microsoft Systems Journal-US Edition*, pages 15–28, 1998

[65] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003

[66] J. Ko, K. Klues, C. Richter, W. Hofer, B. Kusy, M. Bruenig, T. Schmid, Q. Wang, P. Dutta, and A. Terzis. Low power or high performance? a tradeoff whose time has come (and nearly gone). In *Wireless Sensor Networks*, pages 98–114. Springer, 2012

[67] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010

[68] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things (IOT), 2012 3rd International Conference on the*, pages 135–142. IEEE, 2012

[69] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, 1990

[70] B. Len, C. Paul, and K. Rick. Software architecture in practice. *Boston, Massachusetts Addison*, 2003

[71] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002

[72] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and others. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005

[73] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Wireless Sensor Networks*, pages 212–227. Springer, 2006

[74] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *Computer*, (7):56–64, 2004

[75] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, 2000

[76] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, RFC Editor, September 2007

[77] B. Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability*. PhD thesis, Université de Rennes 1, 2010

[78] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009

[79] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009

[80] L. Mottola, G. P. Picco, and A. A. Sheikh. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In *Wireless Sensor Networks*, pages 286–304. Springer, 2008

[81] I. Murdock. Overview of the debian gnu/linux system. *Linux Journal*, 1994(6es): 15, 1994

[82] G. Nain, E. Daubert, O. Barais, and J.-M. Jézéquel. *Using mde to build a schizophrenic middleware for home/building automation*. Springer, 2008

[83] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner. Web services in building automation: Mapping knx to obix. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1, pages 87–92, June 2007

[84] R. Oliver, A. Wilde, and E. Zaluska. Reprogramming embedded systems at runtime. In *Proceedings of the 8th International Conference on Sensing Technology*, pages 124–129. IEEE Instrumentation and Measurement Society, 2014

[85] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, (3):54–62, 1999

[86] A. Paepcke. *Object-oriented programming: the CLOS perspective*. MIT press, 1993

[87] L. Pérez-Lombard, J. Ortiz, and C. Pout. A review on buildings energy consumption information. *Energy and buildings*, 40(3):394–398, 2008

[88] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67. ACM, 2003

[89] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 164–182. Springer, 2009

[90] M. Sarwar and T. R. Soomro. Impact of smartphone's on society. *European Journal of Scientific Research*, 98(2):216–226, 2013

[91] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014

[92] Z. Shelby. Embedded web services. *Wireless Commun.*, 17(6):52–57, December 2010

[93] G. N. Silva. Apt howto, 2001

[94] O. A. Specification. Deployment and configuration of component-based distributed applications specification, 2006

[95] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.*, pages 369–374. IEEE, 2009

[96] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, edition 2nd, 2002

[97] C. Szyperski. Component technology: what, where, and how? In *Proceedings of the 25th international conference on Software engineering*, pages 684–693. IEEE Computer Society, 2003

[98] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen. Optimizing sensor network reprogramming via in situ reconfigurable components. *ACM Transactions on Sensor Networks (TOSN)*, 9(2):14, 2013

[99] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, and G.-P. Jung. Approaches for service deployment. *Internet Computing, IEEE*, 9(2):70–80, 2005

[100] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 171–180. IEEE, 2009

[101] I. Tsekoura, G. Rebel, P. Glosekotter, and M. Berekovic. An evaluation of energy efficient microcontrollers. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–5. IEEE, 2014

[102] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, San Francisco, USA, April 2009

[103] N. Tsiftes, J. Eriksson, and A. Dunkels. Low-power wireless ipv6 routing with contikirpl. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 406–407, New York, NY, USA, 2010. ACM

[104] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014

[105] A. Van Moorsel. Metrics for the internet age: Quality of experience and quality of business. In *Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany*, volume 34, pages 26–31. Citeseer, 2001

[106] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000

[107] S. Vidal and J. Antill. Upgrade and downgrade in package update operations, August 12 2014.

[108] J.-B. Waldner and J. B. Waldner. *Nano-informatique et intelligence ambiante: inventer l'ordinateur du XXIe siècle*. Hermès Science, 2007

[109] X. Wang, W. Li, H. Liu, and Z. Xu. A language-based approach to service deployment. In *Services Computing, 2006. SCC'06. IEEE International Conference on*, pages 69–76. IEEE, 2006

[110] M. Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999

[111] E. Winter, T., E. Thubert, P., A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, RFC Editor, March 2012

[112] B. Woolf and R. Johnson. The type object pattern. *Pattern Languages of Program Design*, 3:132, 1996

[113] M. Younis and K. Akkaya. Strategies and techniques for node placement in wireless sensor networks: A survey. *Ad Hoc Networks*, 6(4):621 – 655, 2008

[114] J. Zhang and B. H. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380. ACM, 2006

Abstract

The Internet of Things (IoT) is covering little by little every aspect on our lives. As these systems become more pervasive, the need of managing this complex infrastructure comes with several challenges. Indeed, plenty of small interconnected devices are now providing more than a service in several aspects of our everyday life, which need to be adapted to new contexts without the interruption of such services. However, this new computing system differs from classical Internet systems mainly on the type, physical size and access of the nodes. Thus, typical methods to manage the distributed software layer on large distributed systems as usual cannot be employed on this context. Indeed, this is due to the very different capacities on computing power and network connectivity, which are very constrained for IoT devices. Moreover, the complexity which was before managed by experts on several fields, such as embedded systems and Wireless Sensor Networks (WSN), is now increased by the larger quantity and heterogeneity of the node's software and hardware. Therefore, we need efficient methods to manage the software layer of these systems, taking into account the very limited resources. This underlying hardware infrastructure raises new challenges in the way we administrate the software layer of these systems. These challenges can be divided into:

- *intra-node*, on which we face the limited memory and CPU of IoT nodes, in order to manage the software layer
- and *inter-node*, on which a new way to distribute the updates is needed, due to the different network topology and cost in energy for battery powered devices.

Indeed, the limited computing power and battery life of each node combined with the very distributed nature of these systems, greatly adds complexity to the distributed software layer management.

Software reconfiguration of nodes in the Internet of Things is a major concern for various application fields. In particular, distributing the code of updated or new software features to their final node destination in order to adapt it to new requirements, has a huge impact on energy consumption. Most current algorithms for disseminating code over the air (OTA) are meant to disseminate a complete firmware through small chunks and are often implemented at the network layer, thus ignoring all guiding information from the application layer.

Contribution 1: A new middleware to manage IoT Software Updates (SU)

In this thesis, we first propose a design of a new middleware dedicated to IoT devices to enable the management of software deployment and the dynamic reconfiguration of these systems, by updating its original firmware. This middleware will act as an abstraction layer, inspired from the Models@Runtime paradigm coupled with Component Based Software Engineering (CBSE), in order to manage the inherent adaptive behaviour of an IoT system. This new design will take into account the very constrained nature of the nodes being part of the IoT, which was not addressed by previous implementations of models@runtime and CBSE. It includes the manual transformation into C code of a models@runtime meta-model, which represents both the network and the application layer of the Internet of Things systems, which would be part of each device's firmware. Moreover, a new implementation is provided and an evaluation on a typical IoT infrastructure demonstrates the feasibility of providing a model@runtime middleware for these systems, addressing what we called the *intra-node* challenges. Thus, distribution, instantiation and (re)configuration of the software layer, now decoupled into small pieces represented by software components, is facilitated through the abstraction layer, which can be manipulated before its actual execution on the real system.

Contribution 2: A new algorithm to distribute SU in IoT networks

In our second contribution, we address the *inter-node* challenges. At runtime, we leverage the model@runtime coupled with the network information (obtained from the routing protocol) through a new algorithm to distribute software components only to those devices that need it, while adapting to a new context. Indeed, this new algorithm, called *Calpulli*, aims at minimizing energy consumption during the reconfiguration step. We have evaluated our algorithms on representative scenario taken from real deployment scenario. The firsts results show that our approach performs better regarding energy efficiency and deployment time in comparison to state of the art algorithms for the application layer maintenance. Moreover, a simulation to evaluate the scalability of the approach was performed, showing that our algorithm can perform better when the network grows.

Résumé

L'Internet des Objets (IdO) couvre peu à peu tous les aspects de notre vie. À mesure que ces systèmes deviennent plus répandus, le besoin de gérer cette infrastructure complexe comporte plusieurs défis. En effet, beaucoup de petits appareils interconnectés fournissent maintenant plus d'un service dans plusieurs aspects de notre vie quotidienne, qui doivent être adaptés à de nouveaux contextes sans l'interruption de tels services. Cependant, ce nouveau système informatique diffère des systèmes classiques principalement sur le type, la taille physique et l'accès des noeuds. Ainsi, des méthodes typiques pour gérer la couche logicielle sur de grands systèmes distribués comme on fait traditionnellement ne peuvent pas être employées dans ce contexte. En effet, cela est dû aux capacités très différentes sur la puissance de calcul et la connectivité réseau, qui sont très contraintes pour les appareils de l'IdO. De plus, la complexité qui était auparavant gérée par des experts de plusieurs domaines, tels que les systèmes embarqués et les réseaux de capteurs sans fil (WSN), est maintenant accrue par la plus grande quantité et hétérogénéité des logiciels et du matériel des noeuds. Par conséquent, nous avons besoin de méthodes efficaces pour gérer la couche logicielle de ces systèmes, en tenant compte les ressources très limitées. Cette infrastructure matérielle sous-jacente pose de nouveaux défis dans la manière dont nous administrons la couche logicielle de ces systèmes. Ces défis peuvent être divisés en:

- *Intra-noeud*, sur lequel nous faisons face à la mémoire limitée et à la puissance de calcul des noeuds IdO, afin de gérer les mises à jour sur ces appareils.
- *Inter-node*, sur lequel une nouvelle façon de distribuer les mises à jour est nécessaire, en raison de la topologie réseau différente et le coût en énergie pour les appareils alimentés par batterie.

En effet, la puissance de calcul limitée et la durée de vie de chaque noeud combinée à la nature très distribuée de ces systèmes, ajoute de la complexité à la gestion de la couche logicielle distribuée.

La reconfiguration logicielle des noeuds dans l'Internet des objets est une préoccupation majeure dans plusieurs domaines d'application. En particulier, la distribution du code pour fournir des nouvelles fonctionnalités ou mettre à jour le logiciel déjà installé afin de l'adapter aux nouvelles exigences, a un impact énorme sur la consommation d'énergie. La plupart des algorithmes actuels de diffusion du code sur l'air (OTA) sont destinés à diffuser un microprogramme complet à travers de petits fragments, et sont souvent mis

en oeuvre dans la couche réseau, ignorant ainsi toutes les informations de guidage de la couche applicative.

Contribution 1: Un nouveau middleware pour gérer les mises à jour logicielles dans l'Ido

Dans cette thèse, nous proposons d'abord de concevoir un nouveau middleware dédié aux appareils de l'IdO pour fournir la gestion du déploiement logiciel et la reconfiguration dynamique de ces systèmes, en mettant à jour son microprogramme original. Ce middleware agira comme une couche d'abstraction, inspirée du paradigme Models@Runtime couplé au CBSE (Component Based Software Engineering), afin de gérer le comportement adaptatif inhérent d'un système IdO. Cette nouvelle conception prendra en compte la nature très contrainte des noeuds faisant partie de l'IdO, qui n'a pas été abordée par les implémentations précédentes des models@runtime et CBSE. Il inclut la transformation manuelle en code C d'un modèle métier models@runtime, qui représente à la fois le réseau et la couche applicative des systèmes Internet des Objets, qui feraient partie du microprogramme de chaque appareil. De plus, une nouvelle implémentation est fournie et une évaluation sur une infrastructure IdO typique démontre la faisabilité de fournir un middleware models@runtime pour ces systèmes, en répondant à ce que nous appelons les défis *intra-noeud*. Ainsi, la distribution, l'instanciation et la (ré)configuration de la couche logicielle, maintenant découplée en petits morceaux représentés par des composants logiciels, est facilitée par la couche d'abstraction, qui peut être manipulée avant son exécution sur le système réel.

Contribution 2: Un nouvel algorithme pour la distribution de MAJ dans les réseaux IoT

Dans notre deuxième contribution, nous abordons les défis *inter-noeud*. Au moment de l'exécution, nous exploitons le modèle couplé aux informations réseau (obtenues à partir du protocole de routage) par un nouvel algorithme pour distribuer les composants logiciels uniquement aux appareils qui en ont besoin, tout en s'adaptant à un nouveau contexte. En effet, ce nouvel algorithme, appelé *Calpulli*, vise à minimiser la consommation d'énergie lors de l'étape de reconfiguration. Nous avons évalué nos algorithmes sur un scénario représentatif tiré d'un scénario de déploiement réel. Les premiers résultats montrent que notre approche se comporte mieux en termes d'efficacité énergétique et de temps de déploiement par rapport aux algorithmes de pointe pour la maintenance de la couche applicative. De plus, une simulation pour évaluer l'évolutivité de l'approche a été réalisée, ce qui montre que notre algorithme peut fonctionner mieux lorsque le réseau s'agrandit.