Service-Oriented Middleware for the Large-Scale Mobile Internet of Things

Sara Hachem

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Inria INVENTORS FOR THE DIGITAL WORLD

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Spécialité

Informatique

École doctorale Sciences et Technologies de Versailles

Présentée par

Sara HACHEM

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

Sujet de la thèse :

Service-Oriented Middleware for the Large-Scale Mobile Internet of Things

Middleware pour l'Internet des Objets Intelligents

soutenance prévue le 10 Février 2014

devant le jury composé de :

Licia CAPRA (University College London, GB)	Rapporteur
Daniele MIORANDI (CREATE-NET, IT)	Rapporteur
Philippe PUCHERAL (Université de Versailles Saint-Quentin-En-Yvelines, FR)	Examinateur
Marcelo DIAS DE AMORIM (CNRS, FR)	Examinateur
Thiago TEIXEIRA (Google, USA)	Examinateur
Valérie ISSARNY (Inria, FR)	Directrice de thèse
Animesh PATHAK (Inria, FR)	Co-directeur de thèse

Abstract

The Internet of Things (IoT) is characterized by a wide penetration in the regular user's life through an increasing number of Things embedding sensing, actuating, processing, and communication capacities. A considerable portion of those Things will be mobile Things, which come with several advantages yet lead to unprecedented challenges. The most critical challenges, that are directly inherited from, yet amplify, today's Internet issues, lie in handling the large scale of users and mobile Things, providing interoperability across the heterogeneous Things, and overcoming the unknown dynamic nature of the environment, due to the mobility of an ultra-large number of Things.

This thesis addresses the aforementioned challenges by revisiting the commonly employed Service-Oriented Architecture (SOA) which allows the functionalities of sensors/actuators embedded in Things to be provided as services, while ensuring loose-coupling between those services and their hosts, thus abstracting their heterogeneous nature. In spite of its benefits, SOA has not been designed to address the ultra-large scale of the mobile IoT. Consequently, our main contribution lies in conceiving a Thing-based Service-Oriented Architecture, that revisits SOA interactions and functionalities, service discovery and composition in particular. We concretize the novel architecture within MobIoT, a middleware solution that is specifically designed to manage and control the ultra-large number of mobile Things in partaking in IoT-related tasks. To assess the validity of our proposed architecture, we provide a prototype implementation of MobIoT and evaluate its performance through extensive experiments that demonstrate the correctness, viability, and scalability of our solution.

Keywords: Internet of Things, Scalability, Mobility, Middleware, Service-Oriented Architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The earliest recorded mention of the term "Internet of Things" (IoT) goes back to a presentation by MIT's Kevin Ashton in 1999. In it, he famously stated that adding Radio Frequency Identification (RFID) tags to everyday objects would create an Internet of Things.[1] The initial vision was to use RFID tags on objects in supply chains to point to Internet databases holding information about the tagged objects. Although his predicted IoT is undoubtedly becoming a reality, it now goes far beyond the original concept, to be characterized by the integration of large numbers of real-world objects onto the Internet, with the aim of turning high-level interactions with the physical world into a matter as simple as interacting with the virtual world today, and eventually blending both worlds together. As such, two types of physical objects that will play a key role in the IoT are *sensors* and *actuators*. In fact, such devices are already seeing widespread adoption in the highly-localized systems within our cars, mobile phones, laptops, home appliances, and wearable fabric.

However, due to the still growing hype surrounding the IoT vision, there is not a unanimous definition that can be adopted. Based on our survey of the literature, the most representative definitions, from our perspective, are:

- The definition provided by the CASAGRAS Project [CASAGRAS, EU, 2012]:

  > *A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capa-*

[1] Kevin Ashton, RFID Journal, 22 June 2009: "I could be wrong, but I'm fairly sure the phrase 'Internet of Things' started life as the title of a presentation I made at Procter & Gamble (P&G) in 1999"

> *bilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network connectivity and interoperability.*

- Future Internet Assembly/Real World Internet[1] definition:

> *The IoT concept was initially based around enabling technologies such as Radio Frequency Identification (RFID) or wireless sensor and actuator networks (WSAN), but nowadays spawns a wide variety of devices with different computing and communication capabilities generically termed networked embedded devices (NED). [...] More recent ideas have driven the IoT towards an all encompassing vision to integrate the real world into the Internet [...].*

- SAP definition by Haller *et al.* [Haller et al., 2009]:

> *A world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these 'smart objects' over the Internet, query and change their state and any information associated with them, taking into account security and privacy issues.*

On the one hand, while CASAGRAS perceives the IoT as an actual infrastructure of interconnected Things with sensing, communication and computation capacities, FIA focuses more on the characteristics of those Things. SAP, on the other hand, considers it more as a paradigm and a vision in an enterprise context, with security and privacy being important requirements. Although they do not provide an actual definition, Miorandi *et al.* [Miorandi et al., 2012] conclude, after thorough analysis of the literature, that the IoT actually spans three layers: i) a network of smart Things interconnected through technologies that extend today's Internet; ii) a physical layer

[1] FIA:http://www.future-internet.eu/.

of supporting technologies (RFID, sensors/actuators, etc.); and iii) a variety of applications and services that leverage those technologies. Moreover, authors identify six common attributes that smart Things should exhibit: i) have a physical embodiment; ii) possess communication capacities; iii) have a human readable name and machine readable address; iv) have a unique identifier; v) have computation capacities; and vi) have sensing/actuating capacities.

Our conclusions from surveying the literature are in accordance with the categorization of the attributes of Things, provided by Miorandi *et al.* Distilling the ideas from the above, we conceive the following definitions:

Definition 1.0.1. *A* Thing *is an addressable physical object with computation power and communication interfaces with embedded or attached sensors and actuators that allow it to interact with the physical world. A Thing must be aware of its own state regarding its resources and possibly its location.*

Definition 1.0.2. *The* Internet of Things*, an extension of today's Internet, is an infrastructure of networked Things that connects the real world to the virtual world by exploiting the Things' data capture, actuating, communicating and processing capacities, which can be provided as services.*

A major constituent of the IoT is *mobile Things* (e.g., robots, drones, smart fabric, smartphones, tablets, laptops and so on), all designed to facilitate our daily lives [Papadimitriou, 2009]. A mobile Thing has, in addition to the Thing's attributes presented above, the capacity to communicate wirelessly and change its location, either autonomously or with human assistance. A mobile Thing can also be aware of its location and its displacements, past and future ones.

Mobile Things are no longer a vision for the future. They are widely available and within everyone's reach. In fact, it is common knowledge that nowadays all mobile phones host at least two sensors: a camera and a microphone. As of 2011, there are 5.3 billion phone users of whom more than 1 billion own a smartphone[1] containing other sensors such as gyroscopes and barometers, in addition to computation power and open programming platforms (i.e., software systems with open APIs that allow developers to integrate third party applications with the platform).

The success of mobile computing and the mobile sensing paradigm are highlighted by the numerous research and industrial efforts towards making them a reality [Camp-

[1]US Strategy Analytics:http://www.strategyanalytics.com.

bell et al., 2008]. Additionally, they are motivated by two benefits that mobility introduces [Pereral et al., 2012]. Firstly, with the increasing number of mobile Things hosting various sensors and actuators, there is no need to spend large amounts of money on static sensor/actuator deployments. It now becomes possible to deploy a few powerful sensors/actuators that are complemented by the numerous cheaper ones embedded in mobile Things. It is also possible, in numerous cases, to require no deployment at all, especially since mobile Things can cover more areas than their static counterparts. Finally, unlike static sensors/actuators, which, in many cases, are placed in remote areas, the ones embedded in mobile Things such as mobile phones are regularly recharged.

In light of the above, we restrict our research focus in this thesis on the mobile portion of the IoT, comprising in particular mobile Things with 3G/4G cellular and WiFi communication capacity and endowed with self localization capabilities, henceforth referred to as *mobile IoT*. Additionally, we target a class of IoT applications that require location-based services exploited in scenarios such as outdoors environmental monitoring. However, the realization of the mobile IoT is not without its challenges, which we illustrate in the following section.

1.1 Mobile IoT Challenges

At first glance, the challenges facing the realization of the mobile IoT vision are similar to some already observed in today's Internet, which are related to the *large scale of available information and highly demanding users*, and the *heterogenous nature of its virtual and physical constituents*. However, looking deeper, these challenges are significantly different when taking into consideration the complexity of handling knowledge about the physical environment, the never-before-seen scale and most importantly, the continuous motion of mobile Things.

1.1.1 Scale

If we look at smartphones alone, there were 647 million smartphones in 2010 [Cisco, 2011] while there are more than 1 billion smartphones and phones equipped with Web access in 2013. Those values grow exponentially when accounting for the different types of sensors and actuators they host and the data they produce. Moreover, the

number of Internet users is foreseen to reach 7 billion by 2020 [Blackman et al., 2010]. Bearing this in mind, it is unrealistic and extremely inefficient to expect today's networking infrastructure to handle the load of even a small percentage of those users interacting with the mobile Things. There are in fact several constraints to take into account, especially, communication costs, since, a single sensing task, say measuring temperature in Rome, will require the direct involvement of millions of mobile Things, and in addition, lead to time delays and wasted energy. In more detail, exploiting the anticipated ultra large number of Things comes with numerous issues pertaining to managing the large volumes of data, provided by sensors for instance, which will lead to higher storage requirements. Another issue lies in identifying and accessing appropriate Things, among billions, to provide needed functionalities [Toma et al., 2009], which comes with high communication and computation costs. Additionally, processing data streams and the in-network processing of sensor data will also be another challenge directly related to the anticipated IoT scale [Stuckmann and Zimmermann, 2009, Papadimitriou, 2009]. Finally, composing the functionalities provided by the Things will be challenging with millions of heterogeneous alternatives expected to be available [Vermesan and Friess, 2011].

1.1.2 Heterogeneity

For the IoT to successfully become a global infrastructure of Things, those Things should be able to interoperate. However, for one, mobile Things, and the sensors/actuators they host, are manufactured by an assortment of vendors and will have different attributes concerning their communication, addressability, sensing, and actuating capacities [Mattern and Floerkemeier, 2010]. This multi-faceted heterogeneity has numerous repercussions related to identifying appropriate Things and functionalities as they can have various heterogeneous descriptions and description languages that hinder their discoverability. Similarly, the heterogeneous nature of the IoT hampers the composition of the functionalities they provide and their capacity to communicate and exchange data among each other or with the systems exploiting them [Vermesan and Friess, 2011]. All of these parameters lead to a rather challenging heterogeneity that makes the IoT extremely hard to work with. As their diversity increases, delegating tasks of refining and homogenizing those attributes to humans will simply not be feasible. In such a dynamic environment, with so many unknowns, it is cru-

cial to conceive high-level knowledge representation that all Things and humans can understand.

1.1.3 Dynamic Network Topology

One of the main characteristics of the mobile IoT is an infrastructure that is both dynamic and unknown, therefore subject to intermittent availability. The reason for this dynamicity is twofold. Firstly, it is possible that the Thing does not have sufficient resources to partake in a sensing, actuating or processing task. Secondly, the Thing may no longer be at the location of interest. As a consequence, applications will often end up depending on services that are no longer available. As for the "unknown" attribute, it stems from the fact that adding the scale to the mobility of Things entails keeping track of the locations of, not a few, but all millions of Things at all times. This is a tedious, near to impossible task, which requires high performance capabilities within devices whose monetary cost is proportional to processing power. Consequently, the most challenging requirement directly associated with the mobility of an ultra large number of Things is managing the underlying network of such Things [Zorzi et al., 2010] and enabling the systems requiring their functionalities to localize and communicate with them, in a timely manner, without being obliged to continuously track their mobility.

1.2 Thesis Contribution: Towards a Thing-based Service-Oriented Middleware

Engineering a solution to the aforementioned issues depends on the answer to the following question: *How to architecture a scalable global solution that abstracts the real world into a virtual world?*

Firstly, with the heterogeneous nature of the mobile IoT constituents, it is paramount to decouple the hardware details of mobile Things and their embedded sensors/actuators, along with the implementation technologies of their software components, from the functionalities they provide, i.e., their services. Consequently, Service Oriented Computing (SOC), a software engineering paradigm that builds on services as the core components of distributed heterogeneous software systems, is the most appropriate choice. The Service-Oriented Architecture (SOA) [Papazoglou, 2003]

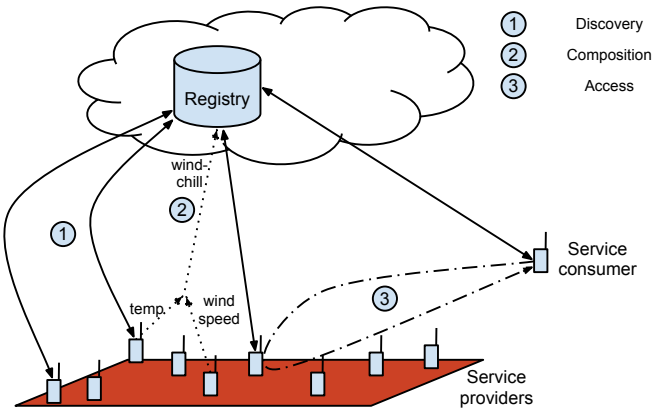

Figure 1.1. The SOA interactions in the Mobile Internet of Things.

provides a tangible architecture design for SOC, and it proved its worth for more than a decade [Issarny et al., 2005, King et al., 2006, Gu et al., 2005, Kalasapur et al., 2007, Guinard et al., 2010].

In this architecture, a service is a software entity that provides a public functionality, acquired through an invokable and discoverable interface while its internal operations and the attributes of its host are hidden. SOA involves three main actors that interact directly with one another (shown in Figure 1.1): a *service provider* (the host providing a running instance of a service), a *service consumer* (application that consumes the services), and a *Registry* holding descriptions of accessible service instances (running services).

To allow the interaction between the consumers and the providers, SOA systems are expected to provide three functionalities: *discovery*, *composition* of, and *access* to service instances (Figure 1.1). Specifically, *discovery* is used to publish (register) service instances in registries to later enable the search for and retrieval (look-up) of the ones that can satisfy a need. *Composition* of services is used when discovered services do not provide the needed functionality. In such case, other existing services are combined to provide a new, more convenient functionality. Finally, *access* enables the interaction with the discovered services.

The aforestated functionalities are essential, as they allow the system to handle the dynamic nature of the mobile IoT. However, there is a gap between the scale they are designed for and the scale they will have to encounter in the mobile IoT.

The scale SOA systems were expected to operate in are reflected by the nature of the interactions among the SOA players, where:

- During registration, *all* willing service providers are able to register (publish) the descriptions of their services.
- During look-up, *all* registered services that provide a needed functionality are retrieved to be accessed.
- The aggregation/composition logic should be applied by the consumer(s).
- Access to selected services is performed *directly* by consumer(s).

In fact, in the usual application of SOA systems, all tasks revolve around a business logic that can be satisfied by one or few services. Whereas in IoT, services are meant to monitor and modify a feature in the real world and should be numerous. Hence, expecting the service consumer to talk to all the service providers individually to access their services and acquire their measurements, then know how to treat each and every value (with different possible formats, types, units, etc.) requires communication and computation capabilities that the consumer will most likely not possess.

As an alternative, we revisit the SOA and its interaction patterns to support better scalability and exempt consumers from directly interacting with providers. Specifically, we provide in this thesis a solution that wraps discovery, composition, and access in a novel *Thing-based Service-Oriented middleware* that, unlike traditional SOA middleware, is aware of the physical properties of the real world and handles all cumbersome tasks on the consumer's behalf. The choice for a middleware-based approach is motivated by our aim to support reusability among a vast set of applications while hiding the underlying heterogeneous nature of the physical world, thus saving development efforts and time.

Thing-based Service-Oriented middleware is an intermediate-layer software system between Operating Systems and IoT applications that abstracts the implementation details of lower level system components, including sensing and actuating services and hardware specifications of Things. The Thing-based Service-Oriented middleware should provide discovery, composition and access functionalities to be exploited by an IoT application. An IoT application is a software system that builds on the Thing-based Service-Oriented middleware to provide Thing-based services integrated with

application-specific logic to end-users. A Thing-based service is a logical entity that abstracts sensors and actuators hosted on Things, with one peculiarity being that services are directly related to the physical location of their hosts.

However, to be inline with the mobile IoT requirements for a scalable solution that can manage the mobility of Things and abstract their heterogeneous nature, the functionalities of traditional Service-Oriented middleware are also to be revisited. Firstly, discovery should smartly manage the ultra-large number of Things willing to provide their services, by controlling the registration process, rather than blindly allowing all Thing-based services to participate. Discovery should also perform a smart retrieval process to determine which of the registered services should actually provide their functionalities. Secondly, composition should be executed transparently and automatically by the middleware, on the consumer's behalf. Moreover, composition should include a smart selection of services to compose, based on real world attributes, including detailed knowledge of the physics behind sensing and actuating tasks. Finally, access should take place transparently without burdening consumers with this task.

Moreover, the common approach to sensing/actuating tasks is by decoupling the sensing from the querying for information, which are asynchronous, and not triggered by a user's request. However, in many cases, the latter leads to more appropriate results. Indeed, sensing the world independently of the measurement request time and intended use of the data is not necessarily the best option; such approach may lead to large out-dated streams of data that can be of low benefit while more useful up-to-date measurements are missing. Consequently, we restrict our focus to on-demand discrete queries only. The issue of data freshness versus response time arises in this case. Therefore, in addition to the above requirements, it becomes crucial for IoT applications to execute instantaneous access to Thing-based services in a timely manner.

Our contribution in this thesis lies in conceiving MobIoT, a Thing-based Service-Oriented middleware solution that ensures the aforementioned requirements are satisfied through novel discovery and composition protocols, and wraps legacy access protocols to execute seamlessly.

Our work revolves around the following thesis statement:

> *The mobile Internet of Things is characterized by a heterogeneous and dynamic infrastructure comprised of an ultra large number of Things. By*

employing novel algorithms and protocols to efficiently discover and compose Thing-based services, a middleware-based approach can provide the necessary abstractions and functionalities towards the realization of truly scalable Mobile IoT applications.

In conformity with the contributions we specified above, we structure our thesis document as follows:

- Chapter 2 surveys the literature for research efforts in the IoT and related domains, Sensor Networks and Mobile Sensing in particular, to address the IoT heterogeneity, scale, and mobility issues.
- Chapter 3 gives an overview of MobIoT, our Thing-based Service-Oriented middleware for the Mobile IoT.
- Chapter 4 introduces our Thing-based service discovery approach to manage the Mobile IoT scale and dynamic aspects through novel probabilistic protocols and algorithms for Thing-based service registration and look-up. The algorithms we conceived are also presented in detail.
- Chapter 5 introduces our Thing-based service composition and the algorithms we designed to allow it to execute automatically, with no involvement from developers or end-users.
- Chapter 6 introduces our prototype implementation of MobIoT along with a set of extensive evaluations that demonstrate, not only the feasibility of our approach, but also the resulting quality of the discovery approach, along with its scalability, as compared to a regular SOA-based approach. This chapter introduces the CHOReOS middleware for large-scale service choreographies in the Future Internet that was conceived as part of the CHOReOS research project. Our contribution in this thesis is provided as a component of the CHOReOS middleware.
- Chapter 7 provides a comprehensive summary of our contributions along with the remaining research issues to be further investigated.

A considerable portion of our work has been published in various conferences, journal papers, and research reports as follows:

- **Journal Papers**
  - In [Issarny et al., 2011], we surveyed the literature and performed a state of the art analysis of SOA approaches in the Future Internet context.
  - In [Hachem et al., 2013b], we presented our work on the Thing-based Service-Oriented middleware, with special focus on the Registration contribution.
- **Peer Reviewed Conference Paper**
  - In [Hachem et al., 2013a], we presented our Registration contribution built on probability models.
- **Peer-Reviewed Symposium Paper**
  - In [Hachem et al., 2011], we presented in detail our contribution on knowledge modeling of the real-world.
- **Invited Paper**
  - in [Teixeira et al., 2011], we gave an overview of MobIoT architecture and initial goals.
- **Project Deliverables**
  - In [CHOReOS consortium, 2011a], we presented the CHOReOS perspective on the Future Internet and specified the initial CHOReOS conceptual model.
  - In [CHOReOS consortium, 2012a], we specified the initial architecture of the CHOReOS middleware.
  - In [CHOReOS consortium, 2011b], we provided the details of the CHOReOS middleware initial specification.
  - In [CHOReOS consortium, 2012b], we provided the details of the initial CHOReOS middleware implementation.
  - In [CHOReOS consortium, 2013], we provided the details of the final CHOReOS middleware implementation.

Chapter 2

The Mobile Internet of Things: State of the Art

Recalling from Chapter 1, in the mobile IoT vision, on-demand mobile sensing/actuation should be seamlessly enabled in software applications and be executed in a timely manner. This vision falls at the junction of two trends that govern a considerable portion of today's research focus and technological evolutions: i) exploiting ubiquitous mobile devices such as smartphones that are constantly connected to the Internet and have solid communication capabilities, which enable them to easily exchange messages or publish information; ii) establishing the IoT vision to ubiquitously integrate computation, sensing/actuation and communication capabilities in everyday objects. However, for the mobile IoT vision to become a reality and reach its true potential in rendering information on today's physical world remotely accessible while a maintaining a connection with and access to the physical constituents of the real world, several challenges must first be addressed. As elaborated in what follows, there are three challenges to overcome in order to realize this vision: 1) the heterogeneous nature of the IoT, 2) the IoT scale, and 3) the mobility of Things.

The *heterogeneity* issue can be perceived at three levels from a bottom-up perspective:

- **Sensor/actuator hardware**: An important aspect of the IoT is that new sensing/actuating hardware embedded in Things, will often not replace older generations of already deployed sensor/actuator networks. Rather, different generations of devices will operate alongside one another. Deployed hardware

is manufactured by distinct entities and displays heterogeneous attributes such as operating characteristics, e.g., sampling rates and error distributions.

- **Things**: The degree of heterogeneity becomes more complex as we move up from sensor/actuator components to the hosting Things, which are also produced by different manufacturers and possess distinct functional and non-functional characteristics.

- **Data**: The Things and their sensors/actuators produce heterogeneous data with different types and formats because physical phenomena can be described in various ways.

Evidently, to properly interact with Things, the above diversities should be managed.

The IoT *scale* issue can be perceived at three levels: 1) the ultra large number of devices that can be considered as Things; 2) the numerous functionalities Things can provide, especially with the continuously evolving ubiquitous sensor/actuator technologies; and 3) the large loads of data streams produced by Things. This scale leads to: unprecedented communication costs when interacting with Things; computation costs to process the generated streams of raw data; and storage costs to hold the information about Things and the generated data.

The third challenge lies in handling the *mobility* of Things. Given that a considerable portion of IoT systems require outdoors environmental monitoring and location-based services [Atzori et al., 2010], the functionalities of Things to exploit by this class of IoT systems, are directly related to their physical locality. Consequently, it is paramount to know their position in an efficient manner. The latter requirement is essential in the case where a request for on-demand measurements/actions at a specific location is received, and must be served, by mobile Things, with minimal delay. Continuously tracking mobile Things is achievable, when there are few Things to track, or if Things have low mobility. However, when the anticipated IoT scale comes into play, the process grows harder to satisfy as the network becomes too complex to track. Consequently any approach to answer queries for on-demand services in the mobile IoT should be able to manage to large number of Things, take their mobility into account, and rapidly provide the needed information.

Various software systems and underlying communication protocols have been proposed to help overcome the aforementioned challenges. In particular, with reusability

and portability as a motive, middleware solutions —that provide APIs to enable developers to exploit the functionalities provided by the Things— are amongst the commonly adopted design choices. However, given that the concept of exploiting large scale mobility for on-demand physical information retrieval or real-time actions has only emerged recently, there are few solutions that have been proposed to tackle, in conjunction, the scale and mobility challenges with timely on-demand services as a requirement. In fact, the proposed approaches tackle the challenges individually.

As a result, we survey in this chapter the various approaches in IoT and mobile IoT related research that tackle the heterogeneity, scale, and mobility issues separately. We analyze the capability of scalable and mobility-tolerant systems to fulfill requests for on-demand sensing/actuating tasks with minimal delay. Moreover, with the majority of real-world information in the IoT being provided by sensors, we include research efforts from the Sensor network domain in our survey.

2.1 Taming the Heterogeneous Internet of Things

The main motivation behind the IoT vision is to seamlessly merge the physical world with the virtual world. In accordance with this vision, several approaches were conceived to enable developers to integrate virtual abstractions of functionalities provided by Things into applications.

From our survey of the literature, we identify three complementary perspectives towards achieving this goal and harmonizing the heterogeneous nature of the IoT: i) providing *uniform representations* of Things, their sensors and their capabilities, in order to enable systems to reason over those capabilities; ii) wrapping the capabilities of Things in *uniform abstractions* to enable systems to exploit them; and iii) wrapping the communications with/among Things in *uniform networking protocols*. We review each in turn.

2.1.1 Representing Things

When requesting measurements/actions from Things, it is crucial to understand the specifications of Things, i.e., metadata, in order to identify which of them is appropriate to take part in performing the sensing/actuating task. This requirement is satisfied through *descriptions* —provided by developers— of Things, that in par-

ticular characterize: the hosted sensors/actuators, the provided functionalities, and the produced data, which are compared against the application request. However, given the heterogeneous nature of the IoT, the process of characterizing Things is not straightforward, due to the diversity of Things and thereby related descriptions.

One possible way to tackle this issue is through the approach proposed in [Guinard et al., 2010] to extend a request for a functionality with synonyms found through, for instance, a Yahoo! Web search. The result is then matched to as many different descriptions as possible. However, this approach is rather limited as it does not enable semantic reasoning over the descriptions. The latter is accounted for, by the majority of relevant research efforts, through *Semantic Web* technologies.

Semantic Web technologies allow machines to understand the meaning behind data, while keeping its representation simple enough for humans to understand it as well. The core technologies for developing the Semantic Web are *ontologies*. An ontology is defined as "a formal, explicit specification of a shared conceptualization" [Guarino et al., 1994] and is used to represent knowledge within a domain as a set of concepts related to each other.

Indeed, most existing efforts to provide uniform representations for the IoT entities adopt the semantic approach and exploit ontologies. A big portion of ontologies for the IoT is inherited from efforts in WSN to model sensors and actuators (e.g., [Compton et al., 2012, 2009, Russomanno et al., 2005]). The most influencing contribution to modeling sensor/actuator networks is the taxonomy provided by the Sensor Web Enablement (SWE) initiative of the Open Geospatial Consortium (OGC)[Botts et al., 2008], in particular:

- **Sensor Model Language (SensorML)**. SensorML Describes the geometric and observational characteristics of sensors and sensor systems. It provides a vocabulary to model sensors at different granularities, starting from thermometers to satellites.

- **Transducer Model Language (TransducerML)**. TransducerML describes transducers, their characteristics and the data they capture/produce. A transducer can be a sensor or an actuator. Unlike SensorML, TransducerML describes the internal components of a sensor and how they capture and produce the output data.

- **Observations & Measurements (O&M)**. O&M models observations and

measurements along with constructs to accessing and exchanging them. It models concepts related to a sampling task such as features of interest, the observation result, the observation time, and phenomenon time.

The taxonomy is provided in XML, which is useful for data presentation. It allows users to structure their data while adding tags and labels. Similar efforts were made in senML[1], a lightweight markup language for sensors to enable the representation of their measurements and parameters, using JSON, XML or Efficient XML Interchange (EXI). However, the above media types do not provide any possibility to reason over the meaning beyond the syntactic level. Consequently, several efforts were made to build on the SWE taxonomy while providing ontologies with an agreed upon vocabulary that matches the SWE taxonomy. A commonly exploited ontology, to reason about sensors, that builds on **SensorML** and **O&M** is SSN [Compton et al., 2012], provided by the *W3C Semantic Sensor Network Incubator Group*. SSN models sensing specific information from different perspectives: a) *Sensor* perspective to model sensors, what they sense and how they sense; b) *System* perspective to model systems of sensors and their deployment; c) *Feature* perspective to model what senses a specific property and how it is sensed; d) *Observation* perspective to model observation values and their metadata. SSN is aligned with the *DOLCE-Ultralite* (DUL) foundational ontology[2], which captures the logic underlying natural language and human common sense. It introduces categories as accepted by human perception and social conventions.

For examples on other sensor ontologies, we refer the interested reader to the survey by d'Aquin and Noy [d'Aquin and Noy, 2012]. Many of the ontologies surveyed by d'Aquin and Noy provide solid basis for the representation of sensors, actuators, and their data. However, those entities are only a portion of the IoT. More efforts have been made recently to extend the ontologies with IoT-specific semantics, including Things and the functionalities they provide. For instance, *Sense2Web* [De et al., 2012] provides an ontology that models the following Thing-related concepts: the *Entity* (equivalent to a feature on interest); the *Device*, which is the hardware component (equivalent to a Thing); the *Resource*, which is a software component representing the entity; and the *Service* through which a resource is accessed. A resource can be a sensor, actuator, RFID tag, processor or a storage resource. To model sensors for

[1]senML:http://tools.ietf.org/html/draft-jennings-senml-08
[2]DUL:http://www.loa-cnr.it/ontologies/DUL.owl.

instance, a *hasType* property links a resource to an instance of a sensor that complies with an available sensor ontology (e.g. SSN). A location for instance, that of a Device or a Resource, can be described by exploiting ontologies such as GeoNames.[1]

With the large palette of available ontologies, we posit that the solutions to the representations of Things have reached a mature stage. However, exploiting Semantic Web technologies does not address the heterogeneity of the functionalities provided by those Things, which can be abstracted using various technologies discussed next.

2.1.2 Abstracting Things

Several efforts have been undertaken to wrap the heterogeneous functionalities of Things within higher level abstractions, mostly services, that can be exploited by IoT applications. This is essential to decouple the heterogeneous hardware specifications from the functionalities of Things. The needed abstractions can, in particular, be provided by Service-Oriented Computing (SOC), which builds on services as the core components of software systems. Service-Oriented Architecture (SOA) provides a tangible design for SOC. It is characterized by its ability to provide loose coupling between services and their hosts, which in this case, correspond to sensing/actuating services and Things respectively. Given its benefits, SOA has been adopted by the majority of IoT-related research as the goto approach for abstracting Things as services, and managing the interactions specific to the technologies employed when enacting the services, i.e., technologies employed when creating, discovering and accessing the services. We proceed in the following to present how SOA has been exploited to abstract Things as services, which belong to one of the two following paradigms:

- A *Web Service* is defined in [W3C] as "*a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using (traditionally) SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*". However, WSDL, albeit being well-structured and beneficial to overcome heterogeneity issues, is a heavyweight XML document that requires expensive parsing, which

[1] Geonames:http://www.geonames.org/.

hinders the usability of Web Services on Things. Additionally Web Services come with a set of standards that are hard to implement on constrained Things. To overcome the latter issue, an alternative is to exploit the Device Profile for Web Service (DPWS), which builds on a limited set of required standard implementations that are more appropriate for resource constrained Things.

- A *RESTful Service* is defined by [Richardson and Ruby, 2008] as "*a simple Web service implemented using HTTP and the principles of REST. It is a collection of resources, with three defined aspects: the base URI for the Web service; the Internet media type of the data supported by the Web service (this is often JSON, XML or YAML but can be any other valid Internet media type); the set of operations supported by the Web service using HTTP methods (e.g., POST, GET, PUT or DELETE)*". Unlike Web Services, RESTful services abide by no standard to define service interfaces, which can be specified in various ways: informal text, formal text, structured XML, WSDL, etc. With the various options to describe services, REST provides less support for interoperability, especially as compared to Web Services. However, by relying solely on HTTP, therefore requiring no additional messaging layer, and the possibility to exploit lightweight message formats (JSON, plain text, etc.), RESTful services require less computation efforts than Web Services and can be exploited within resource constrained Things. Moreover, exploiting REST simplifies the development process by no longer requiring developers to make architectural decisions pertaining to the various layers of the Web Service standards to implement [Pautasso et al., 2008].

The above paradigms enable the merging of the physical and the virtual worlds as they abstract the functionalities of Things as services. There are two approaches to employing them: i) Abstracting physical Things as services; ii) Abstracting virtual Things as services. We proceed in the following to presenting each approach.

Abstracting physical Things as services. When abstracting physical Things as services, consumers have direct access to the Things providing the sensing/actuating functionalities, and in particular, to the raw data produced, with no processing or aggregation over the data. The functionalities to provide the raw data or perform an action can be consumed as Web Services (e.g. [Guinard et al., 2010]) for Things with

sufficient resources, or REST for constrained Things that cannot host Web Services. The latter is majorly adopted in the Web of Things (Wot) context (e.g. [Guinard et al., 2011]). The difference between the WoT and the IoT is that services, in the former, should be reachable through the World Wide Web, while in the latter, they can also be accessed directly.

Abstracting virtual Things as services. Virtual Things provide consumers with access to the data already produced by Things and decouple between physical Things and data they generate. In virtual Things, data undergoes one or several processing/aggregation steps and can be the result of data provided by one or several data sources. The latter is not feasible in the case of physical Thing abstractions. The benefit of virtual abstractions is that they enable consumers to manipulate the data from several sources at once without interacting with the Things generating it. Similar to abstractions of physical Things, virtual Things can be accessed as Web Services or Rest services. Virtual Things abstractions are adopted in GSN by [Aberer et al., 2007], through the concept of virtual sensors. Virtual sensors abstract data streams generated by one sensor, or several data streams from other virtual sensors, referred to as *complex virtual sensors*. Virtual sensors are accessible through the Web or accessible as Web Services. An example of a virtual sensor in GSN is presented in Listing 2.1. The specification determines that readings provided by a remote temperature sensor should be provided as an average.

Listing 2.1. A sample of a Virtual Sensor Specification.

```
<output-structure>
   <field name="TEMPERATURE" type="integer"/>
</output-structure>
<input-stream name="dummy" rate="100" >
   <stream-source alias="src1" sampling-rate="1"  storage-size="1h">
      <address wrapper="remote">
         <predicate key="type" val="temperature" />
            <predicate key="location" val="bc143" />
      </address>
      <query>select avg(temperature) from WRAPPER </query>
   </stream-source>
   <query>select * from src1</query>
</input-stream>
```

2.1.3 Interacting With Things

Even with Things uniformly represented, and services uniformly created, the heterogeneity issue remains unresolved at the networking level when discovering and accessing the services. There are several networking protocols that can be adopted following two approaches: Direct interactions among Things in the same local network or remote interactions over the Internet.

When interacting with the Things locally, to discover and access their services, there are various networking protocols that can be adopted depending on the technologies the Thing possess (e.g., Bluetooth, uPnP, ZigBee). A commonly adopted approach to hide the heterogeneous protocols is through protocol-specific plugins within Service-Oriented middleware solutions. The plugins extract descriptions from each protocol and hides them behind —in most cases semantic— descriptions (e.g., [Song et al., 2010b]).

To enable remote discovery and access to services, through the Internet, in addition to REST-based or SOAP-based (for Web Services) communication protocols, efforts are being undertaken to provide uniform protocols that enable lightweight communications with constrained devices, such as *CoAP* (Constrained Application Protocol) [Bormann et al., 2012] proposed as a REST-based communication protocol, which replaces HTTP/TCP with CoAP/UDP. Another example is 6LowPAN, which builds on IPv6 to enable small constrained devices to be connected to the Internet.

As can be concluded, Things and produced data can be abstracted into more or less uniform representations by exploiting Semantic Technologies. The functionalities of Things can be abstracted, based on a Service-Oriented Architecture, into services compliant either with REST or Web Services. Finally, the networking protocols to discover and access the services can be hidden by Service-Oriented middleware solutions. Consequently, we posit that solutions to heterogeneity issues in the Internet of Things have reached a mature stage and can be exploited to provide interoperability among Things. A summary of the approaches presented in this section and their limitations is presented in Table 2.1.

Req	Approach	Idea	Evaluation
Heterogeneity	Uniformly Representing Things	SWE markup languages as standard vocabulary	+ Well adopted vocabulary + Enable syntactically uniform descriptions - No semantic reasoning
		SenML: Lightweight sensor markup language	+ Lightweight + Enable syntactically uniform descriptions - No semantic reasoning
		Ontologies with semantic reasoning support	+ Semantic reasoning + Overcome and reason over heterogeneous descriptions - Lack representation of real world knowledge
	Abstracting Things	SOA decoupling functionalities from hosts	+ Overcome heterogeneous host specifications - Not scalable

Table 2.1. Comparison of existing solutions to the heterogeneous IoT.

2.2 Managing the Large Scale Internet of Things

When it was initially conceived, the Internet of Things was aimed at connecting objects with RFID tags into Internet databases to monitor products. Some efforts were made to transition from passive RFID tags to active tags. Active tags incorporate sensing capabilities and can contain onboard batteries [Want, 2004, 2006]. To enhance battery lifetime, active tags can leverage Bluetooth Low Energy (BLE) technologies for instance, where batteries such as C32 batteries can last up to 24 months in real life deployments. Other efforts were also made to integrate passive RFID tags into deployed sensor networks with some or all nodes acting as gateways to the Internet. Product monitoring evolved, along with the evolution from simple RFID tags to active tags and entire sensor networks, into monitoring: products, traffic, the environment, personal smart spaces, etc. In line with this evolution, the scale of the IoT is perceived, first and foremost, from a *data scale* perspective, where a large number of Things provide bulks of data streams to manage, analyze and process. Attempts to provide those requirements can be grouped into: in-network process-

ing; data aggregation; distributed processing; or outsourcing data processing to the Cloud. We present those approaches in the following and analyze their ability to address the on-demand large-scale sensing/actuating issues. The issues are related to the communication overhead resulting from large-scale access to sensors/actuators to acquire their measurements, especially with minimal delay constraints.

In-network processing and aggregation. In-network processing is adopted to decrease data dissemination by requiring Things to perform in situ computations over their data prior to forwarding it. It exploits the capacity of involved Things to identify redundancy in their measurements and forward them upon change only, as done for instance in [Ehyaei et al., 2011, Mietz et al., 2013, Cho et al., 2011]. Another approach is to require each Thing to aggregate or process its own data [Rana et al., 2010]. It is also possible to delegate the task to intermediate, more powerful Things to perform data aggregations [Ferreira et al., 2010, Schmidt et al., 2011]. In-network processing can partially address the communication overhead that would result from the interactions with a large number of Things, by forwarding only compressed values from one Thing to the other and if possible, not forwarding any. The former is more beneficial in the context of on-demand services where aggregating results from several Things and passing them along the network decreases the size of messages being forwarded to the system employing the sensing/actuating services. The latter option performs better for asynchronous sensing systems as it is left to the producer to determine when fresh data should be generated. Asynchronous sensing refers to the case where Things produce data independently from consumers' needs.

Distributed data analysis. The most commonly adopted approach to distributing data analysis is through the *MapReduce* programming model [Dean and Ghemawat, 2008] and its various implementations (e.g., Hadoop by Apache[1]), which are directed towards optimizing the processing of available large scale data with special focus on fault tolerance. Programmers use `map()` and `reduce()` functions to distribute synchronized computations across large scale clusters of machines and schedule inter-machine communication times for efficient use of the network. The map function sends the task to a main node to divide it into sub-tasks and distribute them over the clusters of machines. The reduce collects the answers from all machines and

[1]Hadoop:http://hadoop.apache.org/

returns a combined answer to the user. The model and its implementations exploit the shared-nothing architecture [Barroso et al., 2003] (independent and self-sufficient nodes) on clusters with low end commodity machines (several machines that provide higher performance at less cost than a system comprised of fewer but more expensive high end machines).

The above techniques are beneficial for processing large amounts of data with an infrastructure comprising numerous machines that can take part in the computation process, which is suitable in the context of continuous flows of raw data streams that should be processed. However, it introduces low benefits when data is discrete and requires more simple aggregations, which is the case of on-demand instantaneous sensing/actuating services. In the latter case, in-network data processing among Things themselves can prove more efficient and less costly. In any case, large scale access to discrete data does not impose the same challenges as large scale access to data streams where the issues lie in managing the large volumes of data being continuously generated and the need for continuous processing and possibly storage of the resulting data, which comes with requirements for an infrastructure that can provide powerful computations and large storage capacities. To abstract this infrastructure and provide uniform access to the distributed computational resources as a single resource, the Cloud can prove to be the best solution.

Outsourcing to the Cloud. Empowered by underlying techniques like MapReduce to provide powerful computing as a service, and technologies like Amazon S3 [1] to provide elastic storage as a service, the Cloud is gaining momentum [SENSEI consortium, 2011, Mitton et al., 2012]. For instance, in addition to managing large volumes of data, the default approach to enabling systems to handle large numbers of Things providing their functionalities, is through distributed registries holding their metadata, which can be hosted on the Cloud. In fact, the latter can be exploited to provide three types of services exposing a single entry point regardless of the underlying technologies and the possibly distributed nature of computations [Mell and Grance, 2011]: *Software as a Service (SaaS)*, which enables consumers to use applications running on a cloud with no control of the underlying cloud infrastructure (network, servers, operating systems, storage, etc); *Platform as a Service (PaaS)*, which enables consumers to deploy onto the cloud infrastructure their own appli-

[1] Amazon S3:http://aws.amazon.com/s3/.

cations based on a pay-per-use or charge-per-use basis; *Infrastructure as a Service (IaaS)*, which allows consumers to provision computing resources such as processing, storage, networks, etc., with no control over the underlying cloud infrastructure, but only operating systems, storage, and deployed applications. However, the Cloud comes with money costs either at fixed rates or following the pay as you go models and high bandwidth requirements, which are not always an option. Moreover, it does not manage the communication overhead that results from interacting with the ultra large number of Things, in order to access them and acquire their functionalities.

If not accounting for the high communication and monetary costs, given the fixed nature of the IoT, or in most cases the mobility of Things with predetermined itinerary (e.g., enterprise products), even if a large number of Things can provide their services, the situation remains manageable since the mobility is not random, consequently Things might not have to update their location information as frequently as Things with random mobility. Managing the IoT scale is especially feasible with the Cloud service provisioning, to handle, store, and process the produced data and the metadata of Things. However, when random mobility is introduced, the network becomes too complex to track especially with the anticipated Thing-scale. While it is feasible on the Cloud, storage and computation-wise, to store and process the mobility information of Things, the corresponding communication costs and generated traffic will grow unmanageable. Consequently, introducing and managing mobility of Things becomes truly an issue —especially in the context of location-based services where answers to user queries should be provided in a timely manner— and solely relying on the Cloud as the ultimate solution is not as a scalable option for the reasons stated above. A summary of the approaches presented in this section and their limitations is presented in Table 2.2.

2.3 From the Internet of Things to the Mobile Internet of Things

As discussed above, the IoT today assumes either a fixed environment or controlled predetermined mobility. Research efforts in this context do not provide yet the required setting for a truly *mobile IoT*. A more convenient setting is provided in Mobile Sensing systems where mobility is the rule rather than the exception. In the Mobile

Req	Approach	Idea	Evaluation
Scale	In-network processing	Aggregate data before forwarding or forward on change only	+ Decrease traffic and communication cost - More suitable for data streams and asynchronous tasks not discrete data
	Distributed data analysis	Distributed data computations over numerous machimes	+ Higher computation power + Faster computation time - More suitable for large data volumes not discrete data
	Cloud	Software/platform /infrastructure as a service	+ Scalable Storage +Powerful computations - High bandwidth requirement and communication cost - Comes with monetary cost

Table 2.2. Comparison of existing solutions towards a scalable IoT.

Sensing field, Things are portable objects usually carried by humans and adhere to their mobility patterns, which do not exhibit controlled mobility characteristics. Additionally, with few exceptions, Things are expected to have WiFi or 3G cellular networks, which entails that they can (always) be reachable.

Based on the literature, mobile sensing, also known as people-centric sensing can be divided into two categories [Lane et al., 2008]: i) participatory sensing, and ii) opportunistic sensing. In more detail, *participatory sensing* entails direct involvement of humans controlling the mobile devices, while *opportunistic sensing* requires the mobile device itself to determine whether or not to perform the sensing task. By depending on the custodian's willingness to partake in a sensing task, participatory sensing is not appropriate for the context of real-time instantaneous sensing/actuation requests which can be better satisfied by opportunistic sensing. Orthogonal to the above categorization, mobile sensing can be [Khan et al., 2013]: i) personal sensing, mostly to monitor a person's context and wellbeing; ii) social sensing, where updates are about individuals, especially their social and emotional statuses; or iii) urban (public) sensing, where public data is generated by the public and for the public to exploit. Personal sensing is aimed towards personal monitoring and poses no

(Thing) scale issue as it involves one or just a few devices in direct relationship with their custodian (e.g., [Lu et al., 2009]). For instance, *SoundSense* [Lu et al., 2009] is a system that enables each person's mobile device to learn the types of sounds the owner faces through unsupervised learning on individual phones. Consequently, personal sensing will not be further investigated in this chapter. We proceed in the following to illustrate how social sensing and urban sensing contribute to enabling systems to adapt to the mobility of a large number of Things with special focus on their capabilities to manage large-scale on-demand service requests.

2.3.1 Social Sensing

In social sensing, the Thing or its owner decide what social information to share about the owner or the owner's environment, with individuals or groups of friends [Khan et al., 2013].

Based on the state of the art, Social sensing is mostly participatory, in the sense that it is the custodian of the Thing who determines when and where data should be generated. Consequently, continuously tracking the mobility of Things is not a requirement. The latter is accounted for at the time of the data generation by localizing the Thing only then and tagging (if needed) the data with location information. The scale issue can be perceived in terms of large volumes of produced data and concurrent interactions between users and the social network, managing their information, to provide social data. As illustrated by powerful social networks today, those scale issues can be well handled with powerful processors and elastic Cloud backend storage systems.

In addition to the scale issues above, in opportunistic social sensing, localizing Things continuously can prove more essential.

There are two categories of opportunistic social sensing systems: Systems that require small scale one-to-one relation among two Things, which similar to above, might lead to a scale issue regarding the data being produced (if shared with the system managing the interactions) and concurrent access requests to the system if need be (e.g. [Beach et al., 2008]); or large scale sensing, where requests from individuals are posed for some social information (e.g. restaurant recommendation) at a remote location of interest, which should be satisfied by other individuals at that location (e.g., [Gaonkar et al., 2008]). To acquire needed information from Things

without involving the owners, the system managing the requests must be able to localize them. The common approach to satisfying this requirement and tracking the mobility of a large number of Things, is by requiring all involved Things to register their information and update their location in a repository. Once more, by exploiting the Cloud' elastic backend support, the repositories can be scaled as needed. However, as stated earlier, scalable repositories do not account for the communication and generated traffic overhead that will result from continuously, even periodically tracking an ultra large number of Things.

2.3.2 Urban Sensing

In urban sensing, also known as public sensing, data can be generated by everyone (or their Things) and exploited by everyone for public knowledge, including environment monitoring, or traffic updates [Khan et al., 2013].

In participatory urban sensing, users participate in providing information about the environment by exploiting the sensors/actuators embedded in their Things (which can be smartphones, vehicles, tablets etc.). However data is only generated according to the owner's willingness to participate. Participatory urban sensing is especially characterized by scale issues at the data level, where data is generated by numerous individuals and should be processed and aggregated for knowledge to be inferred, which is where data scaling approaches presented in the previous section are exploited. Similar to social sensing, participatory urban sensing does not require the system to actively manage the mobility of Things as localization needs to take place only when generating the data. Additionally, by decoupling request times for information from information generation, timeless is not an essential requirement. *Ikarus* [Von Kaenel et al., 2011] is an example of participatory sensing, where data is collected by a large number of paragliders throughout their flights. The focus is on aggregating the data and rendering the results on a thermal map.

Opportunistic urban sensing comes with more benefits than its participatory counterpart, since it bypasses the Thing owner. Sensors/actuators provide their data/actions independently, without being restricted to humans' willingness to participate. However, it also comes with more challenges as it becomes crucial to conceive automatic ways to identify available devices, their suitability to perform a sensing/actuating task, and their physical locality. The latter becomes hard to track when

mobility is involved. In fact, in opportunistic urban sensing, location information is a primary criteria in the tasking of devices to provide their measurements. The device selection process is performed either by exploiting a fixed infrastructure support to perform opportunistic discovery or by directly communicating with devices, in a full mobile environment. The latter entails that all devices should frequently update their location information upon displacement.

Fixed infrastructure support. Opportunistic discovery of mobile devices (e.g., mobile phones, bicycles, etc.) can be performed by access points in their communication range as adopted in the *Metrosense* Project [Eisenman et al., 2006] and *Cartel* [Hull et al., 2006] where access points (static or mobile) can opportunistically task mobile devices within the communication range for a sensing activity, or opportunistically retrieve the sensed data they carry. Rather than tasking devices, an alternative is the pull-based approach where sensing/actuating tasks are stored in a repository until a Thing that can complete the task arrives at the location of interest and pulls the task from a nearby access point. Similar to above, the pull-based approach does not require the system to continuously track mobility of devices as it is up to the device to download the tasks. This approach is adopted in *AnonySense* [Shin et al., 2011], where sensing services are registered by all phone carriers willing for their phones to play part in the sensing system. To request measurements, an application creates a task, which is submitted to a repository in charge, where it remains until a phone pulls it. However, the task execution depends on Things willing to download and execute it, which cannot be applied in the real-time on-demand sensing context where timely responses are an essential requirement.

Although it enables systems to manage the mobility of Things without being required to continuously track them, fixed infrastructure support can become too expensive as it requires continuous maintenance, monetary deployment costs and, in many cases, limits the participation of Things to their vicinity to the access points, which is not a scalable option. An alternative would be to adopt an infrastructure-less environment as presented below.

Mobile environment. To the best of our knowledge, there are two approaches, in the literature, to acquiring measurements from mobile Things based on their location in an infrastructure-less environment. The first option is by broadcasting the request

at a location of interest; the second approach is by storing the request in some repository to then be pushed onto the devices when their location matches the requested location.

In the broadcast-based approach, a mobile Thing keeps broadcasting a request in the area of interest for a sensing/actuating task, until another mobile Thing hears the broadcast and agrees to perform the task. In such case the system is not obliged to track the displacements of Things. This approach is introduced in *Bubble Sensing* [Lu et al., 2010], a system that enables the creation of sensing tasks at specific regions with the task broadcasted by requesters or task anchors. Task anchors are other mobile users/Things at the location of interest, which accept to manage the task. Things entering the region, whose functionalities fit the requirements, sense the task and return the information. However, this system entails a strong dependence on the availability of a local anchor and sensing devices willing to participate, simultaneously, at the specified region, which can strongly delay the process of completing the user request and might not be appropriate for on-demand sensing requests where waiting, for the desired information, beyond some minimal duration is not acceptable as the information returned can be of no use or value.

In the push-based approach, a user's requests, i.e., sensing/actuating tasks, are stored in a repository until mobile Things, which can complete the task, come into the location of interest. Afterwards the task is automatically pushed onto the devices. To be able to automatically push tasks onto appropriate devices, the system managing the tasks requires full knowledge of available devices and their mobility information. As stated earlier, although localizing Things is essential when providing location-based services with minimal delay, requiring all Things to update their location information periodically incurs large communication overhead, especially when the anticipated IoT scale is in play. An example of a push based approach is proposed in *PRISM* [Das et al., 2010], where a server is expected to specify the *type of sensors, number* and *locations* where mobile phones should be, in order to execute a task, and the location is specified on two levels: a *coarse-grained* location, and a *fine-grained* location. The task is firstly deployed on candidate phones, which belong to the big specified area and is executed later on, if their location matches the fine-grained specification. To enable the system to keep track of their locations, mobile phones should keep updating the server of their location and available resources, which will grow costly with the repeatedly generated traffic among an ultra large number of mobile

Things and the server. As stated earlier, this is not a scalable option.

There is a trade-off in the systems surveyed above between keeping track of the mobility of Things, the capability of the systems to provide timely responses to on-demand sensing requests, and the resulting communication overhead they induce. Systems, which continuously track mobile Things ignore the communication overhead as they rely on continuous location updates, while systems which require no frequent mobility updates do not provide instantaneous on-demand sensing as information is generated independently of request times. Moreover, the majority of the frameworks are designed to handle participatory sensing requests, where tasks are less time critical than on-demand real-time sensing queries. Moreover, participatory systems are designed by nature to be delay-tolerant while waiting for the Thing owners to participate. A summary of the approaches presented in this section and their limitations is presented in Table 2.3.

2.4 Discussion

Earlier in this chapter, we presented three challenges hindering the realization of the mobile IoT vision. The challenges lie in overcoming the heterogeneous nature of mobile Things and conceiving a scalable solution that can efficiently handle the mobility of an ultra large number Things as the rule rather than the exception (unlike the case in the IoT vision today). However, as we can conclude from our survey, available solutions do not provide the needed scalable approaches in a comprehensive manner. In fact, much of the focus in the IoT is towards data scalability, by exploiting scalable processing and distributed data analysis techniques to manage the raw streams being produced, mostly by sensors and RFID tags. To handle the large number of Things, the default option is to exploit repositories that benefit from the Cloud's elastic backend storage holding the information about Things, which is not sufficient when mobility is involved. What lacks in the current literature is a scalable approach, in the mobile IoT context, that can enable any system employing it to localize mobile Things in a timely manner in order to answer a request for on- demand sensing/actuation at any location of interest (assuming there are appropriate Things at that location) without drastically increasing communication costs, traffic overhead and response delays.

Consequently, the solution we devise throughout this thesis is especially directed

Req	Approach	Idea	Evaluation
Mobility	Participatory social mobile sensing	Provide social data according to user's desire only	+ No need for tracking mobility - No solution for scalable mobility tracking - No Thing scalability issues to address
	Opportunistic social mobile sensing	Opportunistically generate social data independent of custodian	+ Overcome dependence on custodian - Default localization by continuously tracking mobility - High communication cost and traffic
	Participatory urban sensing	Provide sensor data according to user's desire	+ No need for mobility tracking -No (Thing) scalability issue - No solution to timely on demand sensing
	Opportunistic urban sensing: fixed infrastructure support	Opportunistically generate sensor data independent of custodian with no need for mobility tracking by exploiting access points	+ No need for mobility tracking + Overcome dependence on custodian - Expensive deployment and maintenance - Restricted to vicinity to access points
	Opportunistic urban sensing: mobile environment broadcast based	Opportunistically task appropriate sensors by broadcasting requests to neighborhood	+ No need for mobility tracking + Overcome dependence on custodian - Can incur delay for broadcasted requests to be accepted and executed
	Opportunistic urban sensing: mobile environment push based	Opportunistically push tasks onto phones when at location of interest	+ Overcome dependence on custodian - Default localization by continuously tracking mobility - High communication cost and traffic

Table 2.3. Comparison of existing solutions towards a Mobile IoT.

towards removing the obstacle presented above. Our solution is focused on the mobile IoT context, and in particular, location-based services where requests for discrete real-time information/actions must be completed in a timely manner. The solution is threefold: i) to overcome the heterogeneous nature of the mobile IoT, we exploit existing technologies, in particular semantic technologies and SOA decoupling capabilities to abstract and uniformly represent Things and their functionalities. Given the complex nature of the physical world and the various attributes of its constituents we also exploit semantic technologies to model related information; ii) to enable the system employing our solution to scale to the mobile IoT level, we revisit SOA to actively manage and control the interactions with Things and their involvement in a sensing/actuation task according to need, thus decreasing the communication, storage and computation overhead; and iii) to account for the mobility of the ultra large number of Things as an integral part of the solution, we design our approach, to efficiently account for their location in a timely manner without the need to continuously track each and every potential Thing in the network but only the ones allowed, by our solution, to participate in the task at hand.

2.5 Summary

We presented in this chapter the approaches towards handling the IoT challenges, in particular, the IoT heterogeneity, scale, and mobility. The heterogeneity issue is not recent, and numerous efforts have been provided to tackle it by exploiting ontologies to model Things, and/or by exploiting SOA to decouple heterogeneous Things from their functionalities. However, the scale and mobility of Things remain to be further investigated. When the Things become mobile, the philosophy of scaling systems at the data level alone can no longer be applied, especially in the case of on-demand sensing/actuating where knowledge of physical locations of Things becomes a first class criteria to their selection. In such case, continuously updating their location information can prove rather costly and impose unprecedented communication overhead. Managing the mobility of an ultra large number of Things to provide on-demand services has not been fully addressed yet. As illustrated throughout this chapter, the IoT challenges combined remain largely unresolved. Consequently, a global solution has yet to devised to manage a fully mobile environment comprised of an ultra large number of Things that can be interrogated for on-demand services and provide

timely responses without incurring large communication overhead. We proceed in the following chapter to present our approach towards reaching these requirements.

Chapter 3

MobIoT: Service-Oriented Middleware for the Mobile IoT

Designing a middleware system based on a Service-Oriented Architecture entails providing three functionalities: service discovery, service composition, and service access. When it was initially introduced, SOA was directed towards business services where even if millions are registered, there is no need to select and access them all simultaneously. This is not the case for Thing-based services. Requesting information on ambient features in the real world, hereafter referred to as physical concepts, requires the involvement of a large diaspora of sensing/actuation services hosted on mobile Things. Certainly, this will lead to a complex weave of interactions and high communication costs. Firstly, discovery will return a large set of accessible service instances, many of which provide redundant functionalities. Secondly, consumers are expected to have sufficient resources to directly interact with, i.e., access, the providers. Consumers are also expected to have a deep understanding of the sciences behind sensing/actuation tasks, to be able to extract meaningful information from the provided results. Consequently, and despite its numerous benefits, adopting SOA, as is, to design a middleware for the mobile IoT will not scale and will impose a heavy burden on consumers.

In light of the above, SOA interaction patterns should be revisited and executed seamlessly to exempt consumers from directly interacting with providers. Consequently, SOA functionalities, namely, discovery, composition, and access are also to be revisited to better manage the ultra large number of heterogeneous mobile Things

and their services. This is achieved within MobIoT, a middleware system that builds on Thing-Based SOA, a variation of SOA. In accordance with SOA, MobIoT comprises *Composition & Estimation*, *Discovery*, and *Access* components, in addition to a *Registry*. However, the internal functionalities of those components are modified with respect to traditional SOA functionalities. Consequently, the Thing-based SOA functionalities can be executed independent of or in combination with their traditional SOA counterparts. MobIoT also comprises a Query component that handles user queries forwarded from IoT applications. However, for the components to operate and cooperate properly in the IoT context, knowledge of the physical environment is essential and therefore should be modeled carefully by leveraging semantic technologies, ontologies in particular.

3.1 MobIoT Ontologies

Abstracting the physical world into a virtual world requires above average understanding of the complex attributes behind the nature, dynamics, sciences of the real world, and the heterogeneity of the constituents of both worlds (e.g., physical Things, services, physical concepts). Delegating the knowledge creation and representation tasks to developers is a tedious, and in many cases redundant, burden. Semantic technologies, ontologies in particular, are a promising approach towards alleviating that burden. They allow easily portable and widely reusable semantic modeling of knowledge that is both machine interpretable and human understandable. There are four main components that compose an ontology in the Semantic Web: classes, relations, attributes and individuals. Classes are the main concepts to describe (any describable object in the real world). Each class can have one or several children, known as subclasses, used to define more specific concepts. Classes and subclasses have attributes that represent their properties and characteristics. Individuals are instances of classes or their properties. Finally, relations are the edges that connect all the above presented components.

MobIoT leverages the benefits of Semantic Web technologies to emulate real world sciences and integrate them with models of Things and embedded sensors/actuators in order to conceive a comprehensive representation of the real world. Our work on ontologies was published in [Hachem et al., 2011].

There are numerous ontologies that model sensors/actuators. Example ontologies

are OntoSensor [Russomanno et al., 2005], which models sensors and their properties, SSN [Compton et al., 2012], which models sensors, systems of sensors, how a feature is sensed, and sensor observations. 52North[1] provides a model for sensors, events, observation, interfaces to access sensor observations, etc.

The SSN Ontology is generic, it does not model any information related to the actual sensed domain, and requires the domain-specific semantics, units of measurement, etc. to be attached when instantiating the ontology. We partially build our ontologies on SSN as it provides an adequate basis for our ontology design.

However, sensors represent only one IoT component. Recent efforts were made to provide more comprehensive IoT ontologies that model Things, their components, and the features (physical concepts) they measure (e.g., [Wang et al., 2012, Christophe et al., 2011, Christophe, 2012, Villalonga et al., 2010, De et al., 2012]).

Yet, a common limitation to all ontologies above is that they still lack a very important requirement: modeling the physics and mathematics, which are at the core of any sensing/actuation task, as first class entities. In more detail, it is important to correlate various quantifiable and measurable features (physical concepts) through mathematical formulas to define, in a user understandable and machine readable manner the processes behind single or combined sensing/actuation tasks. This correlation enables the system exploiting the ontologies to have a better understanding of the sensing/actuating task at hand and consequently better analyze its outcomes or substitute it more efficiently if need be.

There are several ontologies that attempted to provide a vocabulary for real world sciences. For instance, *Linked Data*[2] provides a large collection of datasets encoded in RDF, spanning different areas that include science, nature, and geography. Another example is *Semantic Web for Earth and Environmental Terminology* (SWEET)[3] suit of ontologies which constitutes a set of upper ontologies for earth sciences, used by NASA. Information modeled in SWEET ontologies includes, among others, earth sciences, environmental knowledge, measurement units, conversions, and mathematical functions.

Since our interest is in scientific models, physics and mathematics in particular, we deem the SWEET ontologies to be most appropriate. We exploit knowledge modeled

[1] 52north:https://wiki.52north.org/bin/view/Semantics/WebHome/.
[2] Linked Data:http://linkeddata.org/.
[3] SWEET:http://sweet.jpl.nasa.gov/.

in the *reprMath* ontology, which provides a representation of mathematical entities such as mathematical datatypes and mathematical operations; and *reprSciUnits* ontology, which models units of measurement. Details on the ontologies are available at http://sweet.jpl.nasa.gov/. We also integrate several concepts from the SSN ontology.

We extend the vocabulary provided by the aforementioned SWEET ontologies and the vocabulary provided by the SSN ontology (restricted to sensors) within the two following ontologies:

- **Device Ontology.** The Device Ontology describes attributes of hardware *devices*, i.e., sensors, actuators, processors, and Things that host them. For MobIoT, it can be regarded as a repository containing descriptions useful for discovery purposes.
- **Domain Ontology.** The (Physics and Mathematics) Domain Ontology models information about real world physical concepts and possible relationships they share among each other. For MobIoT, it can be regarded as the a repository holding information for service composition.

Our ontologies are specified using RDF (Resource Description Language). An RDF graph is a set of interconnected RDF triples. An RDF triple is: `(subject, predicate, object)` $\in$ `(U` $\cup$ `B)x U x(U` $\cup$ `B` $\cup$ `L)`, with U: URI reference, B: Blank node, and L: Literal. If literals are specified in a lexical form, they are known as Plain Literals. They can also have a lexical form and a datatype, in which case they are known as Typed Literals. RDF assumes an infinite set of URIs and an infinite set of Blank Nodes. From a graphical perspective, an RDF triple consists of a subject node, object node and a directed arc (predicate) as follows: $s \xrightarrow{p} o$. The predicate is also known as *property*.

3.1.1 Device Ontology

IoT applications should be network and (science) domain agnostic, as opposed to business-domain specific. In other terms, when developers create their IoT application business logic they do not need to be burdened with the specific details of the types and internal functionalities of sensors/actuators nor the physics behind those functionalities. Therefore, it becomes the task of the middleware to identify what

types of devices to seek in order to provide the needed Thing-based services. For this purpose, the ontology should clearly describe and yet not over-specify device metadata.

The device ontology holds knowledge that is **independent of device deployments**, i.e., information related to the device's actual location. Instead, deployment information is reported by Things, when they register their services. Our goal is to avoid frequently updating the ontology upon displacement of Things, which is an expensive process, especially that ontologies are not designed for highly varying environments. To elaborate on the ontology, we consider that types of IoT *devices*, represented in Figure 3.1 can be divided into four main classes:

- *SensingDevice*: A device that has the capability to measure a physical property of the real world. The class is provided by the SSN ontology.
- *ActuatingDevice*: A device that has the capability to perform an operation on or control a system/physical entity in the real world.
- *ProcessingDevice*: A device that has the capability to perform computation operations on data.
- *CompositeDevice*: A device that consists of at least 2 of the devices above. We represent mobile Things as instances of this class.

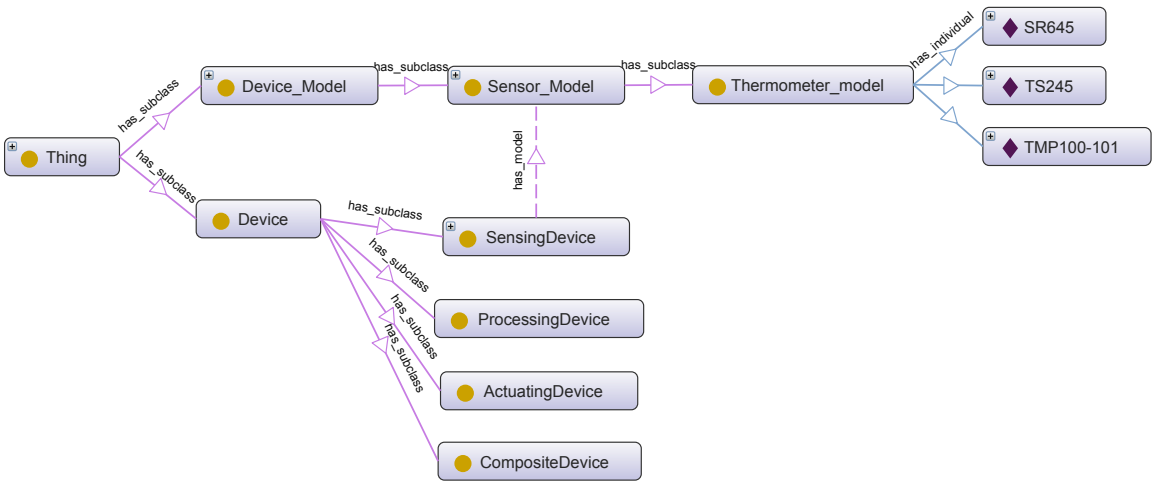

Figure 3.1. A sample of the Device Ontology.

As mentioned earlier, it is important to understand the logic behind sensing/actuation tasks and know, not only what types of devices are needed to execute a task but also how they operate internally to reach that goal. This requires knowledge about the *Sensor/Actuator component type* representing the sensor/actuator internal hardware components; the sensor/actuator *Sampling method* to know the way the sensor samples its environment (e.g., periodic); the *Data transfer method*, modeling the way the sensor/actuator transfers its readings or action results to the middleware (e.g., push); and most importantly, the *Transition function* modeling the process, usually a mathematical operation, used to convert the input phenomenon, detected through a set of electrical signals, into a digital value that bears meaning. It is also important to know the *Manufacturer* of the sensor/actuator, which provides better insight into the quality of the sensor/actuator hardware. This information can be coupled with the device's serial number, and other device information that can be extracted from publicly available Transducer Data Sheets (TEDS) specific to a sensor/actuator model, such as its sensing/actuation range. It is important to mention that although the ontology models low level details related to sensors and actuators, we do not assume the information to be provided by the ontology alone. It is possible to find such details in OS-level APIs abstracting sensors on a smartphone for instance. However, as stated earlier, our ontology is deployment independent, i.e., it is contacted prior to discovering Things as we deem it more beneficial to acquire this information in advance. In particular, such knowledge leads to a more informed discovery and access choice by knowing a priori the characteristics of sensors and actuators to select rather than having to contact Things first to acquire the details on the sensors they host and then decide if they should take part in the sensing/actuation task.

Last but not least, the device ontology models *Physical concepts*, which are the real world properties measured by the sensor (e.g., temperature, wind speed, etc.) or acted on by an actuator. Physical concepts are the glue, binding the **Device Ontology** and the **Physics Domain Ontology**. It is equivalent to `ssn:FeatureOfInterest` class, which models concepts to measure in the SSN ontology.

An example of a thermometer model, an instance of the `Senser_Model` class, along with its metadata is presented in Figure 3.2.

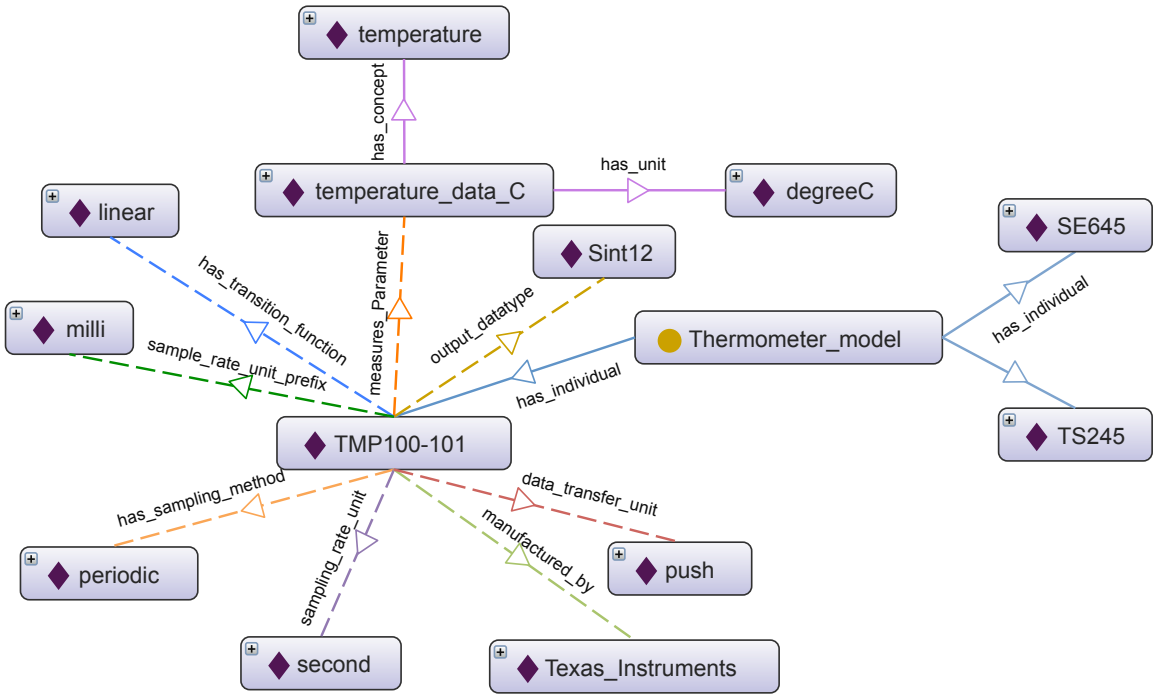

Figure 3.2. Thermometer model example.

3.1.2 Physics Domain Ontology

The Physics Domain Ontology is created with two main goals. The first is to model real world entities, i.e., physical concepts, to assist MobIoT, any other middleware, or any IoT application to extract knowledge about physical concepts. The second is to interconnect physical concepts among each other to emulate the relations they have in the real world. The relations, if available, determine the interdependencies of physical concepts, and allow MobIoT to understand and predict how the variations in one influence the others. The relations among concepts are actually rules of physics, that we model through mathematical formulas, an accurate and interoperable language to present scientific information. In addition to physical concepts, the classes needed to model the information above, illustrated in Figure 3.3, are:

- *Formula*: Mathematical expression that computes a numerical value representing an estimate of the measurement of a physical concept or an approximated action over a physical concept.
- *Mathematical datatype*: Numerical types, usually followed by a measurement

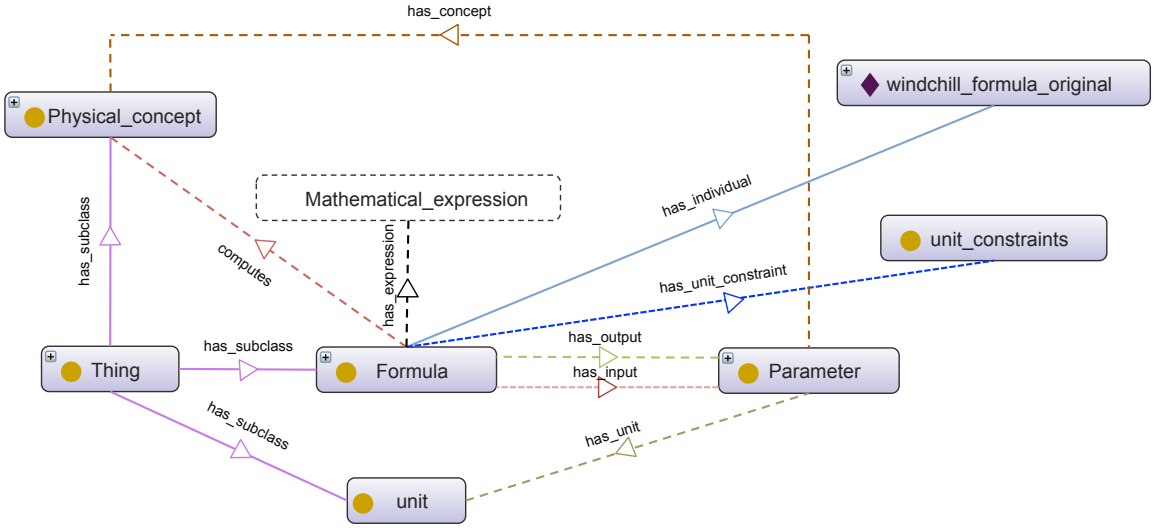

Figure 3.3. A sample of the Domain Ontology.

unit. We exploit numerical types defined in the SWEET *reprMath* ontology (`NumericalEntity, Vector, Matrix`) and extend `NumericalEntity` to model types such as Real, Integer, etc.

- *Unit*: The output unit of a measurement, essential to define the magnitude of numerical quantities (of measurements) in a standard manner. The units are provided by the SWEET *reprSciUnits* ontology. We also use concepts from the SWEET *reprMath* ontology to model mathematical relations in complex units and conversion formulas between different units.

A physical concept can have different measurement units. Temperature for instance, can be measured in Celsius or in Kelvin. It can also be estimated by different formulas with different input/output parameters and different measurement units each. Given the importance of measurement units in understanding a numerical value provided by sensors or provided to actuators, we introduce a novel class: *unit constraints*. The goal is to determine if a formula can have only one output and one input unit per concept, or can have a defined set of such units, or it stands correct for any input/output units, linked to a physical concept. The *unit constraints* class has three instances:

- `unique`. A *unique* constraint determines that the associated formula instance can only have one possible mathematic expression with one specific input unit,

for each input concept, and one possible output unit.

- `set`. A *set* constraint means that the associated formula instance can have several mathematical expressions based on the units of the input concepts, and can have several output units.
- `any`. An *any* constraint means that the associated formula instance has the same expression regardless of the units of the input concepts, and can have several output units.

For instance, on the one hand, the formula $speed = distance/time$ stands correct for any distance unit over any time unit. On the other hand, a windchill formula with temperature in Celsius and wind speed in km/h is different than a windchill formula with temperature in Fahrenheit and wind speed in mph. We bind units and physical concepts provided as input or produced as output by a formula within a *Parameter* class. To determine whether the Parameter is an input to a formula or an output of a formula we use `has_input` and `has_output` properties.

The relation between physical concepts and mathematical formulas is established through a `computes` relation.

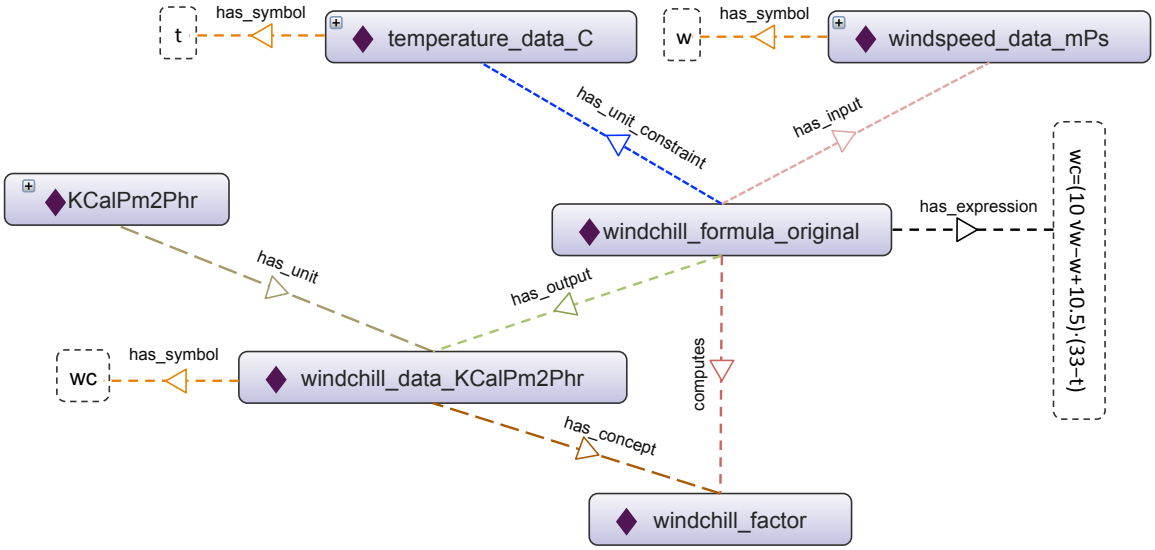

Figure 3.4. Windchill formula example.

Figure 3.4 illustrates the relation between windchill, temperature, and windspeed concepts, in addition to the mathematical formula that determines this relation and

the measurement unit of each; The ontologies we presented are exploited by MobIoT for discovery and composition purposes as will be illustrated through the next section, where we describe the architecture of the middleware.

3.2 MobIoT Architecture

The objective of MobIoT is fourfold: 1) assist developers in creating IoT applications by closely interacting with the dedicated Query component; 2) assist developers in advertising Thing-based services by closely interacting with the Discovery component; 3) manage the large scale of the mobile IoT through a suit of smart computations provided by the Discovery component; and 4) ensure that best efforts are provided to answer a user query forwarded by an IoT application, by exploiting the functionalities of the Composition & Estimation and Access components.

MobIoT operates within a highly dynamic environment specified in the following. The environment comprises an ultra-large number of mobile Things, represented by a set R, with 3G/4G or WiFi communication capacities. The Things host several types of sensors/actuators abstracted by Thing-based services whose metadata is stored in a Registry. A sensor/actuator S_τ has a sensing/actuation range r_τ and a 360° coverage. Consequently, when at location l_i, a Thing hosting S_τ can sense/act on events in a circle C_{l_i,r_τ} with radius r_τ and centered at location l_i. This is known as the boolean disk model [Bai et al., 2006, Wang, 2010]. Note that the sensing/actuation range of each sensor/actuator type and model can be specified in the ontology by a domain expert or in many cases extracted from the sensor/actuator's TEDS.

It is important to mention that, in our current model, we assume the sensing/actuation range to be deterministic, a first order approximation of r_τ, which is not the case in a real world setting. In more detail, a sensor/actuator can actually sense/act on a point beyond this deterministic range and the sensing/actuation probability depends on the distance between the point to sense/act on and the sensor/actuator. Additonally, the sensing/actuation range attenuation is also affected by the environment the sensor/actuator is deployed in. By accounting for this randomness in the sensing/actuation range, it is possible to enhance the sensing/actuation coverage, as illustrated in [Miorandi, 2008] where authors prove that accounting for the randomness in the communication range can actually enhance network connectivity, which can also be applied to sensing coverage. This is also shown in [Gallais et al., 2006]

where points beyond the deterministic sensing range of a sensor can still be sensed with some probability.

Additionally, in the MobIoT environment, Things are expected to be capable of identifying their location as (x, y) coordinates (e.g., through GPS receiver) [Raychaudhuri and Gerla, 2011]. Moreover, as it is very common nowadays for individuals, whether pedestrians or using vehicles, to follow paths specified by the navigation systems on their Things[1], we also expect each Thing in the MobIoT environment to be aware of the path it will follow in the future, and the final destination to reach during a mobility period. Consequently, each mobile Thing κ moves to different locations l_i^κ, starting from l_0^κ and ending at l_{final}^κ.

Note that if navigation systems are not available, an alternative is to exploit learning techniques to predict the most likely path an individual will follow based on his habits and commonly followed routes. In fact, it is shown in [Song et al., 2010a], that there is a 93% potential predictability in human mobility if historical daily mobility records are analyzed. However, this alternative requires further investigation given its need for high computational capabilities to learn and infer information from individuals' historical records. In the case where Things are unable to identify their future paths or their locations, traditional SOA is applied to the totality of the system (IoT application) that exploits MobIoT. In other terms, when Things are unable to provide their location and path information, they will all be required to register their services and all appropriate services will be retrieved for access, thus the system will be unable to leverage the optimizations provided by the Thing-based SOA concretized by MobIoT.

Finally, We expect users to trigger queries through IoT applications, forwarded to the Query component, for remote Thing-based services with the aim of acquiring discrete on-demand data/actions generated at the time of the request. In such case, response should be timely.

The architecture of MobIoT is depicted in Figure 3.5 and the following sections outline the functionalities of and interactions among the different MobIoT components along with their use of the ontologies presented in the previous section. The components are also presented in the component diagram in Figure 3.6 and a sample of the interactions among the components is depicted in the sequence diagram in

[1]Services such as http://www.google.com/mobile/maps/ provide turn-by-turn driving and walking navigation.

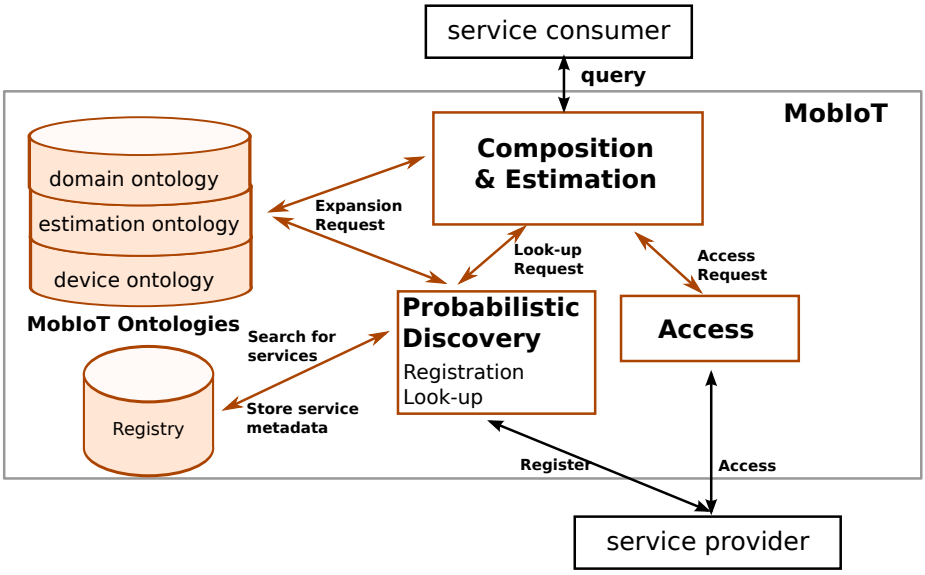

Figure 3.5. The MobIoT middleware architecture.

Figure 3.7. Our initial MobIoT architecture design is published in [Teixeira et al., 2011].

3.2.1 Query Component

The Query component provides two functionalities. Firstly, it enables software developers to exploit available Thing-based services when creating their IoT applications. The only requirement is that they specify the real world information/actions their applications need by adhering to the structure of the MobIoT queries provided by the component. They can then determine any business logic on top of the query outcome. Secondly, it enables IoT applications to better satisfy users' requests for Thing-based services at run time by directly interacting with the Composition & Estimation (C&E) component. In more detail, the queries should contain, similarly to any sensing/actuation query, at least the physical concept to measure/act on and the location of interest. We expect users to specify high-level queries of the form *"Is it windy in Rome?"*. The IoT application should translate those requests to the MobIoT Query structure and forward them to MobIoT. The user can specify additional requirements such as measurement units or measurement accuracy. The Query component extracts the concepts to measure/act on (e.g., windchill factor), the location of interest (e.g., Rome), and the constraints if any are specified, to then forward

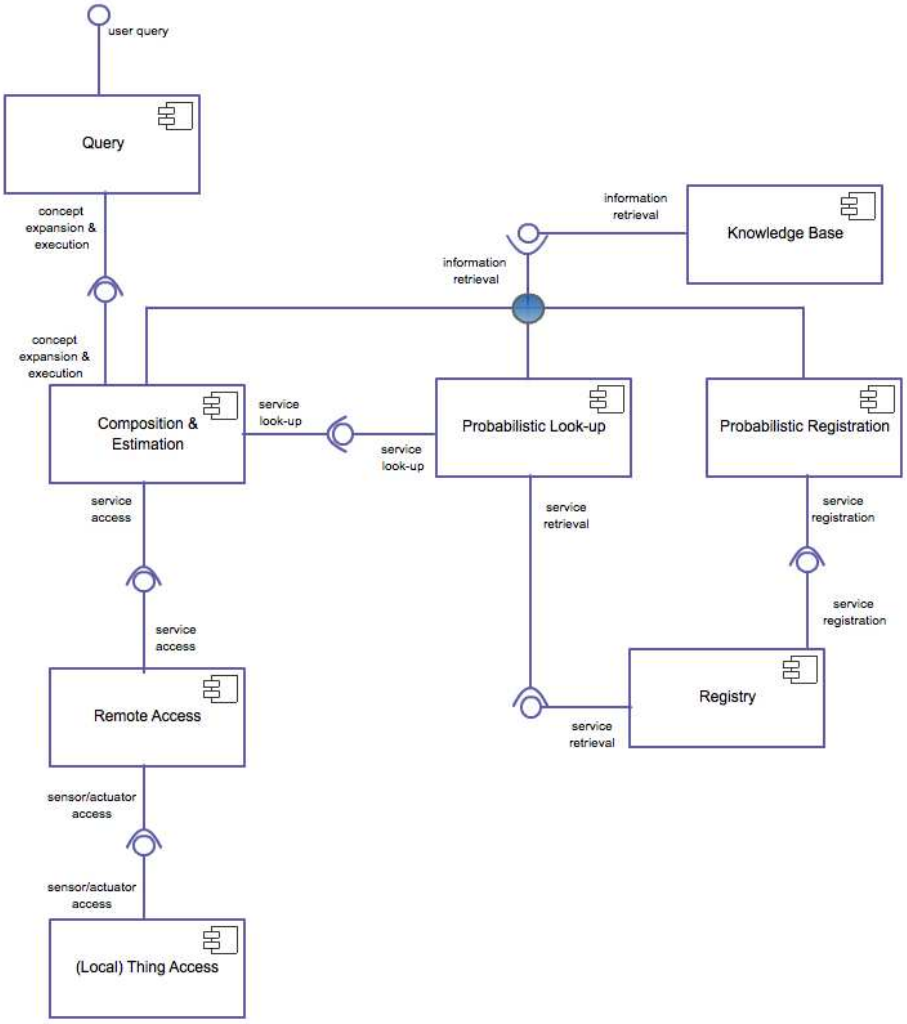

Figure 3.6. The component diagram of MobIoT components.

them to the C&E. The latter identifies the various functionalities needed to provide an answer to the query (e.g., windchill factor, temperature, and windspeed measurement services). A service functionality, also known as service type, refers in the mobile IoT context, to measuring or acting on a specific physical concept. With the cooperation of the Discovery and Access components, the C&E interrogates running services (service instances) of interest, combines their functionalities, and returns the final answer to the Query component.

Our queries are designed based on an object-oriented model, depicted in the class diagram in Figure 3.8. As inspired by SQL[1], a query has the six following components:

[1]SQL: http://www.sql.org.

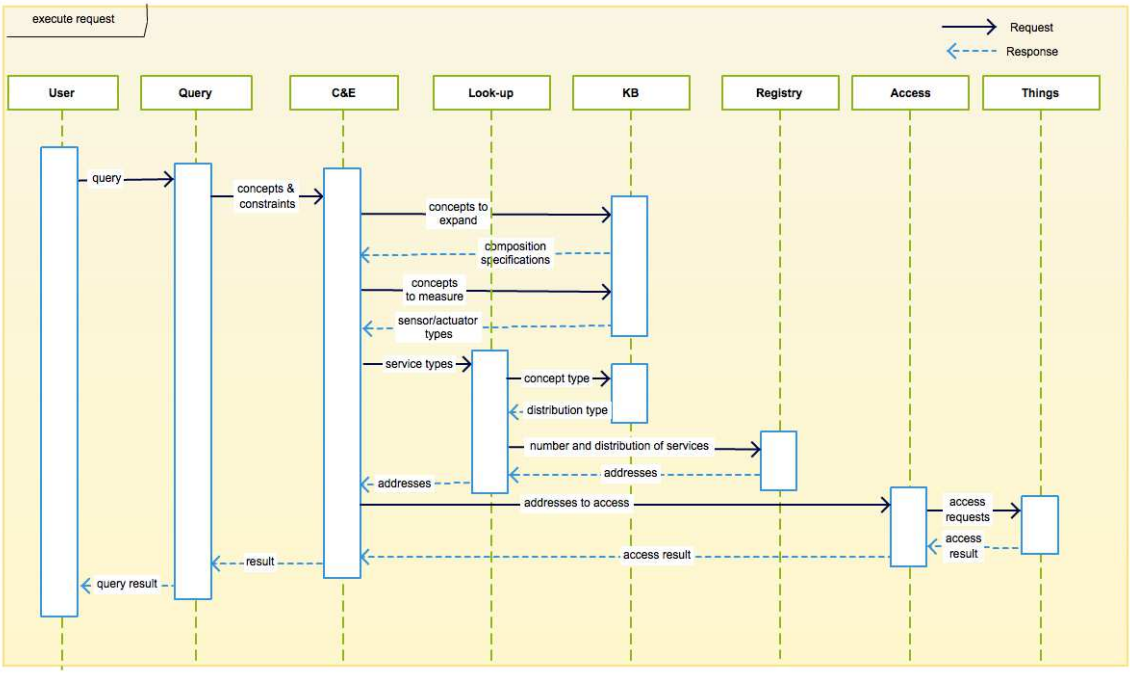

Figure 3.7. The Sequence diagram representing the interactions upon executing a user query.

the `Selector`, selects a (set of) concept(s) to measure/act, and optionally Things with a requested Thing type; `Constraint`, specifies types of constraints and aggregation functions to apply over measurements/action results. The constraint can be a `Where`, an `ORDER BY`, or a `GROUP BY` constraint; `Where`, specifies conditions over the attributes of concepts to measure/act on, their measurement unit in particular. It can also be used to specify conditions over the attributes of sensors/actuators to select, their accuracy in particular; `ORDER BY`, orders the result by the Concept or the Thing to select; `GROUP BY`, groups the result by the Concept or the Thing to select; and `Location`, specifies the location at which the task should take place. It could be a city, a pair of (x, y) coordinates, or a rectangle specified by (x_{min}, y_{min}) and (x_{max}, y_{max}).

The model can be extended by plugging in additional components extracted from existing query languages such as TinyDB [Madden et al., 2005] designed to process streams of sensing data and built on SQL.

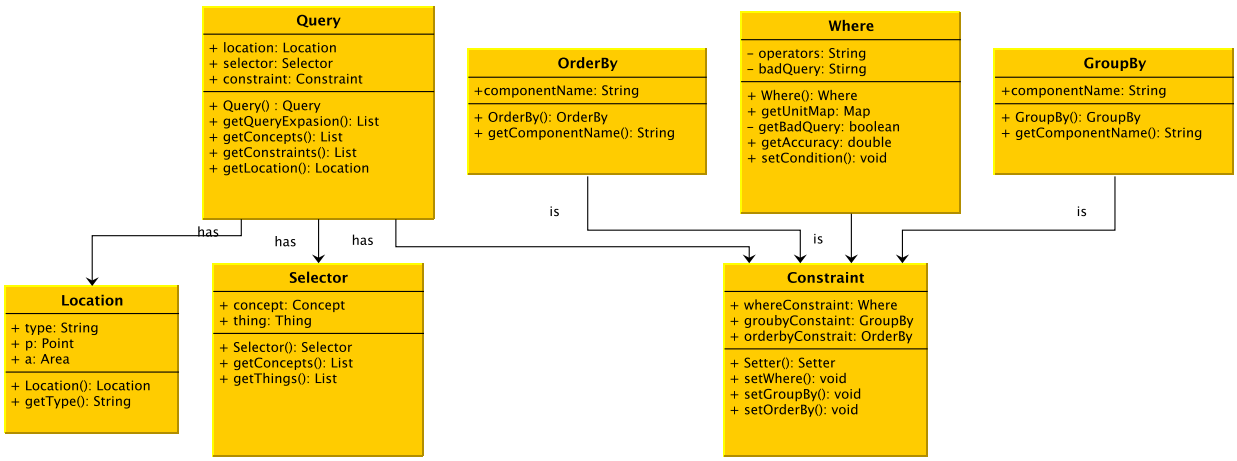

Figure 3.8. The Query object class diagram.

An example of an actual query that instantiates the objects in the model, is as follows:

```
new Query(
      new Selector(
         new Concept(temperature), new Concept(windspeed)),
      new Constraint(
         new Where(
            temperature.unit= Celsius,
            windspeed.unit= meter/second,
            temperature.accuracy = 0.8)),
      new Location(Rome));
```

where `Concept` refers to the physical attribute to measure, `Selector` is used to specify the `Concept(s)` of interest, `Constraint` is used to set the concept's unit and additional constraints such as the accuracy of a measurement, and finally `Location` allows the consumer to specify the location at which measurements/actions should take place. Note that if the values of the concepts and the units do not have matching instances in the domain ontology the query fails, otherwise the Query component forwards the extracted information to the C&E component.

3.2.2 Composition & Estimation Component

The Composition & Estimation component (C&E) provides automatic, semantic composition of Thing-based services. As stated earlier, Thing-based SOA can be combined with traditional SOA functionalities. Consequently, the C&E component can include, in addition to the Thing-based service composition, Web service composition functionalities. However, we restrict our focus to the former where the component exploits laws of physics, modeled in the domain ontology as mathematical expressions, to estimate/modify the state of a physical concept. This capacity is of interest in two cases: a) no service can perform a required measurement/action task directly (based on its atomic functionalities); b) the quality of a desired measurement/action can be enhanced by combining several available measurements/actions. In both cases, the C&E exploits mathematical formulas as composition specifications to estimate/approximate new measurements/actions or provide higher quality measurements by combining lower quality ones. The component constructs a directed graph of physical concepts, identified sequentially, from being the inputs to formulas whose output matches the physical concept in the initial request, or from being the inputs to formulas whose output matches any of the already identified concepts. The concepts are known as expansion concepts. We refer to this step as the *expansion phase*. Afterwards, by closely interacting with the Discovery component, the C&E identifies the types of sensors/actuators to contact and the instances of services (services running on mobile Things) abstracting those types. This represents the *mapping phase*. Finally, by interacting with the Access component, C&E interrogates service instances for their measurements or actions. Once access results are obtained, the C&E computes the formulas and returns the answer to the Query component (*execution phase*). The C&E revisits service-oriented composition by relying on scientific information to determine compositions, and to seamlessly identify and execute composition specifications without burdening consumers with this task. The composition process is presented in more detail in Chapter 5.

3.2.3 Discovery Component

The Discovery component enables Thing-based service registration (to advertise services) and look-up (to retrieve services). In order to handle the ultra large number of mobile Things and their services in the IoT, the component revisits the Service-

Oriented discovery and introduces **probabilistic discovery** to provide, not *all*, but only a sufficient *subset of services that can best approximate* the result that is being sought after. The Probabilistic Discovery functionalities, which will be presented in Chapter 4, are provided by a *Probabilistic Registration* component, and a *Probabilistic Look-up* component.

Probabilistic Registration component. The Probabilistic Registration component is primarily accessed by developers, or Things (if the registration process is designed by the developer to execute automatically) when they request permission to advertise Thing-based services. However, the component does not allow all services to be registered. The decision is based on the physical attributes of the hosts, especially their location and sensing/actuation capacities. To compute the decision, the Registration component is assisted by the Registry, presented in section 3.2.5, with which it communicates upon receiving the registration request.

Probabilistic Look-up component. Even though the Probabilistic Registration component controls which Things register their sensing/actuation services, the number of available services remains large. For this reason, it is essential to further limit the number of service instances to interrogate for their sensing/actuation functionalities. Upon receiving look-up requests from the C&E component, containing the location of interest and sought after service types, the Probabilistic Look-up component selects a subset of service instances hosted on Things that comply with a chosen spatial distribution.

It is noteworthy to mention that we assume trust among Things to truly provide the services they advertise and, to the best of their capacity, provide accurate path/location information. We make the same assumption among developers to provide proper descriptions of the services abstracting sensors/actuators. Nonetheless, we acknowledge the importance of devising trust management and security systems to address malicious intent. Indeed, the system can be subject to various breaches and misleading information. An example of possible security issues takes place when the information of all registered Things is exploited, by the Registry, for malicious purposes. Another possibility is for Things to provide wrongful path and location information, misleading sensor/actuator descriptions, and erroneous measurements and actions that can jeopardize the quality of the functionalities provided by MobIoT.

However, such issues fall beyond the scope of this thesis.

3.2.4 Access Component

The Access component has two benefits. Firstly, it provides an easy to use interface for developers to sample sensors/actuators while abstracting sensor/actuator hardware specifications. Secondly, it revisits Service-Oriented access approaches, where it is usually left to the end-user to access the services and to the application developer to worry about access protocols, by wrapping access functionalities internally and executing access to services transparently. The Access component has two sub-components: the Remote Access component and a Thing Access Middleware, representing the local access component.

The latter is hosted locally on each Thing and executes local access to any type of embedded sensor/actuator through one of the following options:

- *Instantaneous access* is a synchronous request that returns an immediate reply after querying a sensor/actuator for its latest sensing value or action result.
- *Periodic access* is an asynchronous request that returns a reply at a constant rate (until canceled).
- *Event-based access* is an asynchronous request that returns a reply only when triggered by some event of interest (until canceled).

The Remote Access component receives the access requests from the C&E along with the addresses of service instances (returned by the Discovery component). A service address is a URL specifying how the service can be accessed, also known as the service endpoint. The Remote Access component then interacts with the (local) Thing Access Middleware component, on each Thing hosting a discovered service instance, to acquire measurements or to request actions from sensors/actuators abstracted by the services. Afterwards, it returns the results to the C&E.

3.2.5 Registry

The Registry component holds the metadata of sensing/actuation service instances hosted on mobile Things, granted that Discovery component already allowed the

registration to take place. During the registration process, the Registry is contacted by the Discovery component to provide the following information:

- **Number of registered services**. The probabilistic registration decision depends on the number of already registered Thing-based service instances. Only services that provide the same functionality as the registering service and hosted on neighboring Things should be taken into account. Neighboring Things are located in the same region specified in the registration request. The region is a coarse-grained location, specified either as an area with (x, y) coordinates, or by the name of a city or a landmark (e.g. Colosseum).

- **Spatial distribution of Things hosting registered services**. Knowing the number of available service instances is insufficient as we also need to know the approximate locations of their hosts, which is computed by the Registry. However, sending the location information of Things hosting all available services, when there is a very large number of Things involved, will increase communication costs drastically. As an alternative, the Registry computes and returns only the *spatial distribution* of the hosts. The details on the distribution computation are presented in Chapter 4.

Upon registration, each Thing is required to provide at least its unique id, the unique service address, the concept the service measures/acts on, the region it is located in (e.g., Rome), the unit of the measurement the service provides, its location (as a pair of coordinates), its estimated path (as a set of timestamped coordinates), and the registration duration.

Since Things are not expected to register their services indefinitely, the Registry performs the following *clean up* process. After the registration duration expires, if the Registry does not receive a request to keep the service alive, it marks the service as expired and removes it when the next look-up request is received.

Given the dynamic nature of the mobile IoT, the Registry assists the Discovery component with look-up request. The Registry holds the information on the locations of mobile Things when they register their services, therefore, it can estimate their whereabouts at the time of the look-up request. This knowledge is highly important given that observing/acting on the real world is tightly related to the location of Things performing those actions. The Registry can thus know which of them best provides the required action. It supports two levels of location specifications: *coordinate-*

based, and *area-based* specifications. It can also be extended to identify regions within a city, and automatically map them to coordinate-based areas. This information is already modeled in the *geonames* ontology available at www.geonames.org/ontology/, which can be integrated with our MobIoT ontologies.

3.3 Summary

In this chapter, we presented a high-level overview and the architecture of MobIoT, a semantic middleware built on a Thing-based SOA for the mobile IoT. MobIoT is designed to address the large scale, heterogeneity and dynamicity issues of the mobile IoT through the cooperation of the six following components that revisit discovery, composition and access functionalities of SOA: a Query component, a Composition & Estimation component, a Discovery component, an Access component, a Registry and a set of MobIoT ontologies. The Query component handles user queries and hides discovery, composition and access functionalities. The Composition component identifies all types of services needed to better satisfy the user query, and orchestrates discovery and access functionalities. The Discovery component handles the scale and dynamicity issues of the mobile IoT, through a set of probabilistic computations to register/retrieve only a subset of available service instances. The Discovery component is assisted by the Registry, which holds metadata of registered service instances. The Access component wraps access functionalities internally and alleviates that burden from users, initially in charge of this task. Finally, an orthogonal element assisting all components is the set of ontologies, we presented, to model and exploit not only semantic information about Things, sensors and actuators, but also physics and mathematics, which are at the core of sensing/actuation tasks. It is the key element towards an interoperable IoT. We proceed in the following chapters to describe, in detail, the theoretical and technical aspects of the probabilistic Discovery and Composition & Estimation functionalities.

Chapter 4

Probabilistic Thing-Based Service Discovery

There are two different approaches that govern sensing/actuating networks: a) few but highly accurate, powerful, and expensive sensors/actuators; b) numerous, and cheap but less accurate sensors/actuators. The latter applies to the mobile IoT environment, which comprises a large number of mobile Things hosting cheap sensors/actuators. In such an environment, numerous Things will be within the same area and provide similar functionalities simultaneously, leading to overlapping coverage of the area. Moreover, in the mobile IoT, numerous scenarios require only partial sampling of the area of interest to monitor/act on, as opposed to measuring/acting on every single point within that area. Based on those two facts, it becomes sufficient to select only a subset of the participating Things to provide the services they advertise and hence, better handle the large number of Things. In this chapter, we use service and service instance interchangeably to refer to services running on Things.

Advertising and selecting services are the building blocks of Service Discovery, a key functionality in Service-Oriented Architecture. In the Mobile IoT, service discovery revolves around services that abstract sensors/actuators hosted on mobile Things carried by humans in their smartphones, gadgets and even their clothing, which makes them subject to human mobility patterns.

Knowledge of the mobility patterns is crucial to identifying the appropriate subset of services to select. Their mobility determines which portions of the area the host Things can actually cover while in motion. It also makes it possible to estimate

Symbol	Meaning
	Mobility
D	Diffusion constant
v	Speed of mobile Things
	Deployment/Area
A	Total area of deployment
a	Area of interest
C_{l,r_τ}	Circle of center l and radius r_τ
σ_{l_i,r_τ}	Largest square inside circle C_{l_i,r_τ} with edges parallel to the coordinate axes
$\Sigma_{l_i,d^i_{max}}$	Smallest square outside $C_{l_i,d^i_{max}}$ with edges parallel to the coordinate axes
Q	Grid constructed over a
q_i	Square in the grid Q. $q_i \in Q$
r_i	Distance of the edge of square q_i from the event of interest
	Device
T	The set of all sensor/actuator types
R	The set of all registered services
k	A mobile Thing hosting a registered service $\in R$
τ	A type of sensor/actuator, $\tau \in T$
R_τ	The subset of R with services abstracting sensors/actuator of type τ
S_τ	Sensor/actuator of type τ abstracted by services $\in R_\tau$
e^j_τ	Expansion set $e^j_\tau \subseteq T$ containing concept types that together can substitute a concept measured/acted on by a sensor/actuator of type τ
E_τ	Set of all expansion sets that can each replace a sensor of type τ
r_τ	The identical coverage (sensing/actuating) range r_τ of Things hosting sensors/actuators of type τ

Table 4.1. Discovery Related Notations (a).

which of the Things will be at the same location and provide the same functionalities. By exploiting the functional similarities in Thing-based services, the locations and mobility patterns of their hosts, and the overlapping coverage they provide, we conceived an approach towards a scalable discovery in the mobile IoT, for each of the discovery phases, referred to as: *Thing-based service registration* and *Thing-based service look-up*. The approach builds on two key ideas: *subset selection* and *mobile coverage*. We proceed in the following with an overview of the background and related works that build on those ideas along with necessary definitions. All necessary annotations in this chapter are presented in Tables 4.1 and 4.2.

Symbol	Meaning
	Registration
t_0	Time at which the new Thing joins the network
N^i_{range}	Number of services in R hosted on Things that can reach location l_i at time $l_i.t$
l_i	Location of the new Thing κ^1 at time t_i
l^k_0	Location of device k at time t_0
l^k_i	Location of device k at time t_i
δ^k_i	Displacement of device k during $t_0 \rightarrow t_i$
d^i_{max}	Maximum distance from beyond which a device can not reach l_i at time t_i
	Event
e	Event to measure/execute
$l_e : x_e, y_e$	Location of the event to measure/execute
	Lookup
t_r	Time of the look-up request
I	Set of services with interpolated locations at time t_r
$DIST$	Spatial distribution of devices hosting services to select.
$\|s\|$	Size of the subset of services to select
λ	Number of services from each square q_i
f	Decay function of the event.
d_u	Maximum distance from beyond which the event can almost no longer be detected

Table 4.2. Discovery Related Notations (b).

4.1 Background

In Service-Oriented discovery, service registration is accomplished without taking into account the density of registered service instances, the redundancy of their functionalities, or the overlapping coverage they provide ([Karnouskos et al., 2010, Guinard et al., 2010, Tsiatsis et al., 2010]). In fact, as stated earlier, registering services hosted on Things, in SOA, entails registering services hosted on **all** Things willing to participate. Subset selections and mobile coverage requirement were better addressed in the WSAN and mobile sensing domains, although insufficiently, as will be presented in the following.

Subset selection. In WSAN, several attempts were made to decrease the number of active sensors by adopting duty cycling or device selection techniques where only some provide their services while others sleep. However, even though sensors do

not provide services at one time or the other, they are still known to the system in charge (e.g., [Ahmed et al., 2011, Musolesi et al., 2010, Wang et al., 2009, Chatterjea and Havinga, 2008]). An alternative is to select sensors based on user requirements and a specified number of sensors that are assumed to be sufficient. In [Perera et al., 2013], this logic is exploited within CASSARAM, a sensor search, selection and ranking model. CASSARAM selects a subset of available sensors based on users specifying their preferences, which include accuracy, reliability, response delay, and precision. Afterwards, CASSARAM finds all sensors that can provide the needed measurements, ranks them according to the preferences and returns a number of sensors equal to the size specified by the user. The approach supports a large number of QoS requirements and is highly scalable. However it is designed for static sensors and requires high processing times, while we are interested in subset selections, with minimal delay, of mobile sensors (hosted on mobile Things) based on the mobile coverage they provide.

Mobile coverage. Given the nature of sensing/actuating tasks, a very important criteria that should not be overlooked is *area coverage*, i.e., the portion of the area that can be directly sensed/acted upon by Things. Most efforts in the literature focus on the coverage of static sensor or sensors hosted on robotic entities with controlled mobility [Wang et al., 2007, Li et al., 2007, Aziz et al., 2007, Loscri et al., 2013], while our interest is in coverage provided by Things with "free" mobility, i.e., Things whose displacement is not specifically directed towards enhancing coverage.

A common approach to account for coverage with free mobility is through probabilistic mobility models that estimate the possible future displacement of Things as done in [Keung et al., 2010, Ruan et al., 2011, Ahmed et al., 2011]. However, those solutions adopt general mobility models, i.e., models not specifically designed to report the statistical features that humans exhibit in their mobility, which we require in our approach. Consequently it was not possible to build on those solutions. The need for mobility models that estimate human mobility patterns stems from the fact that mobile Things we focus on in our solution are subject to human displacements. Rather than requiring all individuals to continuously update their location information, which can be rather costly, models that probabilistically estimate their displacements are a better alternative. Additionally, for mathematical clarity, extensibility and the ability to analyze it, it is important that the model be analytically

tractable.

Human mobility. To integrate human mobility with mobile coverage computations, the mobility model should report the following statistical features:

- **Truncated power-law flights and pause-times.** It is shown that humans follow flights consisting of straight line trips, after which they stay at the destination for a time duration (pause period). The length of the flights and the duration of their pause periods, at each destination, have a truncated power-law distribution [Gonzalez et al., 2008, Brockmann et al., 2006]. In other words, human mobility consists of many short flights and few long ones.

- **Truncated power-law inter contact times (ICTs).** The time duration between two consecutive contacts between the same persons has a power law distribution followed by an exponentially decaying tail, after a certain period of time [Chaintreau et al., 2007, Karagiannis et al., 2010].

The above features are satisfied with the **Truncated Lévy Walk (TLW)** model introduced by in [Rhee et al., 2008]. TLW is a continuous Random Walk with truncated Pareto distributions for displacement and pause periods with the following characteristics:

- **Motion speed**: Motion speed is, on average, constant for the same motion path.

- **Time interval of each displacement**: The time interval of each displacement is directly related to the velocity and length of the displacement. In [Rhee et al., 2011], it is presented as $t_i^f = k\xi_i^{1-p}, 0 \leq p \leq 1$ where k and p are constants and ξ_i is the length of the displacement. When p is 0, displacement times are proportional to displacement lengths which models the movement at constant velocity.

- **Direction of the displacement**: Similarly to the Random Walk model, the direction is uniformly distributed with a direction angle $\theta_i \in [0, 2\pi)$.

- **Length of the displacement**: In TLW, the length of the displacement is assumed to have a Lévy distribution. the density function of a Lévy distribution

is:

$$f_l(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} e^{-itx-|ct|^{\alpha}} dt \tag{4.1}$$

If $\alpha = 1$, the distribution becomes a Cauchy distribution. For $\alpha = 2$, it becomes a Gaussian distribution with $\sigma = \sqrt{2c}$. For $\alpha \leq 2$, $f_l(x)$ can be approximated by $\frac{1}{|x|^{1+\alpha}}$. Since it is a *Truncated* Lévy Walk, the range of the displacement length does not vary between $(-\infty, +\infty)$ but between $(0, \tau_\xi)$, with τ_ξ being the maximum allowed displacement length.

- **Pause interval**: TLW includes a pause time, which takes place after the end of each displacement. The pause time T_p has a Lévy distribution, which, similarly to the displacement length, can be approximated to $\frac{1}{T_p^{1+\gamma}}$ with $T_p \in (0, t_p)$, with t_p being the maximum allowed pause interval.
- **Complete displacement time**: This time, Δt_s is the sum of the displacement time and the pause time.

In addition to representing the statistical features of human mobility through the truncated Lévy based flights and pause durations, TLW preserves its analytical tractability and has a closed form mathematical expression that we can build on.

Given their probabilistic nature, using mobility models results in *approximate* computations of coverage as they only build on estimations of the motion of Things. It is also possible to base the coverage computations on more concrete information about anticipated displacements, provided by Things themselves, and therefore obtain more accurate results. Bearing this in mind, we conceived a deterministic registration approach that computes coverage based on full knowledge of the anticipated displacements of Things (hosting registered services), followed by a probabilistic optimization that builds on the TLW model presented above. This work is published in [Hachem et al., 2013b,a].

4.2 Thing-Based Service Registration

In order to handle the large number of Things in the Mobile IoT, the participation of their sensing/actuating services should be controlled according to need. To that end, Things are denied from registering their services as long as those with already registered services provide sufficient coverage of the area of interest. The area of

interest is the area a Thing κ_1 can cover along its future motion path, known as full coverage. Coverage of the area of interest provided by the hosts of already registered services, known as actual coverage, is considered sufficient, if it exceeds a minimum required portion of the path of κ_1, in which case, κ_1 does not need to register its sensing/actuating service. Otherwise, κ_1 should register its service. The minimum required coverage (threshold) is concept-specific and should be specified in the MobIoT ontologies. Full coverage, minimum required coverage, and actual coverage are defined as follows:

Definition 4.2.1. Full Coverage. Given a mobile Thing κ_1 moving in a two-dimensional region A, from location l_0 at time t_0 to location l_{final} at time t_{final} with intermediary locations l_i stamped with times t_i, the union of the circles centered at each location l_i with radius r_τ (the coverage range of κ_1), is the full coverage area a_f that will be provided by κ_1 throughout its path, such that $a_f = \cup_{l_0}^{l_{final}} C_{l_i,r_t}$.

Definition 4.2.2. Minimum Required Coverage: The minimum required coverage is the minimum percentage of a_f that should be covered by κ_1 or any other mobile Thing.

Definition 4.2.3. Actual Coverage: The actual coverage is the percentage of a_f that can be covered by Things, hosting already registered services, other than κ_1.

Note that the full area coverage at all locations l_i is computed simultaneously regardless of the timeline, i.e., time t_i, as it does not affect the final outcome, given that that it is an estimation of the coverage of all possible locations combined along the whole path.

When computing actual coverage, we take two cases into account:

1. *Direct coverage*: Only the set of Things hosting a service of similar type to κ_1's service is taken into account when computing coverage.

2. *Expanded coverage*: The types of sensor/actuator measurements or actions that can together substitute (or approximate) a measurement (or action) by a sensor (or an actuator) S_τ are determined through an *expansion process*, presented in Chapters 3 and 5, by interacting closely with the domain ontology. Services abstracting the corresponding sensors/actuators are known as *expansion services* and concepts measured by those services are known as *expansion concepts*. Each

possible combination of expansion services is referred to as an expansion set e_τ^j with $E_\tau = \cup_j e_\tau^j$ being the set of all expansion sets for a sensor/actuator type τ. For example, a wind-chill concept (measured by a sensor of type τ) can be expanded into (substituted by) a temperature concept (type τ_1) and a windspeed concept (type τ_2). In such case, $e_\tau^1 = \{\tau_1, \tau_2\}$.

Additionally, unlike full coverage computation, the timeline in displacements, i.e., the times at which κ_1 or a registered Thing κ cross a location l_i, plays an essential role in the actual coverage computations. Consequently, knowledge of the future displacements of all Things is a must. This information can be provided by the Things themselves during their registration, which we refer to it as *deterministic Thing-based registration*. This information is stored in the Registry along with the service metadata.

4.2.1 Deterministic Thing-based Registration

The deterministic Thing-based registration builds on paths intersections as the basis for actual coverage computations, to generate the registration decision. We formulate the problem to solve in this context as follows:

> *Given a new device κ_1 wishing to register a hosted service abstracting a sensor/actuator of type τ, in an environment with a known topology, consisting of a set R of location-aware mobile Things hosting different types of sensors/actuators, determine if the path of κ_1 will intersect with the paths of other Things hosting similar registered services, and decide if κ_1 should register its service, based on a required coverage threshold.*

The paths provided by Things upon registration are sets of timestamped coordinates, which might not match the timestamps in κ_1's estimated path. For this purpose, the Registry estimates the location of the Things (based on their paths) at the exact time t_i at which κ_1 is supposed to cross l_i on its path. The estimation is done through an interpolation function (`Algorithm 3`) that uses the locations of a registered Thing κ at time t_j^κ and t_{j+1}^κ such that $t_j^\kappa \leq t_i < t_{(j+1)}^\kappa$ to estimate its location at time t_i. The registration decision generation is presented in `Algorithm 1`, which uses methods presented in `Algorithms 2 - 4`.

Algorithm 1 Deterministic Registration decision process

Require: $L, r_\tau, \tau, threshold, R, E_\tau$
Ensure: $decision \in \{true, false\}$
1: let $C \leftarrow \emptyset$ {C `is the set of covered location`}
2: **if** L *is empty* **then**
3: $decision \leftarrow false$
4: **else**
5: **for** *each* $l_i \in L$ **do**
6: $intersect \leftarrow false$
7: **for** *each* $s \in R_\tau$ **do**
8: $L^s \leftarrow s.path$
{`Check if` s `is at` l_i `at time` t_i.}
9: **if** $intersects(l_i, t_i, L^s, r_\tau)$ **then**
10: $intersect \leftarrow true$
11: $C \leftarrow C \cup \{l_i\}$
12: *break* {`break if intersection found and check next location`}
13: **end if**
14: **end for**
15: **if** $intersect = false$ **then**
16: **for** *each* $e_\tau^j \in E_\tau$ **do**
17: $intersect \leftarrow false$
18: **for** *each type* $\in e_\tau^j$ **do**
19: **for** *each* $ss \in R_{type}$ **do**
20: $L^{ss} \leftarrow ss.path$
21: **if** $intersects(l_i, t_i, L^{ss}, r_{type})$ **then**
22: $intersect \leftarrow true$
23: *break*
24: **else**
25: $intersect \leftarrow false$
26: **end if**
27: **end for**
28: **end for**
29: **if** $intersect = true$ **then**
30: $C \leftarrow C \cup \{l_i\}$
31: *break*
32: **end if**
33: **end for**
34: **end if**
35: **end for**
36: $c \leftarrow getCoveragePercentage(L, C, r_\tau)$
37: **if** $c < threshold$ **then**
38: $decision \leftarrow true$
39: **else**
40: $decision \leftarrow false$
41: **end if**
42: **end if**
43: **return** *decision*

The algorithm firstly checks, for all locations l_i, if a Thing κ intersects with κ_1 at l_i at time t_i (Line 9) using the methods presented in `Algorithm 2`. If so, it adds the location l_i to the set of covered locations C (Line 10-11), and moves to the next location. If not, it checks the Thing hosting the next service in the list R_τ. If all services have been checked, the algorithm determines if the location can be covered by expansion (Line 15-31). For each expansion service, it checks if its hosting Thing intersects with κ_1 at l_i at time t_i (Line 21). If so, it adds l_i to the set of covered locations C (Line 30) and moves to the next location. The process repeats until all

locations on κ_1's path have been checked.

To decide whether or not the path of a Thing κ intersects with that of κ_1 at l_i at time t_i, `Algorithm 2` takes l_i, t_i, the list L^s of timestamped locations representing the path of κ, and a coverage radius as input and determines that the path of κ intersects with l_i if its distance from l_i at time t_i is less than r_τ (Line 4-5). This requires an interpolation to estimate the location of κ at time t_i (Line 3). The interpolation method is presented in `Algorithm 3` which computes the following expressions:

$$x_{t_i} = x_{t_j} + \frac{t_i - t_j}{t_{j+1} - t_j} * (x_{t_{j+1}} - x_{t_j}) \tag{4.2}$$

$$y_{t_i} = y_{t_j} + \frac{t_i - t_j}{t_{j+1} - t_j} * (y_{t_{j+1}} - y_{t_j}) \tag{4.3}$$

Finally, `Algorithm 1` computes the percentage of κ_1's path that is covered by other Things (Line 36). The result is then compared to the required coverage threshold. If it is below the threshold, the registration decision is positive (Line 37-38). If it is above, the registration decision is negative (Line 39-40)

Note that if the path of the incoming Thing comprises one location only, i.e., the Thing is static, the deterministic registration approach can still perform properly by computing intersections between the paths of Things hosting registered services and the unique location of the registering Thing. However, if no location information is provided, the deterministic registration computation will not take place and the decision is directly set to false.

To determine the coverage percentage, it suffices to divide the length of the covered segments of κ_1's path by the total length of the path as presented in `Algorithm 4`. The algorithm computes the length of the covered segments based on the coverage radius r_τ at each location $l_i \in C$ and then divides the result by the total length of the path. Since we adopt a boolean disk model, the length of each segment should be $2 * r_\tau$. However, the covered locations can have overlapping areas (such as the overlapping area between the circles around location l_0 and location l_1 in Figure 4.1). The overlaps are accounted for by considering, at each location $l_i \in C$, the full range r_τ between l_i and l_{i+1} and only the distinct portion of the covered segment between l_i and l_{i-1} (`Line 17`). The general expression to compute the segment length while

Algorithm 2 `intersects` method

Require: l_i, t_i, L^s, r_τ
Ensure: $intersect : true, false$
1: **for** $l_i^s \in L^s$ **do**
2: **if** $t_j^s \leq t_i < t_{j+1}^s$ **then**
3: $l_i^s \leftarrow interpolate(l_j^s, l_{j+1}^s, t_i)$
4: **if** $(l_i^s.x - l_i.x)^2 + (l_i^s.y - l_i.y)^2 \leq r_\tau^2$ **then**
5: $intersect \leftarrow true$
6: $break$
7: **else**
8: $intersect \leftarrow false$
9: **end if**
10: **end if**
11: **end for**
12: **return** $intersect$

Algorithm 3 `interpolate` method

Require: $locationBefore, locationAfter, t$
Ensure: $newLocation \leftarrow 0$
1: $alpha = \frac{t - locationBefore.time}{(locationAfter.time - locationBefore.time}$
2: $newLocation.X = locationBefore.x + alpha * (locationAfter.x - locationBefore.x)$
3: $newLocation.Y = locationBefore.y + alpha * (locationAfter.y - locationBefore.y)$
4: **return** $newLocation$

accounting for overlaps is:

$$d = d + r_\tau + (\sqrt{(l_i.x - l_{i-1}.x)^2 + (l_i.y - l_{i-1}.y)^2} - r_\tau) \tag{4.4}$$

$(\sqrt{(l_i.x - l_{i-1}.x)^2 + (l_i.y - l_{i-1}.y)^2} - r_\tau)$ presents the portion of the segment between l_i and l_{i-1} that is only covered by l_i. After simplification, the expression becomes:

$$d = d + \sqrt{(l_i.x - l_{i-1}.x)^2 + (l_i.y - l_{i-1}.y)^2} \tag{4.5}$$

There are two special cases: i) if the location is l_0, there is no coverage to take into account before l_0 ; and ii) if the location is l_{final}, there is no coverage to into account after l_{final}, Consequently, the length of the covered segment at l_0, assuming it was added to

Algorithm 4 `getCoveragePercentage` method

Require: L, C, r_τ
Ensure: p
1: $l_{old}.x \leftarrow l_0.x, l_{old}.y \leftarrow l_0.y, x_{old} \leftarrow 0, y_{old} \leftarrow 0, td \leftarrow 0, d \leftarrow 0$
2: **for** $l \in L$ **do**
3: $\quad td \leftarrow td + \sqrt{(l.x - l_{old}.x)^2 + (l.y - l_{old}.y)^2}$
4: $\quad$ **if** $l \in C$ **then**
5: $\quad\quad$ **if** $l = l_0$ **then**
6: $\quad\quad\quad d \leftarrow d + r_\tau$
7: $\quad\quad$ **else if** $l = l_{final}$ **then**
8: $\quad\quad\quad$ **if** $\sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2} \geq 2r_\tau$ **then**
9: $\quad\quad\quad\quad d \leftarrow d + r_\tau$
10: $\quad\quad\quad$ **else**
11: $\quad\quad\quad\quad d \leftarrow d + \sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2} - r_\tau$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **else**
14: $\quad\quad\quad$ **if** $\sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2} \geq 2r_\tau$ **then**
15: $\quad\quad\quad\quad d \leftarrow d + 2r_\tau$
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad d \leftarrow d + \sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2}$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end if**
20: $\quad\quad x_{old} \leftarrow l.x$
21: $\quad\quad y_{old} \leftarrow l.y$
22: $\quad$ **end if**
23: $\quad l_{old}.x \leftarrow l.x$
24: $\quad l_{old}.y \leftarrow l.x$
25: **end for**
26: $p \leftarrow \frac{d}{td}$
27: **return** p

C, is always equal to r_τ (`Line 6`) regardless of possible overlaps. The overlaps will be accounted for when computing the length of the covered segment at location l_1 (assuming l_1 is covered). The length of the covered segment at l_{final}, if there is an overlap with the previous covered location, is $d = d + \sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2} - r_\tau$, since unlike the computation in Equation 4.5, there is no coverage after l_{final} to account for.

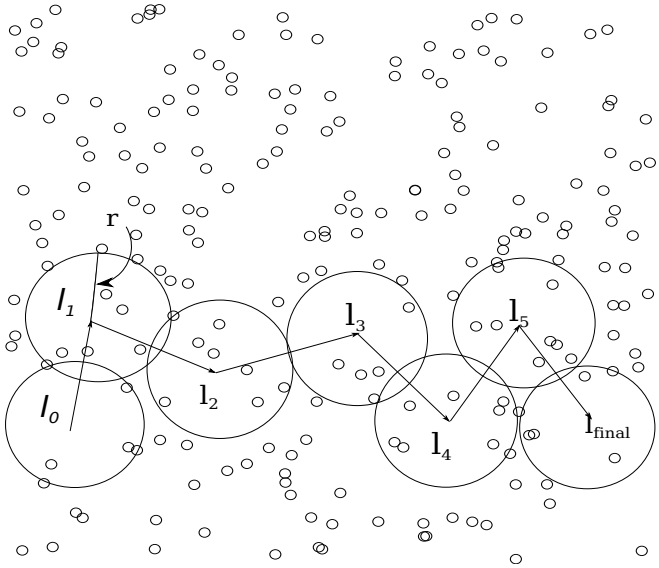

Figure 4.1. An example of the coverage provided by a mobile Thing at the locations l_i throughout its path.

`Algorithm 4` takes a list of timestamped locations representing κ_1's path, the set of covered locations on the path, and the coverage range as inputs. It produces the coverage percentage as output. `Algorithm 4` starts by computing, for each location l_i on κ_1's path, the total distance between l_i and l_{i-1} (Line 3). Afterwards, it checks if l_i is covered, i.e. if $l_i \in C$ (Line 4). It then checks if any of the special cases presented above applies. If l_i is actually l_0, then the coverage distance is $d = d + r_\tau$ (`Line 6`). If l_i is l_{final} and there is no overlap with the last covered location before l_{final} in C, the covered distance is $d = d + r_\tau$ (Line 8-9) otherwise, it is $d = d + \sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2} - r_\tau$ (Line 11). If l_i is an intermediary location, the algorithm checks if there is no overlap with the last covered location before l_i (Line 17), leading the covered distance to be $d = d + 2 * r_\tau$, (Line 16) otherwise it is $d = d + \sqrt{(l.x - x_{old})^2 + (l.y - y_{old})^2}$ (Line 17). To keep track of the last checked location on κ_1's path when computing the total distance, and the last checked location in C to compute coverage distance, the currently checked location coordinate values are assigned to temporary variables (Line 20-24) before moving to the next location. Once all locations are checked, the ratio of the covered distance to the total distance is computed and returned as the coverage percentage (Line 26).

Complexity analysis. `Algorithm 1` terminates when a decision is reached after the end of the outermost loop (Line 5 - 31). All the elements in the loops (both inner and outer) are finite and therefore the loops are sure to terminate. `Algorithm 2` iterates over the locations in L^s, which is a finite set, and therefore it is sure to terminate. `Algorithm 3` performs simple computations with no iteration, consequently, it is sure to terminate. Finally, `Algorithm 4` iterates over the locations in L, which is a finite set, and is therefore sure to terminate.

`Algorithm 1` depends on the number of services in R_τ, the number of locations in L, the number of expansion sets in E_τ, the number of expansion types in each expansion set e_τ, and the number of services in R_{type}. `Algorithm 1` also includes multiple invocations of `Algorithms 2 to 4`.

`Algorithm 2` depends on the number of locations in L^s, it has a time complexity $\mathrm{O}(L^s)$. `Algorithm 3` has a constant time complexity $\theta(1)$.

Finally, `Algorithm 4` depends on the number of locations (on the path of κ_1) in L, it has therefore a time complexity $\mathrm{O}(|L|)$.

Consequently, `Algorithm 1` has a time complexity $\mathrm{O}(|L|(|R_\tau| + |R_{type}||E_\tau||e_\tau^j|))$.

`Algorithm 1` uses three sets, C, L and R and requires no preprocessing, or complex data structures. Therefore, `Algorithm 1` has a space complexity $\mathrm{O}(|C| + |L| + |R|)$. In the worst case, all the locations of the path of κ_1 will be covered, leading to $|C| = |L|$. The resulting space complexity is $\mathrm{O}(|L| + |R|)$.

`Algorithm 2` uses one simple set, L^s, and has a space complexity $\mathrm{O}(|L|)$. `Algorithm 3` uses no sets or complex structures and has a constant space complexity $\theta(1)$.

Finally, `Algorithm 4` uses two sets: L, containing the locations on the path of κ_1 and C, containing the covered locations on the path. `Algorithm 4` has a space complexity $\mathrm{O}(|L| + |C|)$.

Discussion. Our deterministic registration approach can be regarded as a search problem since we are looking for intersections between the paths of registered Things and the path of the new Thing κ_1. In traditional search solutions, a preprocessing step is preformed to optimize the search process (e.g., sorting data prior to executing the search requests). In our algorithm, the data to sort is the locations of registered Things at the time at which a new Thing joins the network, which in the traditional case, would have a time complexity $\mathrm{O}(\log(|R|))$. However, Things are mobile and we do not know, a priori, the time at which a new Thing will join the network.

Therefore we cannot perform the sorting before it actually joins, especially since time is a continuous variable and we cannot sort the locations of all registered Things for all possible time values. As such, we can not reuse preprocessed results from one Thing for the other in the general case.

The approach can also be regarded as a *continuous range query problem* on *Mobile Object Databases* common in the Geographical Information Systems (GIS) [Star and Estes, 1990] domain or the mobile navigation domain. There are two categories of range computations: based on Euclidean distance, or based on road network distance. We are interested in the former. Mobile Object Databases (MOD), introduced in the late nineties by several parallel research efforts ([Prasad Sistla et al., 1997, Erwig et al., 1999]) can support information of the form: *(location, time)*, corresponding to past movements or future movements provided by a path planning tool [Lema et al., 2003, Ding et al., 2008, Grumbach et al., 2001]. They can also handle spatial information [Pelekis et al., 2004]. Although those characteristics match our requirements and can be integrated in the Registry, the MOD performance, with timeliness being an important requirement, is proportional to the load they are subjected to. The load, in our current context, is defined in terms of the number of concurrent queries to compute the geometric overlaps of an ultra large number of mobile Things. Moreover, even though MODs are designated to perform well in a similar context to ours, the computation time will certainly grow with the dataset size.

In conclusion, the deterministic registration approach checks the locations of all similar or expansion services to determine if the service provided by κ_1 is needed. As can be deduced from the complexity analysis it can grow linearly with the size of R, which in our context, can be very large, in the order of billions. This problem can be addressed by adopting a probabilistic mobility model to separate the coverage computations from the actual mobility details and the size of the set of registered services, i.e., separate the registration computation process from the Registry and its backend database. The mobile model-based approach is presented in the following section.

4.2.2 Probabilistic Thing-based Registration

As presented above, the deterministic registration approach provides a relatively precise decision but its time complexity increases linearly with the number of registered

Things. Given that we anticipate, in the mobile IoT, the availability of a large number of Things (millions or more), the approach can be very costly. We propose a probabilistic optimization that builds on the TLW mobility model to estimate the movements of mobile Things. The resulting knowledge can be exploited by the incoming Thing κ_1 to make an approximated decision whether or not to register its services. The decision is based on the probability that, for each location l_i, at least one registered service with similar capabilities is hosted on a mobile Thing that will cross paths with κ_1 at l_i at time t_i.

4.2.2.1 Probabilistic Registration Approach Based on Direct Coverage

The detailed itinerary information provided by each Thing is now substituted by estimations of the possible displacements of Things hosting registered services. The estimations are provided by exploiting TLW. They are computed locally and independently on each new Thing. As such, each Thing can determine, locally, whether or not to register its services. Performing computations locally on each Thing decreases the computation load on the Registry. We formulate the problem to solve as follows:

> *Given a new Thing κ_1 wishing to register a hosted service abstracting a sensor/actuator of type τ, in an environment with an unknown dynamic topology, consisting of a set R of services hosted on location-aware mobile Things and abstracting different types of sensors/actuators, compute the probability $P_{covered}$ that the locations on the path to be followed by κ_1 (with a given itinerary) can be covered by Things hosting registered services in R and determine if κ_1 should register its service, based on a required coverage threshold.*

To be able to compute the registration decision without any knowledge of the actual locations of mobile Things hosting registered services, the Registry is required to provide the number of already registered similar and expansion services along with the distribution of their hosts in space. It is possible to estimate the latter along with its parameters using `ALLFITDIST` method provided by MATLAB.[1] `ALLFISTDIST` method takes a sample of points as input and returns a list of fitting distributions with their corresponding parameters, sorted by Negative of the Log Likelihood (NLogL),

[1]MATLAB: http://www.matlab.com

Bayesian Information Criterion (BIC), or Akaike Information Criterion (AIC), which are common metrics used to compare *goodness of fit*. Goodness of fit describes how well a statistical model fits a set of data values. In this case, the sample points are a set of X-coordinates and a set of Y-coordinates representing the locations of Things hosting the registered services. The output of the method is their distribution in space. The process can be performed periodically or whenever a new Thing registers a service.

As mentioned earlier, we need to compute the probability that the path of the new Thing κ_1 is covered. For this purpose, the approach should compute the probability that at least one Thing will be at each of the locations of interest at the same time as κ_1. Locations of interest should be at most $2 * r_\tau$ away from each other so as not to have uncovered holes in the path. Let $P(\geq 1 \text{ Thing is at}(l_i, t_i))$ be the probability we are looking for at each location.

$$P(\geq 1 \text{ Thing is at } (l_i, t_i)) = 1 - P(\text{no Thing at } (l_i, t_i)) \tag{4.6}$$

Note that we assume that the probabilities of distinct Things being at a location are independent. Consequently, $P(\text{no Thing at } (l_i, t_i))$ is as follows:

$$P(\text{no Thing at } (l_i, t_i)) = \prod_{\kappa \in R} (1 - P(\kappa \text{ is at } (l_i, t_i))) \tag{4.7}$$

The probability that κ will be at location l_i at time t_i is the probability of κ moving from its initial location to l_i. This refers to the total displacement δ_i^κ of Thing κ from its initial location l_0^κ at time t_0 until time t_i. The probability to compute becomes:

$$P(\geq 1 \text{ Thing is at } (l_i, t_i)) = 1 - \prod_{\kappa \in R} (1 - (P(\delta_i^\kappa = l_i - l_0^\kappa))) \tag{4.8}$$

We build on the work by [Sadiq and Kumar, 2011] where the probability function of a Thing being at a specific distance from the origin, i.e., the probability of a Thing having a specific displacement, is provided. The Things are assumed to follow a TLW. In this work, authors show that the *Central Limit Theorem* (CLT) applies to the displacements in TLW. CLT is a theorem that states that the distribution of a sample can be approximated to a normal distribution if some conditions apply as follows:

- Each mobile Thing selects a step length from the same distribution repeatedly, which leads to its position being the sum of the identically independently distributed steps, which is the first condition to satisfy.
- Mean and variance points are finite, which is the second condition to satisfy. They are finite because the TLW has truncation points for the flight and pause time distributions.

Consequently, and given that the displacement is in two dimensional cartesian coordinates, authors approximate the displacement probability function to a bivariate normal distribution of the form:

$$\phi(X, Y, t) = \frac{1}{\sqrt{2\pi Dt}} * e^{\frac{-X^2}{2Dt}} * \frac{1}{\sqrt{2\pi Dt}} * e^{\frac{-Y^2}{2Dt}} \tag{4.9}$$

D is the diffusion factor in the TLW and it is equal to $\frac{\sigma_\xi^2}{\mu_t}$ [Rhee et al., 2008], where σ_ξ^2 is the variance of the displacement length and μ_t is the mean of the complete displacement time distribution. Note that σ_ξ^2 and μ_t are parameters that depend on the real life scenario and the mobility category.

Equation 4.9 is applicable if the displacement starts from the origin $(0, 0)$. However, in our case it starts from location $l_\kappa^0 = (X_\kappa^0, Y_\kappa^0)$. Further, if we try to compute $P(\delta_i^\kappa = l_i - l_k^0)$, i.e. compute the probability of Thing κ being at an exact location, the outcome will be 0. Therefore, we define an area over which the probability should be estimated. We consider that the location should be approximated to the area of the circle C_{l_i,r_τ}, with radius equal to the coverage range r_τ of the Thing and centered at l_i.

Consequently, the probability that κ is in C_{l_i,r_τ} at time t_i starting from a known location $l_0 = (X_0^\kappa, Y_0^\kappa)$ becomes:

$$P(\kappa \text{ is in } C_{l_i,r_\tau} \text{ at time } t_i) = \frac{1}{2\pi Dt_i} \oint_{l^\kappa \in C_{l_i,r_\tau}} e^{-\frac{(X^\kappa - X_0^\kappa)^2 + (Y^\kappa - Y_0^\kappa)^2}{2Dt_i}} dl^\kappa \tag{4.10}$$

where $l^\kappa = (X^\kappa, Y^\kappa)$.

Since the locations of already registered Things are not assumed to be known, the initial location l_0^κ for Thing κ at time t_0 can be anywhere in the deployment area A. Going a step further, we can say that with respect to location l_i, we only care for

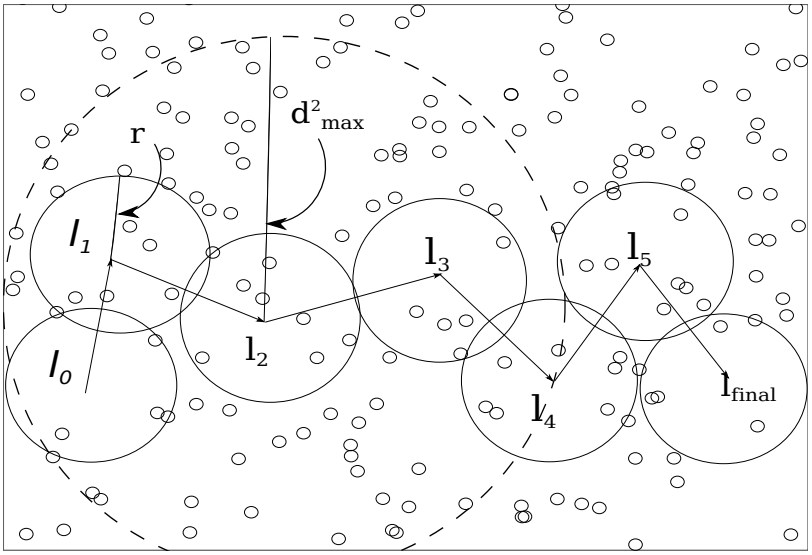

Figure 4.2. An illustration of the maximum reachability area by $C_{l_i,d^2_{max}}$ and the coverage areas around locations l_i.

Things located within the circle $C_{l_i,d^i_{max}}$, with center l_i and radius d^i_{max} (as shown in Figure 4.2); d^i_{max} is the distance from l_i beyond which no Thing κ can reach location l_i at time t_i. Taking into account all locations within circle $C_{l_i,d^i_{max}}$, the probability of a Thing κ being in $C_{l_i,d^i_{max}}$ is:

$$P(\kappa \in C_{l_i,d^i_{max}}) = (\sum_{\text{all x,y in } C_{l_i,d^i_{max}}} P(\kappa \text{ starts at (x,y)}) \tag{4.11}$$

$$P(\kappa \in C_{l_i,d^i_{max}}) = \oint_{C_{l_i,d^i_{max}}} PDF(X^0_\kappa, Y^0_\kappa) dX^0_\kappa dY^0_\kappa \tag{4.12}$$

with $PDF(X^0_\kappa, Y^0_\kappa)$ being the spatial distribution of the locations of Things specified by the Registry. We assume that the location of the Thing on the X-axis is independent from its location on the Y-axis, therefore:

$$P(\kappa \in C_{l_i,d^i_{max}}) = \oint_{C_{l_i,d^i_{max}}} PDF(X^0_\kappa) * PDF(Y^0_\kappa) dX^0_\kappa dY^0_\kappa \tag{4.13}$$

Consequently, the probability of Thing κ moving to C_{l_i,r_τ} can be computed as follows:

$$g(l_i,t_i,r) = \oint_{l_0^\kappa \in C_{l_i,d^i_{max}}} PDF_{X_0^\kappa}(X^\kappa) * PDF_{Y_0^\kappa}(Y^\kappa) dl_0^\kappa * \frac{1}{2\pi D t_i} \oint_{l^\kappa \in C_{l_i,r_\tau}} (e^{-\frac{(X^\kappa - X_0^\kappa)^2 + (Y^\kappa - Y_0^\kappa)^2}{2Dt_i}} dl^\kappa) \tag{4.14}$$

If we go back to the probability in Equation 4.8, building on the assumption that Things move independently from one another, we obtain by substitution:

$$P(\geq 1 \text{ Thing at } (l_i,t_i)) = 1 - \prod_{\kappa \in n_i} (1 - g(l_i,r_\tau,t_i)) \tag{4.15}$$

where n_i is the set containing all Things in $C_{l_i,d^i_{max}}$. The assumption that Things move independently stems from the fact that we consider the displacements of individuals separately without accounting for group mobility or the correlation of displacements between various individuals, which is left for our future work. The probability of displacement from l_0^κ into C_{l_i,r_τ} is identical for all Things in n_i, therefore $P(\geq 1$ device at $(l_i,t_i))$ becomes:

$$P(\geq 1 \text{ Thing at } (l_i,t_i)) = 1 - ((1 - g(l_i,r_\tau,t_i))^{|n_i|}) \tag{4.16}$$

Now that we have determined the probability of coverage at one location, we can repeat the process to obtain the probability of coverage at all locations.

$$P_{cov} = \prod_{l_i \in L} (1 - (1 - g(l_i,r_\tau,t_i))^{|n_i|}) \tag{4.17}$$

4.2.2.2 Probabilistic Registration Based On Expansion

If any location of interest on the path of the mobile Thing κ_1 is not covered by Things hosting sensors/actuators of type τ, κ_1 can check if the type of the service it hosts can be substituted by other types of services. It is important to ensure that expansion services are hosted on Things present at the same location at the same time. For instance, to estimate the wind-chill value, we need to compute the probability that all locations l_i on the path of κ_1 will be covered by both sensor types τ_1 (thermometer) and τ_2 (anemometer), i.e., sensor types that measure concepts $\in e_\tau^j$. In the following

we show how we can compute the coverage probability that takes expansion into account.

Let $P_{cov}^{S_\tau}$ be the probability of coverage by a mobile Thing hosting a sensor/actuator of type τ, and $P_{cov}^{e_\tau^j}$ be the probability of coverage by Things hosting sensors/actuators corresponding to concepts in one of the expansion sets e_τ^j:

$$\begin{aligned} P_{cov}^{e_\tau^j} &= P_{cov}(\text{ by } S_{\tau_1} \textbf{ and } S_{\tau_2}) \\ &= P_{cov}^{S_{\tau_1}} * P_{cov}^{S_{\tau_2}} \end{aligned}$$

In the general case:

$$P_{cov}^{e_\tau^j} = \prod_{S \in e_\tau^j} P_{cov}^{S} \tag{4.18}$$

P_{cov}^{S} is computed using the same equation as P_{cov} (Equation 4.17), with a change in the type of the sensor/actuator. The complete probability of coverage, i.e., the probability including both direct coverage and coverage with expansion cases, is:

$$P_{covered} = P_{cov}(\text{ by } S_\tau \textbf{ or } e_\tau^j) \tag{4.19}$$

$P_{covered}$ is also equal to 1− (the probability of having no coverage by Things hosting sensors/actuators of type τ and the probability of having no coverage by Things hosting sensors/actuators that monitor/act on concepts in e_τ^j), i.e.,:

$$P_{covered} = 1 - ((1 - P_{cov}^{S_\tau}) * (1 - P_{cov}^{e_\tau^j})) \tag{4.20}$$

In a more general form, assuming there is a set E_τ of possible expansion sets, e_τ^j, the complete probability of coverage is as below:

$$P_{covered} = 1 - ((1 - P_{cov}^{S_\tau}) * \prod_{e_\tau^j \in E_\tau} (1 - P_{cov}^{e_\tau^j})) \tag{4.21}$$

`Algorithm 5` summarizes our approach to generate the final registration decision based on the computed $P_{covered}$. The algorithm first computes the direct coverage using Equation 4.17 (Line 1). If the coverage is less than the threshold (Line 2), it computes the probability of coverage by expansion for each type in set e_τ using

Algorithm 5 Probabilistic Registration decision process

Require: $L, r_\tau, E_\tau, threshold$
Ensure: $decision \in \{true, false\}$
1: *compute* $P_{covered}$ using Equation 4.17
2: **if** $(P_{covered}) \leq threshold$ **then**
3: **for** *each* $e_\tau^j \in E_\tau$ **do**
4: **for** *each type* $\in e_\tau^j$ **do**
5: *compute* P_{cov} using Equation 4.17
6: **end for**
7: compute $P_{cov}^{e_\tau^j}$ using Equation 4.18
8: **end for**
9: $p \leftarrow$ *compute* $P_{covered}$ using Equation 4.21
10: **end if**
{`NT is the new threshold generated after computing the coverage probability`}
11: $NT \leftarrow 1 - \frac{(1-threshold)}{(1-p)}$
12: **if** $uniformRand(0,1) < NT$ **then**
13: $decision \leftarrow true$
14: **else**
15: $decision \leftarrow false$
16: **end if**
17: **return** *decision*

Equation 4.17 (Line 3-5), and then for the whole set e_τ using Equation 4.18 (Line 7). Finally the complete probability of coverage is computed using Equation 4.21 (Line 9). The result is then used to generate a new threshold through a utility function presented below. The new threshold is then compared to random uniform number. If the number is below the new threshold, the Thing registers, and vice versa (Line 12-15). The randomized approach and the new threshold generation are chosen to introduce a margin for flexibility in the registration decision because the coverage computations build on estimations of potential coverage and not exact information. The end goal is to minimize the probability that the Thing registers, while still achieving an adequate amount of coverage. The new threshold, which determines the final probability of registration, is generated as presented below, provided that:

- $C = event\ that\ path\ is\ covered$
- $prob(C) = probability\ that\ path\ is\ covered,\ i.e.,\ prob(C) =\ threshold\ th$

- $C_T = event\ that\ path\ is\ covered\ by\ other\ Things$
- $prob(C_T) =\ probability\ that\ path\ is\ covered\ by\ other\ Things = p$
- $Accepted\ Loss = 1 - th$
- $Register = event\ that\ \kappa_1\ registers$
- $prob(Register) = probability\ that\ \kappa_1\ registers, i.e.,\ prob(Register) = new\ threshold\ NT$

The new threshold generation starts with the fact that the accepted loss is equivalent to the path not being covered and the new Thing not registering. Therefore:

$$Accepted\ Loss = prob(\overline{Register} \cap \overline{P}) \tag{4.22}$$

$$1 - th = (1 - prob(Register)) * (1 - p) \tag{4.23}$$

By simplifying the equation we obtain:

$$1 - prob(Register) = \frac{1 - th}{1 - p} \tag{4.24}$$

Finally, the new threshold, equivalent to $prob(Register)$, is:

$$NT = 1 - \frac{1 - th}{1 - p} \tag{4.25}$$

Although we assumed independence between the *Register* and P events, our evaluations in Chapter 6 show that our assumption does not lead to low quality results. Relaxing the assumptions and investigating the correlations between both events is part of our future work.

It is important to clarify that in our current approach, we assume that the time needed to perform a sensing/actuating task is equal to the time needed for a mobile Thing to cross a location l_i. Relaxing this assumption and varying the required availability duration according to the physical concept itself makes part of our future work.

Complexity analysis. `Algorithm 5` terminates either if direct coverage is sufficient or when the end of the outermost loop is reached (Line 3 - 9). The loop depends on the size $|E_\tau|$, which is a finite number. There is one inner loop that depends on the size $|e_\tau^j|$, which is the number of expansion types in each expansion set, also a finite number. Unlike the deterministic registration algorithm, which has to check the actual paths of Things hosting registered services, the current algorithm is not dependent on those values.

`Algorithm 5` depends on the number of locations $|L|$ on κ_1's path, the number of expansion types in e_τ, the number of expansion sets in E_τ, the size of the area $A_{C_{l_i,d^i_{max}}}$ computed for $C_{l_i,d^i_{max}}$ (to determine $|n_i|$, the number of Things that can reach l_i in time t_i), and the size of the area $A_{C_{l_i,r_\tau}}$ computed for C_{l_i,r_τ} (to determine the coverage at location l_i). The time complexity of the algorithm is therefore$\mathrm{O}\left(|E_\tau||e_\tau^j||L||A_{C_{l_i,d^i_{max}}}||A_{C_{l_i,r_\tau}}|\right)$. Note that steps 1, 5, and 6 perform operations, namely integrations, over the areas $A_{C_{l_i,r_\tau}}$, and $A_{C_{l_i,d^i_{max}}}$. This incurs additional time and space costs. However, the operations depend on the parameters of the numerical methods employed and are independent of the problem size.

`Algorithm 5` stores values in two sets: E_τ and e_τ^j. The sets are used when computing Equation 4.17 (for coverage with expansion). Thus, the algorithm has a space complexity $\mathrm{O}\left(|E_\tau| + |e_\tau^j|\right)$.

Finally, given the distributed nature of the algorithm where computations are performed locally on each Thing, additional traffic is generated as the new Thing, performing computations, communicates with the Registry to acquire the spatial distribution and the number $|n|$ of Things hosting services similar to the one provided by the incoming Thing. The message exchange takes place once at the beginning of the registration decision computation. We remind readers that by employing the probabilistic mobility model to estimate the displacement of Things, the Registry is not required to send path information of all registered Things but only their distribution in space. Consequently, the complexity of the communication depends only on the number of bits representing the size of the set n, and the number of bits representing the distribution of the host Things, which are $log(|n|)$ and $log(d)$ respectively, with d representing the spatial distribution and its parameters. As such, the additional communication complexity incurred by distributing computations is $\mathrm{O}\left(log(|n|) + log(d)\right)$.

Clearly, with the computation and communication complexities presented above,

the probabilistic Thing-based registration outperforms its deterministic counterpart, as it requires computations that are independent of the number of registered services. Consequently, the probabilistic approach is more certain to scale. This is also demonstrated in Chapter 6 where we evaluate both deterministic and probabilistic approaches and show their benefits. Additionally, both approaches are designed with mobility as the rule rather than the exception and therefore can handle the dynamic nature of the mobile IoT.

4.2.3 Computation Simplifications

For each location of interest on the path of a mobile Thing κ_1, it suffices to consider $|n_i|$ Things, with n_i being the set of Things that can actually reach l_i at time t_i. To compute the size of n_i, we apply the following steps:

1. For each location l_i, compute $d^i_{max} = v * (t_i - t_0)$.

2. For each location l_i, compute the following values: $x_{min} = x_i - d^i_{max}$ and $x_{max} = x_i + d^i_{max}$ for X_κ and $y_{min} = y_i - d^i_{max}$ and $y_{max} = y_i + d^i_{max}$ for Y_κ. Those values will provide us with boundaries (for squares) that can be used to determine the area around each location l_i beyond which l_i is not accessible in time t_i.

3. The Registry computes the *spatial distributions* and returns $PDF(x)$ and $PDF(y)$ used by Equation 4.14. The locations of Things hosting registered services are sent as sets of X-coordinates and Y-coordinates to MATLAB's `ALLFITDIST` method that returns a list of distributions with errors. The Registry then selects the distribution with the smallest error.

It is shown in [Rhee et al., 2011], that human mobility is scale free. Consequently, looking at the mobility of humans from any scale leads to similar observations and patterns with invariant characteristics. Based on this fact, we consider that the distribution provided by the Registry for a larger region stands correct for the small area within $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$.

Using the new computed area and the spatial distribution of Things, we compute the probability that $l_i = (X_i, Y_i)$ is within the new area limits, i.e. $P(X_i \in [x_{min}, x_{max}])$ and $P(Y_i \in [y_{min}, y_{max}])$. Afterwards, we use the resulting probability value to compute the number of Things expected to be within the defined boundaries

based on the following product:

$$|n_i| = P(X_i \in [x_{min}, x_{max}]) * P(Y_i \in [y_{min}, y_{max}]) * |n|$$

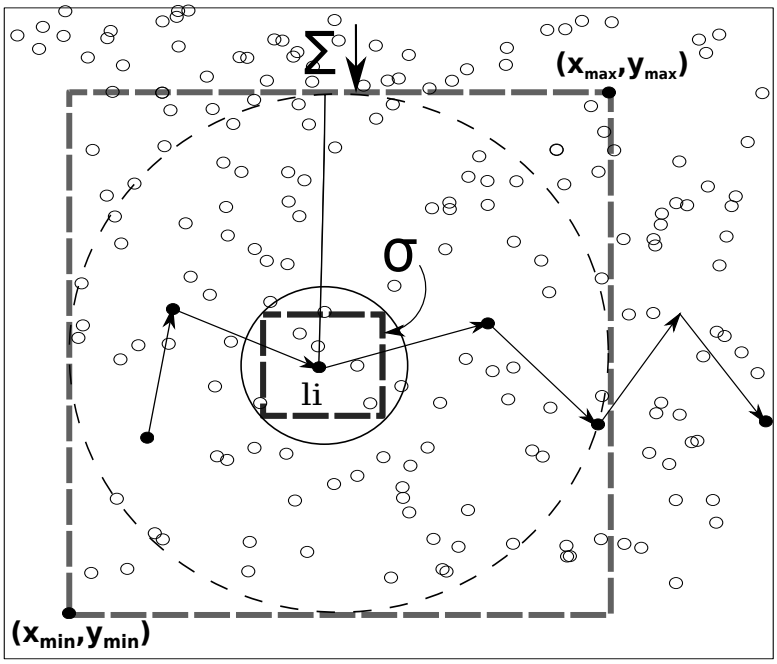

Figure 4.3. A closer look at the substitution of $C_{l_i,r}$ by $\sigma_{l_i,r}$ and $C_{l_i,d^i_{max}}$ by $\Sigma_{l_i,d^i_{max}}$.

To simplify the complex integration computations, we replace the circle C_{l_i,r_τ} by the largest square σ_{l_i,r_τ} within C_{l_i,r_τ} (in order to be pessimistic about our computations). This allows us to split the area integrals into double integrals that have a closed form expression. Similarly, we let $\Sigma_{l_i,d^i_{max}}$ be the smallest square outside the circle $C_{l_i,d^i_{max}}$ (Figure 4.3) to limit the integral bounds. The above simplifications

reduce P_{cov} equation to:

$$P_{cov} \simeq \prod_{l_i \in L}(1 - \prod_{\kappa \in n_i}(1 - \oint_{l^0_\kappa \in \sigma_{l_i, r_\tau}} PDF(X^0_\kappa) * PDF(Y^0_\kappa) dX^0_\kappa dY^0_\kappa * \frac{1}{2\pi D t_i} \oint_{l_\kappa \in \Sigma_{l_i, d^i_{max}}} e^{-\frac{(X_\kappa - X^0_\kappa)^2 + (Y_\kappa - Y^0_\kappa)^2}{2Dt_i}} dX_\kappa dY_\kappa))$$

We can separate the X-axis integrals from the Y-axis integrals, and the equation becomes:

$$P_{cov} \simeq \prod_{l_i \in L}(1 - \prod_{\kappa \in n_i}(1 - \frac{1}{2\pi D t_i} * \Phi(X_i, X_\kappa, X^0_\kappa) * \Phi(Y_i, Y_\kappa, Y^0_\kappa))) \tag{4.26}$$

where

$$\Phi(a, b, c) = \int_{a - d^i_{max}}^{a + d^i_{max}} \int_{a - \frac{\sqrt{2} * r_\tau}{2}}^{a + \frac{\sqrt{2} * r_\tau}{2}} PDF(c) * e^{-\frac{(b-c)^2}{2Dt_i}} dbdc$$

The same simplification logic applies to coverage by expansion, since it also builds on Equation 4.17.

Support for mobility models. We presented above a mathematical solution that is specific to TLW. However, mobility models may evolve and new models can become available. Therefore, it is important to allow the new models to be exploited. To that end, our design permits models to be plugged in easily, as long as they provide a formula to estimate the probability of a mobile Thing being at location l_i at time t_i. Precisely, the model to plug in should provide an equation that computes the same information as Equation 4.10. The new equation will be used to reconstruct Equation 4.26 following the same computation steps we presented throughout this section.

4.3 Thing-based Service Look-up

If the traditional Service-Oriented look-up is to be adopted, all registered services that match a request (i.e., provide the required functionality and satisfy additional constraints, if any are specified) are retrieved. Such an approach leads to the selection of a large number of services located closely to each other and providing redundant functionalities. Even though the Thing-based registration limits the participation of Things, they remain numerous. We address this issue at the look-up phase by selecting, probabilistically, a subset of registered services based on an estimation of the location of their hosts and the coverage they provide. Unlike the Thing-based registration where coverage is computed per Thing, along its own path, coverage in the look-up phase is computed for the whole area of interest specified in the user query.

4.3.1 Probabilistic Thing-based Lookup

As sensing/actuating tasks are tightly bound to a real world geographical setting, location information should be accounted for when selecting the subset of Things. However, selecting Things based on specific coordinate points in the area of interest is a good approach if Things are static and deployed to cover those locations, as they will most likely be available at those locations. Introducing mobility complicates the matter as chances of their unavailability are high. However, unlike registration, using mobility models in this case is not beneficial because concrete location information is more crucial when answering location-based queries. Yet, asking Things for their precise locations, whenever a request to find services is received, incurs high communication costs, due to the recurring message exchanges with a very large number of Things. A solution is to select the subset of Things based on a distribution in space rather than exact coordinates.

We formulate the problem to solve as follows:

> *Given a set R_τ of services abstracting sensors/actuators of type τ, select a subset $s \subseteq R_\tau$ of those services hosted on mobile Things from an area a based on a distribution in space $DIST$ of the mobile Things, and a required subset size $|s|$.*

We start by giving an overview of the approach and proceed with an elaborate explanation of the corresponding algorithm. To be able to select the correct subset, it is important to locate the registered Things at the time of the request t_r. Whenever a Thing registers its services it sends an estimation of the path it will follow as a set of (x,y) coordinates and timestamps. It is possible that none of the timestamps matches t_r, therefore, an estimation of the locations of Things hosting services in R_τ at time t_r is required. Afterwards, in accordance with the commonly adopted approach in the literature [Xu et al., 2001, Ye et al., 2002], the area a is divided into a grid Q consisting of $|Q|$ squares. In order to not bias the selection, the same number of Things is selected from each square of the grid. When services are selected, their access addresses are returned as the result. The address is a URL that determines how the service can be contacted to provide its functionality. The details of the solution are presented in `Algorithm 6`.

`Algorithm 6` first calls the `interpolation()` method to estimate the location of appropriate mobile Things at time t_r (Line 7). The `interpolation()` method is presented in `Algorithm 7`. Afterwards, the algorithm constructs the grid Q over area a using the `constructGrid()` method (Line 8). The details of the grid construction depend on the spatial distribution and are presented in the next section. The algorithm then determines which of the Things are in each square q of Q using the `identifySquares()` method (Line 9). The method determines which Things belong to which of the squares in the grid by comparing the estimated coordinates of the Things to the border coordinates of each square. Afterwards, the algorithm selects an equal number $\frac{|s|}{|Q|}$ of Things from each square (Line 10-15) and returns the subset of selected services.

`Algorithms 7` takes the set of registered services in R_τ and the request time as input. It returns the same set of services with interpolated locations at time t_r as output. The algorithm exploits a binary search approach to determine the timestamps t_j and t_{j+1} that are directly before and after the request time t_r such that $t_j < t_r < t_{j+1}$. It then finds the locations (x_{t_j}, y_{t_j}) and $(x_{t_{j+1}}, y_{t_{j+1}})$ of the mobile Things at times t_j and t_{j+1} respectively, and estimates (x_t, y_t) for each Thing. In more detail, for each service, it performs the binary search to find the index of the largest timestamp t_j in the path of the Thing hosting the service, that is smaller than t_j (Line 4). It then finds the host's location at the returned index (i.e., at time t_j) (Line 5). If the path of the current service does not have a timestamp that matches

Algorithm 6 Probabilistic Lookup

Require: $R_\tau, a, t_r, |s|$
Ensure: s
{I is the set of services with their interpolated locations at the time of request t_r}
1: *let* $I \leftarrow \emptyset$
{Q is the grid constructed over the area of interest. It consists of several equal squares}
2: *let* $Q \leftarrow \emptyset$
{servicesInGrid contains services and the square they belong to.}
3: *let* $servicesInGrid \leftarrow \emptyset$
4: *let* $tempArray \leftarrow \emptyset$
5: *let* $tempServices \leftarrow \emptyset$
6: $counter \leftarrow 0$
7: $I \leftarrow interpolation(R_\tau)$
8: $Q \leftarrow getGrid(a)$ {construct the grid of squares}
{determine services belong to which squares of the grid by comparing their coordinates to the border coordinates of each square}
9: $servicesInGrid \leftarrow identifySquares(I, Q)$
10: **for** *each square* $q_i \in Q$ **do**
11: $tempServices \leftarrow q_i.getServices$ {return all services in s at time t_r}
12: **while** $counter < \frac{|s|}{|Q|}$ **do**
13: $index \leftarrow math.random * tempServices.size$
{select uniform random locations of mobile Things}
14: **if** $s.contains(tempServices.get(index)) = false$ **then**
15: $s.add(tempServices.get(index))$
16: $counter++$
17: **end if**
18: **if** counter = tempServices.size **then**
19: break
20: **end if**
21: **end while**
22: **end for**
23: **return** s

the requirement, i.e., the estimated path of the current service's host starts after t_r, the algorithm skips the service and moves to the next (Line 6-7). Otherwise, it checks if $t_j = t_r$, in which case the location l_j is returned as the answer (Line 9-10).

Algorithm 7 `interpolation` method

Require: R_τ, t_r
{`return the set of services that satisfy the request with their estimated locations`}
Ensure: *interpolatedResult*
1: *let interpolatedResult* $\leftarrow \emptyset$
2: *index* $\leftarrow 0$
3: **for** *each service* $s \in R_\tau$ **do**
4: $index = lowerBinarySearch(s.locations, 0, locations.size() - 1, t)$
5: $selectedLocation = s.locations.get(index)$
6: **if** $index < 0$ **then**
7: *continue*
8: **else**
9: **if** selectedLocation.time = t **then**
10: $locationNow = selectedLocation$
11: **end if**
12: **else**
13: **if** index = locations.size() - 1 **then**
14: *continue* {move to the next service.}
15: **end if**
16: **else**
17: $locationNow = interpolate(selectedLocation, locations.get(index + 1), t_r)$
18: **end if**
19: $s.location \rightarrow locationNow$
20: $interpolatedResult.add(s)$
21: **end for**
22: **return** *interpolatedResult*

The algorithm also checks if the returned index corresponds to the last location on the host's path, in which case the service is ignored (Line 13-14). If none of the issues above is true, the algorithm finds the host's location at the $index + 1$ (at time t_{j+1}) and passes it along with the location l_j and time t_r to the interpolate method presented in `Algorithm 3` to estimate the host's location at time t_r (Line 16-17). Once all service have been checked, a list of services with interpolated locations at time t_r is returned (Line 22).

We present in the following, the grid construction approach based on two distributions: uniform distribution and normal distribution.

4.3.1.1 Distribution-based Grid Construction

The first step towards the construction of the Grid Q over the area of interest a, is to identify the spatial distribution of Things, hosting services to interrogate, along with its parameters. The spatial distribution type depends solely on the type of the event to measure/modify.

After surveying the literature, we concluded that events to monitor/modify can belong to one of the following categories:

- *Boolean events*: A boolean event does not spread. If it is not directly within the coverage range of a sensor/actuator it can neither be detected, nor acted on (e.g., pressure sensor).
- *Uniformly spreading events*: A uniformly spreading event in an area a is an event that has the same state across the whole area.
- *Decaying events*: A decaying event is an event that spreads over some distance with its amplitude decaying based on a decay function f.

Uniformly spreading events within an area a can be measured/acted on from any location within a. It is very likely that they will have the same value given their uniform nature. In this case, we select the Things to measure/act on based on a uniform spatial distribution, where every Thing has a probability $\frac{1}{|R_\tau|}$ of being selected. The uniform distribution is commonly adopted for sensor selection, based on a required area coverage (e.g., [Wang, 2011, Choi and Das, 2009]).

Decaying events have an inversely proportional amplitude to the distance from the location l_e of the event. Consequently, to estimate the state of the event at location l_e, where it takes place, it is more beneficial to select more Things around l_e and fewer as the distance increases. This setting corresponds, among others, to the outcome of a normal distribution based sampling. The normal distribution, characterized by the Gaussian bell curve shown in Figure 4.4, has the following characteristics:

- 68.2% of the values are within one variance from the mean.
- 27.2% of the values are between the first and second variance from the mean.
- 4.2% of the values are between the second and third variance from the mean.

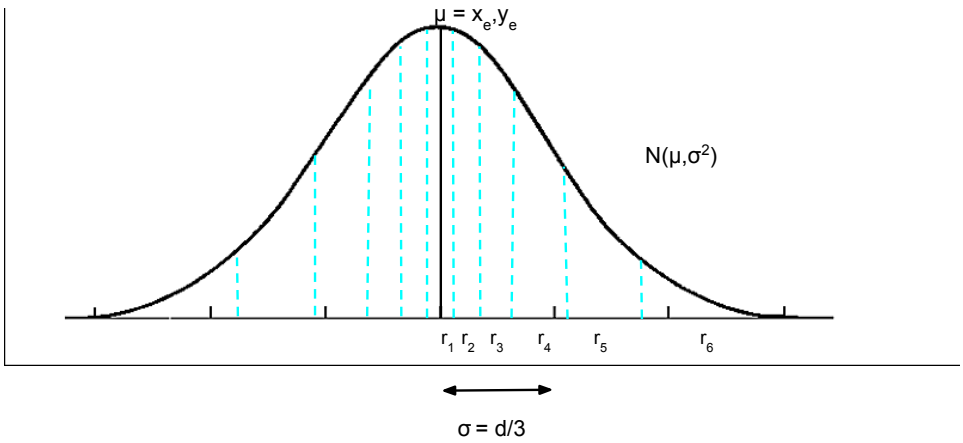

Figure 4.4. The PDF of a normal distribution with $\sigma = \frac{d_u}{3}$. The area under the curve is divided into 12 equal areas.

An important characteristic of normal distribution, is that there is almost no point beyond 3 variances. Specifically, 99% of values are within 3σ from the mean μ, with σ^2 being the variance of the distribution. This matches a peculiarity of decaying events, where their amplitude can no longer be detected beyond a distance d_u. Consequently, the parameters of the normal distribution are specified by mapping the normal distribution density function to the real world setting where $3\sigma = d_u$, with the mean being the location of the event of interest l_e, such that $l_e : (x_e, y_e) = \mu$. The parameters of the normal distribution are therefor $\mu = (x_e, y_e)$ and $\sigma^2 = (\frac{d_u}{3})^2$.

Contrary to uniformly spreading events, selecting Things in a from beyond d_u is guaranteed to incur additional costs with no additional benefit. Therefore it becomes crucial to compute the value of d_u depending of the nature of the event itself. Decaying events abide to well defined laws of physics and have well specified formulas that model their decay and their decay distance. Concept-specific decay functions and decay-distance functions are defined in the domain ontology. Using the decay-distance functions, d_u can be computed and the area from which Things should be selected can be determined as follows:

1. $x_{max} = x_e + d_u$
2. $y_{max} = y_e + d_u$

3. $x_{min} = x_e - d_u$

4. $y_{min} = y_e - d_u$

For instance, the *acoustic attenuation* function is:

$$A_d = A_0 * e^{(-\alpha * d)} \tag{4.27}$$

where A_d is the acoustic value at distance d from the origin, A_0 is the acoustic value at the origin and α is the attenuation factor. An acoustic domain expert can extract d from Equation 4.27 by setting, for instance, $A_d = 0.001\%$ of A_0, and obtain $d = -\frac{1}{\alpha}\ln(0.1\%)$, which is the decay-distance function.

To properly select Things based on the spatial distributions above, the grid should be constructed in a manner that inclines the subset selection process towards results that match the expected distributions' outcome. We note that, the normal distribution is not the only solution to be adopted for decaying events. It is possible that some decay formulas match different distributions, such as for instance, an exponential distribution. Our current solution, where normal distributed was chosen as an illustration for its analytical tractability, can be extended to include any other distribution by properly revising the grid construction process presented below.

Uniform distribution. If the distribution of the subset to select should be uniform, the grid computation is straightforward. It consists of dividing a into $|Q|$ equal squares q_i [Xu et al., 2001, Ye et al., 2002]. Afterwards, a size $\lambda = \frac{|s|}{|Q|}$ of Things is uniformly selected from R_{q_i}, the set of registered mobile Things that will be within square q_i at time t_r.

It is possible that a square q_i has less than the required λ at the time of the request t_r. An alternative is to retrieve the remaining mobile Things from the nearest square to q_i. The second alternative is to not replace the missing Things as they would have been missing even if we opt for a full Thing selection. We chose the latter in order to not increase the computation complexity.

Normal distribution. The process is more complex for the normal distribution. The grid to construct should reflect the fact that there are more points around the mean, i.e. the event of interest, and fewer as we move farther. Towards that goal,

instead of dividing the area into equal squares, it is divided into differently sized concentric squares. To compute the size of the squares, a is approximated to the area under the probability density function (PDF) of the normal distribution. The PDF area is divided into smaller equal areas. To have equal areas under a decreasing curve, the width of each area should increase while the heights decrease. Thus, the range r_i of each area will increase as illustrated in Figure 4.4 (the desired area is $1/12th$ of the total area). To compute the different range values r_i, we use the *inverse of the Cumulative Distribution Function* (CDF) also known as *quantile function*. A quantile function returns, for a given probability, the value at which a random variable will be. The probability corresponds to the area between the mean and the value of the random variable on the X-axis. In the real world setting, this value also corresponds to the width of ith square q_i. The quantile function is defined in terms of the cumulative distribution function as:

$$x = F^{-1}(p|\mu, \sigma) = [x : F(x|\mu, \sigma) = p] \tag{4.28}$$

where

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \tag{4.29}$$

As mentioned earlier, the grid is constructed as several concentric squares around the event of interest. Each square has a width equal to one of the different range values r_i computed by the quantile function. A example of a resulting grid is illustrated in Figure 4.5. The squares closer to the event of interest are smaller and have a higher density of Things since we select the same number $\lambda = \frac{|s|}{|Q|}$ of Things, uniformly, from the set $R_{q_i} \setminus R_{q_{i-1}}$, for each square q_i. Things are selected from this set as we should only consider the region that is not covered by the inner square q_{i-1}. The resulting regions to select from are illustrated by the differently shaded regions in Figure 4.5.

In the approaches presented above, it is assumed that the area a is either specified by the user and provided as input or computed dynamically by MobIoT. However, this imposes some limitations in the case where neither the user nor MobIoT can properly compute that area, which should rather be computed at the application level. Evidently, this fact does not affect MobIoT's performance as it will still receive the area value as input from the application. Nonetheless, it would be highly beneficial if the middleware can assist developers in specifying their application logic to dynamically compute the area, which makes part of our future work.

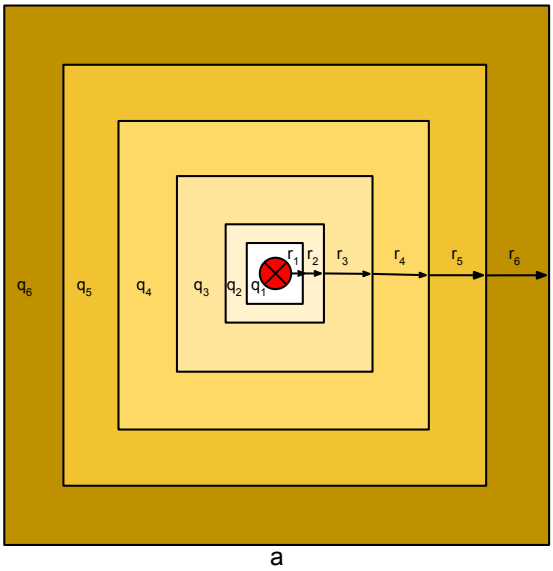

Figure 4.5. The final grid from which $|s|$ normally distributed mobile Things should be selected.

Complexity analysis. `Algorithm 6` terminates either if, for each square q_i in the grid, the density requirement λ is satisfied, or if all services are selected (in the case where the square has less services than the required density). The algorithm depends on `Algorithm 7`, which is sure to terminate as it iterates over a finite number of services in R_τ. It also depends on the `getGrid()` method. The method depends on the area of interest, which has a predefined width and height, therefore the method is sure to terminate. The `identifySquares()` method depends on the number of squares in the constructed grid and the number of services, both of which are finite. Therefore, this method is also sure to terminate. Consequently, `Algorithm 6` is sure to terminate.

The `interpolation` method in `Algorithm 7`, iterates over services in R_τ once, to estimate their locations, and therefore has a time complexity of order $\mathrm{O}(|R_\tau|)$. In the worst case, `Algorithm 7` will interpolate the locations of all registered services in R leading to $|R_\tau| = |R|$. Therefore it has a worst case complexity $\mathrm{O}(|R|)$.

The `getGrid()` method has a time complexity $\theta(\frac{|a|}{ratio})$. The `identifySquares()` method goes through every service in I once and has a time complexity of order $\mathrm{O}(|I| * |Q|)$, as it also iterates over the squares in Q in order to determine which service in I belongs to which square q_i. In the worst case, the method will go through

all services in R, leading to $|I| = |R|$. Consequently, the worst case complexity of the `identifySquares()` method is $\mathrm{O}(|R| + |R| + |R| * |Q|)$. therefore, the complexity becomes $\mathrm{O}(|R| * |Q|)$.

`Algorithm 6` invokes `Algorithm 7` and the internal methods presented above. Additionally, `Algorithm 6` depends on a `while` loop (Lines 12 - 19) over λ where $\lambda = \frac{|s|}{|Q|}$. To ensure that the loop does not iterate indefinitely, we check at every iteration if all services have been selected at the current square, and break the loop if this condition applies. `Algorithm 6` also depends on the number of squares in Q as it iterates over each square to randomly select services (Lines 10 - 19). As such, `Algorithm 6` has a time complexity of order $\mathrm{O}(I * |Q| * |R| * \lambda)$. In the worst case, I and $s$will contain all registered service in R, leading to $|I| = |s| = |R|$, therefore the worst case complexity of `Algorithm 6` is $\mathrm{O}\left(|Q| * \frac{|R|}{|Q|}\right)$.

The space complexity of `Algorithm 7`, which uses two sets: i) R_τ, and ii) the interpolated set $interpolatedResult$ is $\mathrm{O}(|R_\tau| + |interpolatedResult|)$. The algorithm estimates a location for each service in R_τ and adds the result to $|interpolatedResult|$, therefore $|interpolatedResult| = |R_\tau|$. The space complexity reduces to $\mathrm{O}(|R_\tau|)$. In the worst case, R_τ will contain all services in R leading to: $|R_\tau| = |interpolatedResult| = |R|$. Therefore, `Algorithm 7` has a worst case complexity $\mathrm{O}(|R|)$.

The `getGrid()` has a space complexity $\theta(\frac{|a|}{ratio})$. Given that the area should be specified before the algorithm initiates the selection process, the area size is finite.

The `identifySquares()` method uses two sets: i) Q, containing the grid squares, and ii) I, containing the interpolated services. It determines if each service in I belongs to one of the squares $q_i \in Q$. The space complexity of this method is $\mathrm{O}(|I| + |Q|)$ since it does not use any complex data structures or perform any preprocessing. In the worst case, the method has a space complexity $\mathrm{O}(|R| + |Q|)$, if all registered services are in the interpolated set, leading to $|I| = |R|$.

`Algorithm 6` uses six sets: i) R_τ, containing registered services abstracting sensors/actuators of type τ; ii) I, containing services with interpolated locations; iii) Q, the grid constructed over a; iv) $servicesInGrid$, containing services in Q; v) $tempServices$, containing randomly selected services in q_i; and vi) $finalSubset$, containing the addresses of all randomly selected services. Consequently, the space complexity of the algorithm is $O(|R_\tau| + |I| + |Q| + |servicesInGrid| + |tempServices| + |finalSubset|)$. It should be noted that $|I| = |R_\tau|$ since I contains the same number of services as R_τ, with the difference being the new interpolated locations. Therefore, the complexity

becomes $O(|R_\tau| + |I| + |Q| + |servicesInGrid| + |tempServices| + |finalSubset|)$. In the worst case scenario, all services in R will be in R_τ, and all services in R will be in Q at time t_r, leading to $|servicesInGrid| = |R_\tau| = |R|$, all services can be in one square q_i leading to $|tempServices| = |R|$ and all services will be selected to be accessed, leading to $|finalSubSet| = |R|$. The worst case space complexity is $O(|Q| * |R|)$.

It is likely in some cases to not know a priori the best subset size that can provide enough information/actions in an area a. Similar to registration, the actual coverage provided by Things can play an important role in the performance of sensing/actuating tasks. We exploit this information and build a new subset selection approach. The approach determines the size of the subset to select dynamically, based on the coverage the Things hosting the selected services provide over the whole area a. In the case of uniform distribution, [Choi and Das, 2009] provide an expression to compute the size of the subset to select, based on a required coverage probability ψ. The expression is:

$$\psi = 1 - (\frac{a - C_{r_\tau, l_\kappa}}{a})^\lambda \tag{4.30}$$

with C_{r_τ, l_κ} being the coverage circle around a Thing κ to select and λ being the size of the subset to select. Based on Equation 4.30, it is possible to compute λ as follows:

$$\lambda = \frac{log(1 - \psi)}{log(\frac{a - C_{r_\tau, l_t}}{a})} \tag{4.31}$$

The same computations do not apply in the case of normal distribution. To address that issue we provide a lookup approach that determines λ dynamically, based on whether or not the selected subset provides enough coverage.

4.3.2 Coverage-Based Probabilistic Lookup

The coverage-based probabilistic lookup allows MobIoT to select a subset of services with no a priori knowledge of the exact number of services to interrogate. We formulate the problem to solve as follows:

> *Given a set R_τ of services abstracting sensors/actuators of type τ, a coverage requirement c, which is a percentage $\in [0, 100]$ of area a, select a subset $s \subseteq R_\tau$ of those services, hosted on mobile Things in area a that provide the coverage c, based on a distribution in space $DIST$.*

As stated earlier, unlike the registration approach where coverage is computed per Thing along its path alone, the coverage in the look-up case is for the whole subset and over the whole area of interest a. As such, it is important to compute the coverage provided by all Things hosting services in R_τ first, to check whether or not they are all needed to provide their sensing/actuating services. We refer to the computed coverage as Maximum Possible Coverage (MPC). MPC is then compared to the minimum coverage threshold. If the MPC is above c, the subset selection process is initiated.

The process builds on the idea governing many existing solutions, where sensors are selected based on the benefit they introduce [Dhillon and Chakrabarty, 2003, Bijarbooneh et al., 2012, Chamam and Pierre, 2009]. In this case, the benefit is the coverage they provide. However, unlike existing solutions where sensors are static and the selection process is performed offline in a preprocessing step, our sensors/actuators are hosted on mobile Things. Further, the selection process is done online and should not incur important delays, as it is meant to answer on-demand queries. Therefore, instead of adding Things individually to check the additional coverage they provide, we opted for a randomized approach that relies on a binary search to specify the size of the subset to select as follows:

1. Compute MPC.
2. If $MPC < c$, select the whole set.
3. If $MPC > c$, compute the coverage of a randomly selected subset s of size $|s| = 1$.
4. If the coverage is higher than c, return the selected subset. Otherwise, select a larger subset of size $|s| = |s| + \frac{R_\tau - |s|}{2}$ and compute the new coverage.
5. Repeat step 4 if needed.

Coverage is computed by approximating the covered area to a set of rectangles R_i, with each rectangle being a bounding box around $\zeta_{r_\tau,l}$ (the area covered by one Thing at location l). We leverage the *SweepLine algorithm* (`Algorithm 8`), illustrated in Figure 4.6, that takes the set of rectangles as input and generates an area value as output. In this algorithm, a line $L(x)$ is imagined to sweep a region and compute the area of the union of all rectangles within the region as follows :

Algorithm 8 `SweepLine` algorithm

Require: *Recs*
Ensure: *area*
{`A is the set of areas of rectangles of each consecutive x-coordinates`}
1: *let* $A \leftarrow \emptyset$
2: *let* $intersectingRectangles \leftarrow \emptyset$
3: $area \leftarrow 0$
4: $intersectionFound \leftarrow false$
5: $Ev \leftarrow sort(Recs, x)$ {`sort the x-coordinates of the rectangles`}
6: **for** *each* $\epsilon_{i+1} \in Ev$ **do**
7: $intersectingRectangles \leftarrow \emptyset$
8: $intersectionFound \leftarrow false$
9: **for** *each* $\epsilon_i \leq \epsilon_{i+1}$ **do**
10: $r \leftarrow \epsilon_i.Rectangle$
11: **if** r *intersects with* $\epsilon_i.Rectangle$ **then**
12: $intersectingRectangles \leftarrow r$
13: $intersectionFound \leftarrow true$
14: **end if**
15: **end for**
16: **if** $intersectionFound = true$ **then**
17: $length \leftarrow computeLength(intersectingRectangles)$
18: **else**
19: $length \leftarrow \epsilon_{i+1}.getRectangle.yMax - \epsilon_{i+1}.getRectangle.yMin$
20: **end if**
21: $A_i \leftarrow length * (\epsilon_{i+1} - \epsilon_i)$
22: $A.add(A_i)$
23: **end for**
24: **for** *each* $A_i \in A$ **do**
25: $area \leftarrow area + A_i$
26: **end for**
27: **return** *area*

1. Sort all X-coordinates of the rectangles in an event queue Ev, with each event ϵ_j, representing a new X-coordinate in Ev (Line 5).

2. Check if there is an intersection between ϵ_j and ϵ_{j+1} by checking if the coordinate of the upper border point of one of the rectangles is between the upper and lower border points of the other (Line 11).

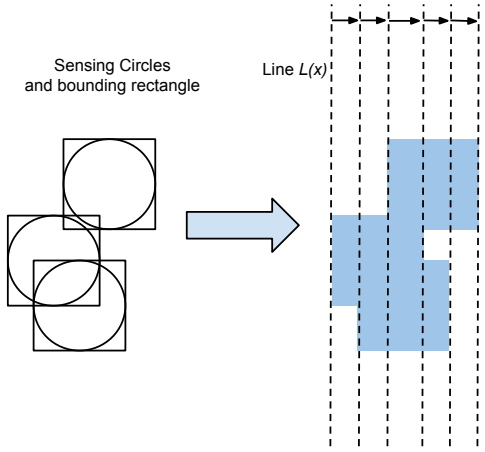

Figure 4.6. An illustration of the `SweepLine` Algorithm.

3. Compute the length of the intersection between the line $L(x)$ and the region between two events ϵ_j and ϵ_{j+1}. The length is computed based on the smallest and highest Y-coordinates at the current ϵ_j. The length remains constant until ϵ_{j+1} (Line 17). The method to compute the intersection length is presented in `Algorithm 9`.

4. The area $A_{\epsilon_j,\epsilon_{j+1}}$ between each two consecutive events is $L(\epsilon_j) * (x_{\epsilon_{j+1}} - x_{\epsilon_j})$, since we only consider rectangles (Line 21).

5. Sum all the areas to obtain the final coverage area (Line 24-27).

`Algorithm 9` computes the length of the intersection between a virtual line and a set of rectangles, some of which might have overlaps. It takes the set of rectangles as input and computes the intersection length as output, by summing up the intersection lengths (with the imaginary line) of different groups of intersecting rectangles. Each length is computed based on the difference between the lowest and highest Y-coordinates of the current group. The algorithm first sorts all rectangles by their Y-coordinates (Line 1). Afterwards, for each Y-coordinate, it checks whether it is the bottom border coordinate, or the upper border coordinate of a rectangle. If it is a bottom coordinate (Line 7), a counter of bottom points is incremented (Line 8). If the counter is 1, i.e., the coordinate is the first bottom coordinate to be encountered, it is set to be the lowest Y-coordinate (Line 9-10). If the coordinate is an upper

Algorithm 9 `computeLength` method

Require: $intersectingRectangles$
Ensure: $length$
1: $sortedCoordinates \leftarrow sort(R, y)$
2: $counterOfBottomPoints \leftarrow 0$
3: $counterOfUpperPoints \leftarrow 0$
4: $maxY \leftarrow 0$
5: $minY \leftarrow 0$
6: **for** $each\ c\ \in sortedCoordinates$ **do**
7: **if** $c.Y\ is\ bottom\ y\ coordinate$ **then**
8: $counterOfBottomPoints++$
9: **if** $counterOfBottomPoints = 1$ **then**
10: $minY \leftarrow c.Y$
11: **end if**
12: **else**
13: **if** $c.Y\ is\ upper\ y\ coordinate$ **then**
14: $counterOfUpperPoints++$
15: $maxY \leftarrow c.Y$
16: **end if**
17: **end if**
18: **if** $counterOfUpperPoints - counterOfBottomPoints = 0$ **then**
19: $length \leftarrow length + maxY - minY$
20: $counterOfBottomPoints \leftarrow 0$
21: $counterOfUpperPoints \leftarrow 0$
22: **end if**
23: **end for**
24: **return** $length$

border coordinate, the counter of upper points is incremented and the coordinate is set to be the highest Y-coordinate. Every new upper border coordinate is surely the highest coordinate encountered so far (Line 12-15). If the upper and lower point counters are equal (Line 18), the length of the intersection is computed (Line 19) and the counters are reset (Line 20-21). The equality in the counters reflects that remaining rectangles do not intersect with the so far evaluated ones. The final length is returned when all coordinates have been checked (Line 24).

The details of the approach are presented in `Algorithm 10`, which similarly to `Algorithm 6`, first interpolates the location of Things hosting needed services. It then constructs the grid (depending on the distribution) and identifies which Things

Algorithm 10 Coverage-Based Probabilistic Lookup

Require: $R_\tau, a, l, t_r, r_\tau, c$
Ensure: s
1: *let* $I \leftarrow \emptyset$
2: *let* $Q \leftarrow \emptyset$
3: *let* $servicesInGrid \leftarrow \emptyset$
4: *let* $recs \leftarrow \emptyset$
5: *let* $tempArray \leftarrow \emptyset$
6: *let* $tempServices \leftarrow \emptyset$
7: $counter = 0$
8: $I \leftarrow interpolate(R_\tau)$
9: $Q \leftarrow getGrid(l, a)$
10: $servicesInGrid \leftarrow identifySquares(I, Q)$
11: $recs \leftarrow generateCoverageRectangles(I, r_\tau)$
12: $MPC \leftarrow SweepLine(recs)$
13: $covPercentage = \frac{MPC}{a} * 100$
14: **if** $MPC < c$ **then**
15: $n \leftarrow servicesInGrid$
16: **else**
17: $subsetSize = 1$
18: **while** $covPercentage < c$ **do**
19: **for** *each square* $q_i \in Q$ **do**
20: **if** $|s| < subsetSize$ **then**
21: $tempServices \leftarrow q_i.getServices$
{`remove all services that are in the inner square`}
22: $tempServices.removeAll(tempArray)$
23: $tempArray = tempServices$
24: **while** $counter < \frac{|s|}{|Q|}$ **do**
25: $index \leftarrow math.random * tempServices.size$
26: **if** $s.contains(tempServices.get(index)) = false$ **then**
27: $s.add(tempServices.get(index))$
28: **end if**
29: **if** counter = tempServices.size **then**
30: *break*
31: **end if**
32: **end while**
33: **else**
34: $recs \leftarrow generateCoverageRectangles(s, r_\tau)$
35: $coverage = SweepLine(recs)$
36: $covPercentage = \frac{coverage}{a} * 100$
37: *break*
38: **end if**
39: **end for**
40: $subsetSize = subsetSize + \frac{servicesInGrid.size - |s|}{2}$
41: **end while**
42: **end if**
43: **return** s

belong to which square in the grid (Line 8 to Line 10). Afterwards, the algorithm calls the `generateCoverageRectangles()` method (Line 11), to model the area covered by

Algorithm 11 `generateCoverageRectangles` method

Require: s, r_τ
Ensure: s
1: $covRecs \leftarrow \emptyset$
2: **for** *each service* $\in s$ **do**
3: $loc \leftarrow service.location$
4: $curRec \leftarrow newRectangle(loc.XCoordinate - r_\tau, loc.YCoordinate - r_\tau, loc.XCoordinate + r_\tau, loc.YCoordinate + r_\tau, service)$
5: $covRecs.add(curRec)$
6: **end for**
7: **return** $covRecs$

each Thing as a rectangle. The `generateCoverageRectangles()` method, presented in `Algorithm` 11, creates a $(2r_\tau * 2r_\tau)$ square based on the coordinates of each Thing. The generated rectangles are then passed to the `SweepLine()` method to compute the MPC (Line 12). If the MPC is less than the threshold, all services in R_τ, hosted on Things estimated to be in a, are returned (Line 14-15). Otherwise, the algorithm sets the subset size to 1 (Line 17) and selects one random service hosted on a Thing located in square q_1 (Line 25) and computes its coverage (Line 34-36). If it is less than the threshold, the algorithm increases the subset size following a binary approach (Line 40) as follows :

$$subsetSize = subsetSize + \frac{|R_Q| - |s|}{2} \tag{4.32}$$

with R_Q being the set of services hosted on Things, estimated to be in the grid Q at the time of the request t_r. The algorithm then iterates over the random selection process (Line 18-40) and selects, uniformly, for each $q_i \in Q$, $\lambda = \frac{subsetSize}{|Q|}$ services. Whenever a service is selected, the algorithm checks if it has already been selected to avoid duplicates (Line 26-27). Once the desired number of services has been selected, or if all possible candidate services in the current square have been selected (Line 29-30), the algorithm moves to the next square. When starting the selection for the following square, the algorithm ensures all services selected in the inner squares, are removed from the list of candidate services, since the squares are concentric. (Line 21-23). Afterwards, the coverage by the Things hosting the selected services is computed through the `SweepLine()` method (Line 34-36) and the selection process for the current subset size is stopped (Line 37). If the coverage is less than the threshold, the subset selection is repeated again, and so on, until the required coverage is reached.

Complexity analysis. `Algorithm 10` terminates if the coverage requirement is satisfied. We ensure that the full coverage, which can be provided by all registered services in R_τ, is above the minimum required coverage before going into the outer most `while` loop (Line 18 - 37), making sure not to go into an infinite loop. We also make sure that selecting $\frac{|s|}{|Q|}$ services from a square q_i does not lead to an infinite loop by breaking the loop (Line 24 - 30) if all services in q_i are already selected. `Algorithm 10` invokes `Algorithm 3`, which iterates over the list of services in R_τ once and is sure to terminate. `Algorithm 10` also invokes `Algorithm 11`, which is sure to terminate as it consists of one `for` loop that iterates over a finite set of services. Additionally, `Algorithm 10` invokes `Algorithm 8`, which iterates over two finite sets containing the coverage rectangles and is sure to terminate. Finally, `Algorithm 10` also depends on `Algorithm 9` which contains one `for` loop (Line 6 - 21) that iterates over a finite set of rectangles. Therefore `Algorithm 10` is sure to terminate. Similar to the analysis in `Algorithm 6`, the `interpolation()` method in `Algorithm 7` has a time complexity of order O ($|R_\tau|$). In the worst case, the time complexity is O ($|R|$).

Similar to the analysis in `Algorithm 6`, the `interpolation()` method in `Algorithm 7` has a time complexity of order O ($|R_\tau|$). In the worst case, the time complexity is O ($|R|$).

Similar to `Algorithm 6`, the `getGrid()` method has a constant time complexity $\theta(\frac{2*d_u}{ratio})$.

As shown in the previous section, the `identifySquares()` method has a time complexity of order O ($|I| * |Q|$), in the worst case, the complexity is O ($|R|$).

The `generateCoverageRectangles()` method presented in `Algorithm 11` goes through each service in I once (to generate one rectangle per service) and has a time complexity of order O ($|I|$). In the worst case, the set of interpolated service I will contain all registered services in R, leading to $|I| = |R|$. The worst case complexity becomes O ($|R|$).

The `SweepLine` algorithm (`Algorithms 8 - 9`) is known to have a time complexity of order O ($|Ev|log(|Ev|)$).

`Algorithm 10` invokes the methods and algorithms above (`Algorithms 7, 11 - 9`). Moreover, in `Algorithm 10`, the size of the subset of services to evaluate (for coverage) is selected based on a binary search, therefore, the outer most loop (Lines 18 - 40) will have at most a time complexity of order O ($log(|I|)$). `Algorithm 10` also iterates over Q which is a finite set (the outermost `for` loop (Lines 19 - 37)). The

inner most `while` loop of `Algorithm 10` ((Lines 24 - 30)) depends on $\frac{|s|}{|Q|}$. As such, `Algorithm 10` has a time complexity $O(|R_\tau| + log(|I|) + \frac{2*d_u}{ratio} + |I| * |Q| + |Ev| * log(|Ev|) * |Q| * \frac{|s|}{|Q|})$. In the worst case scenario, all registered services in R will be evaluated and selected to be accessed, and all registered services will be in one square q_i, leading to $|s| = |I| = |Ev| = |R|$. Consequently, in the worst case, the time complexity of `Algorithm 10` becomes $\mathrm{O}\left(|R| + |Q| * \frac{|R|}{|Q|} + \frac{2*d_u}{ratio} + |R| * |Q| + log(|R|)\right)$.

Although the time complexity of the algorithm increases with $|R|$, we show in Chapter 6 that the overhead it incurs is compensated by the faster access times due to the strongly decreased resulting set of services to access.

It was shown in the previous section that the space complexity of `Algorithm 7` is $\mathrm{O}(|R_\tau|)$ and in the worst case, it becomes $\mathrm{O}(|R|)$. Similar to `Algorithm 6`, the `getGrid()` method has a constant space complexity $\theta(\frac{2*d_u}{ratio})$, and the `identifySquares()` method has a space complexity $\mathrm{O}(|I| + |Q|)$. In the worst case, the method has a space complexity $\mathrm{O}(|R| * |Q|)$.

`Algorithm 11` uses two sets: s, the set of services to generate rectangles for, and $covRecs$ the set of generated rectangles, $covRecs$ contains one rectangle of each service in s, leading to $|s| = |covRecs|$. The algorithm has a space complexity $\mathrm{O}(|s|)$. In the worst case, s and $covRecs$ will contain all registered services in R, therefore $|s| = |R|$. The resulting space complexity is $\mathrm{O}(|R|)$.

The `SweepLine` Algorithm (`Algorithms 8 - 9`) is known to have a space complexity $\mathrm{O}(Ev)$ with Ev being the variable, representing the set containing rectangles generated for services selected from R_τ.

`Algorithm 10` uses eight sets: i) R_τ, containing registered services, abstracting sensors/actuators of type τ; ii) I, containing the services with interpolated locations; iii) Q, the grid constructed over a; iv) $servicesInGrid$, containing services in Q; v) $recs$, containing the coverage rectangles generated for services in Q; vi) $tempArray$, holding services in the inner square q_{i_1} to be referred to when identifying services in q_i; vii) $tempServices$, containing randomly selected services in q_i; and viii) $finalSubset$, containing the addresses of all randomly selected services. The algorithm does not perform any preprocessing and contains no complex data structure. It has a space complexity $O(|R_\tau|+|I|+|Q|+|servicesInGrid|+|recs|+|tempArray|+|tempServices|+|finalSubset|)$. $|I| = |R_\tau|$ and $|servicesInGrid| = |recs|$ (since the set $recs$ contains one rectangle per service). Therefore the complexity reduces to $O(|R_\tau| + |servicesInGrid| + |tempArray| + |tempServices| + |finalSubset| + |Q|)$.

In the worst case, all services in R will be in R_τ leading to $|R_\tau| = |R|$, and all service in R will be in Q at time t_r, leading to $|servicesInGrid| = |R|$ and $|recs| = |R|$. Additionally, all services can be in one square q_i leading to $|tempServices| = |R|$ and $|tempArray| = |R|$ and all services will be selected, to be accessed later on, leading to $|finalSubSet| = |R|$. Therefore, the worst case space complexity is $\mathrm{O}(|R| + |Q|)$.

4.4 Summary

In this chapter, we presented our Thing-based service discovery approach, provided by MobIoT, that revisits the traditional Service-Oriented discovery. The novel approach controls, during registration and look-up phases, the participation of Things hosting sensing/actuating services. Firstly, we presented a Thing-based service registration solution that relies on the fact that mobile Things in dense networks are bound to cross paths and as such can substitute one another based on the services they host. This approach requires knowledge of the mobility of registered Things and requires computations proportional to the number of registered services. To separate the coverage computation from the set of registered services, and ensure better scalability, we replace the need to know the mobility of registered Things by a probabilistic mobility model to estimate the displacement of those Things, locally on each registering Thing. Secondly, we presented a probabilistic Thing-based look-up approach that continues on the work of the probabilistic registration phase and further decreases the number of services to select based on the location of their hosts and the coverage they provide.

Consequently, our probabilistic approaches, which are especially conceived to manage the mobility of Things while handling their ultra large number, ensures that any IoT application, exploiting those approaches, will scale and adapt to the dynamicity of the mobile IoT. We proceed to present, in the next Chapter, the smart composition approach that also revisits composition in SOA, to better handle the specificities of the mobile IoT.

Chapter 5

Thing-based Service Composition

Service Composition, a core component of Service-Oriented middleware solutions, creates a new functionality by combining the functionalities of different types of services. It can be exploited to substitute missing services or to increase the overall quality of the functionality provided by services. In MobIoT, services can be composed by finding a graph that, given a description of the inputs and a desired output, connects the available services in order to produce the desired output. In our context, the inputs can —in addition to being regular services if traditional SOA composition is to be applied— be the physical concepts and any constraints specified in the user query, which is the focus on this chapter. The importance of service composition in the mobile IoT is twofold: i) as services are hosted on mobile Things with limited power and resources, they can disappear abruptly. Consequently, the needed functionality, originally provided by the now missing service instance will no longer be available, while available services of other types can be combined together and provide the desired functionality; ii) no service exists to directly provide the required functionalities, while available services of other types can be combined and provide the desired functionality.

For instance, it is possible that at the time of the interaction with an IoT application, either no Thing in the network hosts an instance of the required service type (e.g., no wind-chill sensor exists or the Thing hosting the sensor left the network) or the available service instances do not satisfy the required constraints (e.g., none of the available thermometers provides temperature in Kelvin or none of the available thermometers is in Rome). In the above cases, it becomes necessary to find other

alternatives.

There are two primary requirements for composition of Thing-based services. Firstly, service instances are tightly bound to their hosts' physical aspects, especially their location and sensing/actuating capacities, which should be taken into account. Secondly, the IoT is a virtual representation of the real world governed by laws of physics and mathematics. This too should be taken into account when specifying and executing possible compositions. Thing-based service composition revisits Service-Oriented composition with this tight connection to physical hosts and the extended knowledge of real world sciences. More important, Thing-based service compositions should be executed seamlessly with no involvement from developers or end-users. To realize the aforementioned requirement, we ensure that all composition specifications be: i) hidden in ontologies; ii) specified independently by domain experts as mathematical formulas; and iii) extracted and executed transparently.

In this chapter, we present the details of our Thing-based service composition approach, that executes in three phases: i) *expansion*, where composition specifications are extracted; ii) *mapping*, where actual service instances are selected based on their functionalities and the physical attributes of their hosts; and iii) *execution*, where the services are accessed and the composition specifications are executed.

The key elements of our composition contribution are *exploiting semantically modeled mathematics and physics* to *automatically expand* a user query. We proceed in the following to give an overview of related works and necessary definitions associated with our contributions.

5.1 Background

Exploiting law of physics to compose new sensors to estimate the state of an observable feature in the real world, rather than directly measure it, is not a novel idea [Compton et al., 2009]. The estimation is done based on observations over other features, provided by different types of sensors and aggregated by a specific mathematical expression. However, to the best of our knowledge, focusing on the *physics and mathematics* behind those relations to seamlessly expand user queries and use the results as the basis of automatic composition of Thing-based services has never been proposed yet. Note that query expansion is not a novel idea and has been exploited for several purposes as presented below.

Query expansion. Expansion is usually used to augment a query with additional terms that convey the same meaning. The idea is not recent and has been a topic of interest since the 1960s [Carpineto and Romano, 2012]. The main use of expansion techniques has been directed towards vocabulary-related contexts with the outcome being synonyms to the main query terms [Carpineto and Romano, 2012]. This was recently exploited in SOA, for discovery purposes, as done in SOCRADES [Guinard et al., 2010] where synonyms are found to extend query terms and identify different service types that can answer a user query. Authors build their expansion on Web-based results (e.g., Yahoo! Web search). It is then up to the developer to select the service types he prefers. Bakilla *et al.* [Bakillah and Liang, 2012] also present a query expansion approach to enhance service discovery through expanded SQWRL queries. SQWRL is a query language for OWL ontologies based on SWRL rules. The main goal of the expansion is to return equivalent queries, based on the semantics and semantic distances between terms.

Unlike existing solutions, the query expansion we present in the following section, goes beyond terminology equivalence to exploit scientific equivalence, built on physics and mathematics, as the main expansion criteria. Expansions are specified semantically and extracted using SPARQL SELECT queries.

SPARQL. SPARQL is the W3C query language for RDF. It builds on a graph-matching approach where a SPARQL query, consisting of a set of triples, is matched against an RDF graph to find the subgraph that matches the query triple patterns. SPARQL triple patterns are similar to RDF triple patterns, except that one or more of the elements in the SPARQL triples are unknown. An unknown element is the variable to search for. A SPARQL SELECT query is a tuple `Q = (SELECT,E,G)`, with `E` being the expression to evaluate against the RDF graph `G`. E is built from a set of triples:

`t_s = (RDF-T` $\cup$ `V)x(I` $\cup$ `V)x(RDF-T` $\cup$ `V)`

with `RDF-T = (I` $\cup$ `B` $\cup$ `L)`, `I` the set of IRIs, `L` the set of RDF literals, `B` the set of blank nodes and `V` the set of query variables.

We use SELECT queries that have the following form :

`SELECT <`$v_1, v_2, \ldots, v_i$`> WHERE { <`$t\_s_1, t\_s_2, \ldots, t\_s_j$`>}`

with i being the total number of variables, and j the total number of constraint triples constituting the expression `E` to be matched against `G`.

The following example illustrates a SPARQL query to select all formulas that compute the value of a wind-chill concept. In this example, the variable is the unknown formula(s) to identify and the constraint is the fact that wind-chill is provided by the formula(s).

```
SELECT
?formula
WHERE {
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:wind-chill
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:
provided_by ?formula.}
```

5.2 Semantic Thing-based Service Composition

The Thing-based Service Composition wraps all MobIoT functionalities, namely Discovery and Access and determines the types of services to select and access by extending user queries. The general problem to solve by the Thing-based service composition is:

> *Given a network with an unknown dynamic topology, consisting of a set R of registered services hosted on location-aware mobile Things, determine and access the set S that contains services abstracting various types of sensors/actuators needed to answer a user query.*

Sensing and actuation queries received from IoT applications are expanded into several sub-queries through semantic composition graphs specified in the domain ontology. This takes place during the *expansion* phase. Types of services (abstracting sensor/actuator nodes in the graph) that can satisfy each sub-query are identified and mapped to the network topology to find corresponding instances during the *mapping* phase, Finally, the instances are accessed to provide a set of measurement/action results that should be aggregated or combined during the *execution* phase. The general approach is presented in Figure 5.1. The first question that rises is "*how to compute the wind-chill factor?*". The expansion phase provides an answer that wind-chill can be computed based on wind-chill measurements or temperature and

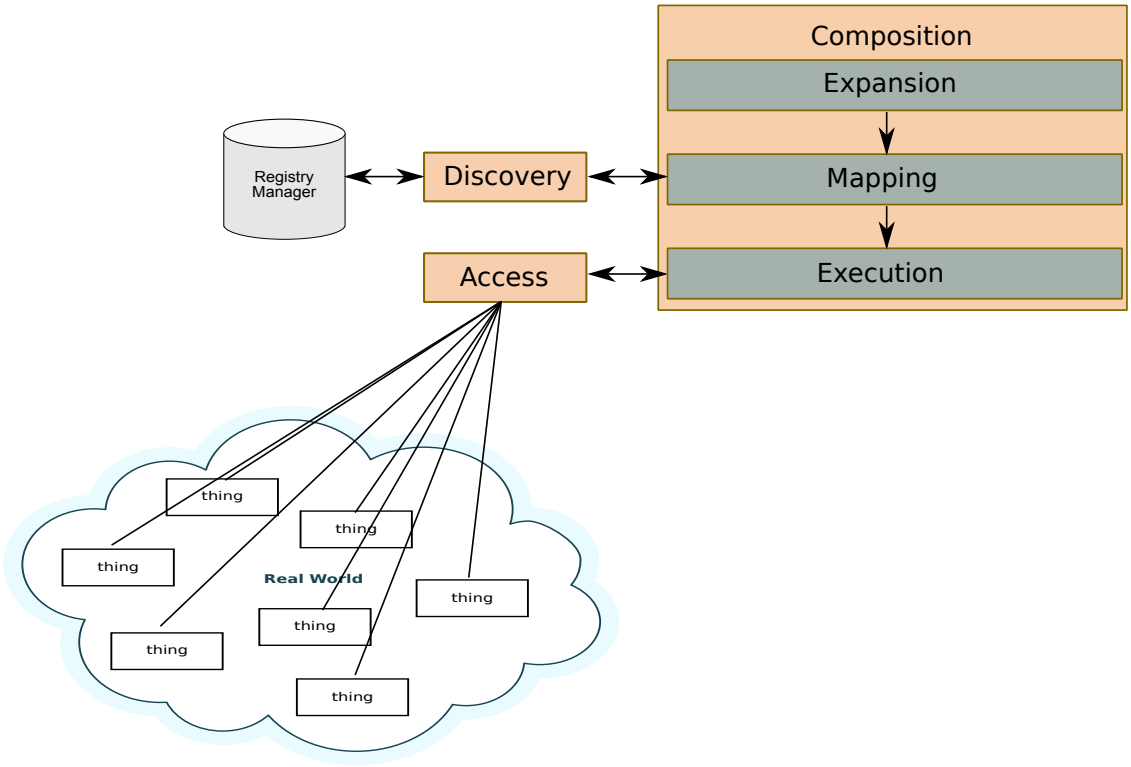

Figure 5.1. An overview of the Thing-based service composition.

wind-speed measurements. The next logical question is: "*how to combine temperature and wind-speed measurements to compute the wind-chill factor?*". The answer is provided within a mathematical expression returned, also, during the expansion phase. The third question that follows is: "*how to measure wind-chill, temperature and wind-speed?*". The mapping phase provides an answer that wind-chill sensors, thermometers and anemometers provide the respective measurements, leading to the final question: "*how to combine wind-chill, temperature and wind-speed measurement values?*". This question is addressed during the execution phase and the final answer is returned to Charlie. Details on the steps taken by each phase to reach the answers above are also presented in the following sections.

The different phases are detailed in the following sections. We also show how each phase takes the system employing the composition functionality a step further towards answering a user query. We illustrate the process through a sequence of logical questions that rise when the query is received. The user is Charlie, he wishes

to visit the Colosseum and he is wondering whether or not he should wear a jacket. An IoT application takes his request and translates it into a query for wind-chill factor information.

5.2.1 Expansion

Given a physical concept c_i initially specified in a query, the expansion phase creates an expression defined in terms of different *expansion concepts*. A physical concept is referred to as an expansion concept when it is used to estimate the state of another physical concept.

The expression is of the form:

$$\begin{aligned} c_i \;=\; & f_1(c^1_{e_1}, c^1_{e_2}, ..., c^1_{e_{m_1}}) \\ & + f_2(c^2_{e_1}, c^2_{e_2}, ..., c^2_{e_{m_1}}) \\ & + ... \\ & + f_i(c^i_{e_1}, c^i_{e_2}, ..., c^i_{e_{m_i}}) \end{aligned}$$

with $c^i_{e_{m_i}}$ being an expansion concept defined in the domain ontology and $f_i()$ a mathematical expression that determines the logic combining several expansion concepts to estimate the state of c_i. $f_i()$ is referred to as an *expansion formula*. Each expansion concept $c^i_{e_{m_i}}$ can have its own expansions, each of which can have different properties (unit of measurements for instance). The expansion is recursive, it does not only search for expansions over the initial concepts in the user query, but over the expansion concepts as well, until no expansion can be found.

All found expansion concepts are provided in a set $C_e = c \cup (\cup(c^i_{e_1}, ..., c^i_{e_{m_i}}))$. C_e is the union of all expansion concepts that should be measured or estimated to compute the expansion formulas. A sub-query is created for each expansion formula with the following parameters: i) c_i; ii) the location of interest, because each sub-query is treated independently; iii) the output parameter that links c to a unit of measurement; iv) the parameters to provide as input to the expansion formula. The parameters link each expansion concept to a measurement unit; and v) the mathematical expression of the expansion formula. The steps of the expansion are presented in `Algorithms` 12 - 14. Henceforth, we refer to a physical concept bound to a specific measurement unit as a `Parameter`. For instance, temperature is a physical concept,

while temperature_data_Kelvin is a `Parameter`.

`Algorithm 12` takes the user query, the set C of initial concepts c_i to measure/act on, and the location of interest as input. It generates a set of sub-queries as output. Before expanding a concept, the algorithm checks if it has already been expanded (Line 6). If so, it skips it and moves to the next concept. Otherwise, all expansion formulas that estimate the state of $c_i \in C$ are extracted from the domain ontology through a SPARQL query of the form (Line 7):

```
SELECT
?formula
WHERE {
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:c_i
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:
provided_by ?formula.}
```

Algorithm 12 Expand Concept Algorithm

Require: $mainQuery, C, location$
Ensure: $subQueries$
{set of mathematical expressions from the domain ontology}
1: $formulaSet \leftarrow \emptyset$
2: $conceptSubQueries \leftarrow \emptyset$ { set of subqueries for one concept}
3: $subQueries \leftarrow \emptyset$ {the set of all subqueries}
4: $subQueryParameters \leftarrow null$ { set of expansion concepts and units}
5: **for** $c_i \in C$ **do**
6: **if** $c_i \notin subQueries.concepts$ **then**
7: $formulaSet \leftarrow getFormulas(c_i)$
8: **for** $f \in formulaSet$ **do**
9: $sq \leftarrow subQuery(location, c_i, f)$
10: $conceptSubQueries.add(sq)$
11: $expandSubQuery(sq, location, c_i, f)$
12: **end for**
13: $subQueries.add(c_i, conceptSubQueries)$
14: **end if**
15: **end for**
16: **return** $subQueries$

Note that c_i should be substituted with the actual physical concept before the SPARQL query can be executed.

Afterwards, a sub-query is created for each formula by calling the `SubQuery` constructor presented in `Algorithm` 13. For each formula, the input and output `Parameters` are extracted through SPARQL queries of the form (Lines 3,4):

```
SELECT
?inputParameter
WHERE {
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:f_i()
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:
has_input ?inputParameter.}
```

```
SELECT
?outputParameter
WHERE {
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:f_i()
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:
has_output ?outputParameter.}
```

Algorithm 13 `SubQuery` Constructor

Require: $location, c, formula$
1: $locationToMeasure \leftarrow location$
2: $subQueryFormula \leftarrow formula$
3: $inputParams \leftarrow formula.InputParameters()$
4: $outputParam \leftarrow formula.OutputParamters()$
5: $mainConcept \leftarrow c$

Afterwards, `Algorithm` 12 triggers a recursive process to expand the expansion concepts (Line 9-11). The recursive process `expandSubQuery()` is presented in `Algorithm` 14 and follows almost the same steps as `Algorithm` 12. It first extracts the concept from each input `Parameter` (Line 1) in the SubQuery object sq, and checks if it has already been expanded (Line 6). If so, it skips it to the next concept. Otherwise, it acquires the expansion formulas from the ontology, and for each formula it creates a sub-query and calls the expansion process again (Line 7-11). The process repeats until no expansion can be found. In such case, the set of formulas will be empty and the `expandSubQuery()` method will not be called.

Algorithm 14 `expandSubQuery` method

Require: $sq, location, c, formula$

```
1: concepts ← sq.inputParams.Concepts {get concepts from Parameters}
2: subQueries ← ∅
3: subQueryList ← ∅
4: subQueryFormulaHolder ← ∅
5: for c ∈ concepts do
6:   if c ∉ subQueries.concepts then
7:     subQueryFormulaHolder ← getFormulaList(c)
8:     for f ∈ subQueryFormulaHolder do
9:       subQueryHolderObject ← SubQuery(location, c, f)
10:      subQueryList.add(subQueryHolderObject)
11:      expandSubQuery(subQueryParams, location, c, f)
12:    end for
13:    subQueries.put(c, subQueryList)
14:  end if
15: end for
```

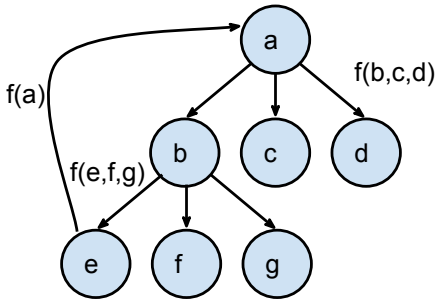

Figure 5.2. An example of a cyclic expansion.

Complexity analysis. `Algorithm 12` terminates when all expansions in `Algorithm 14` have been found. To ensure that there is no infinite loop, `Algorithms 12` and `14` check for cyclic expansions. Cyclic expansions mean that a concept is expanded into a graph with a child node that can be expanded into, among others, an ancestor node (Figure 5.2). The first step in the algorithm is to check whether or not the concept (being checked for expansions) has already been checked. The algorithm only continues if the answer is negative. `Algorithm 12` also depends on two `for` loops which iterate over the set of concepts C (Line 5 - 13) and the set $formulaSet$ (Line 8 - 11) which are finite. Finally, `Algorithm 12` invokes `Algorithm 13` which requires no complex computation or iteration and therefore is sure to terminate. Consequently,

`Algorithm 12` is sure to terminate.

`Algorithm 13` has a time complexity that depends on the SPARQL queries exploited to retrieve the `Parameters` from the domain ontology, which is independent of the problem size ($|R|$). Therefore, `Algorithm 13` has a constant time complexity $\theta(1)$.

`Algorithm 14` depends on the number of concepts to expand (in this case, the input `Parameters` to the expansion formulas) and their own expansions. The time complexity is $\mathrm{O}(|C| * |subQueryFormulaHolder|)$. In the worst case, the algorithm will go through all concepts and extract all formulas from the ontology. Since it only checks a concept once, assuming there are $|P|$ concepts and $|F|$ formulas in the domain ontology, the worst case time complexity will be $\mathrm{O}(|P| * |F|)$

The time complexity of `Algorithm 12` depends on the number of concepts in the initial query, i.e., the set C, and the number of expansion formulas in $formulaSet$. The algorithm also invokes `Algorithms 13` and `14`. `Algorithm 12` has a time complexity $\mathrm{O}(*|P| * |F| * |C| * |formulaSet|)$. In the worst case, the query will contain all concepts in the ontology, leading to $|C| = |P|$ (assuming there are $|P|$ concepts in the ontology), the algorithm will also extract all expansion formulas from the ontology leading to $|formulaSet| = |F|$ (assuming there are $|F|$ formulas in the ontology). Since a concept is only expanded one, the resulting time complexity is $\mathrm{O}(|P| * |F|)$.

`Algorithm 13` uses one set: $inputParams$, which depends on the number of `Parameters` to provide as input to the formula in the SubQuery object. Therefore, `Algorithm 13` has a space complexity $\mathrm{O}(|inputParams|)$. In the worst case, all `Parameters` in the ontology will be provided as input to the formula, leading to $|inputParams| = |M|$ (assuming there are $|M|$ `Parameters` in the ontology). The space complexity of `Algorithm 13` becomes $\mathrm{O}(|M|)$.

The space complexity of `Algorithm 14` depends on: i) $subQueryFormulaHolder$, containing the expansion formulas; ii) $subQueryList$, containing the sub queries that expand each input `Parameter` in f; and iii) $subQueries$, containing all generated sub-queries. The space complexity is $\mathrm{O}(|subQueries| + |subQueryFormulaHolder| + |subQueryList|)$. In the worst case, all sets will contain one sub-query for each formula in the ontology leading to $|subQueries| = |subQueryFormulaHolder| = |F|$, and $|subQueryList|$ will contain all formulas in the ontology, leading to $|subQueryList| = |F|$. The resulting complexity is $\mathrm{O}(|F|)$.

The space complexity of `Algorithm 12` depends on: i) the size of the set of

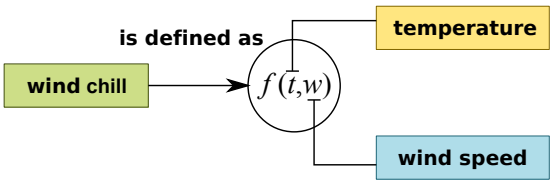

Figure 5.3. An example of the expansion phase.

concepts to expand C; ii) the size of the *subQueries* set, iii) the size of the $formulaSet$ set; and iv) the size of the *conceptSubQueries* set which contains one sub-query for each formula, therefore it is equal to the $formulaSet$ size. The space complexity of `Algorithm 12` is $O(|C| + |subQueries| + |formulaSet| + |conceptSubQueries|)$. In the worst case, the query will contain all concepts in the ontology, leading to $|C| = |P|$, the *subQueries* set will contain a sub-query for each formula in the ontology, leading to $|subQueries| = |F|$ and $|formulaSet| = |F|$. The resulting complexity is $O(|P| + |F|)$.

To answer the first question that rises from Charlie's query: "*how to compute the wind-chill factor?*", all possible physical concepts, i.e., types of measurements that can allow MobIoT to compute/measure the wind-chill factor information, are found. This is accomplished by expanding the initial query and replacing each term with (an) equivalent expression(s), found by traversing the domain ontology. In the wind-chill factor case, there are two possibilities:

1. Wind-chill factor measurements, assumed by default as a potential solution.
2. Temperature and wind-speed measurements, extracted from the domain ontology (Figure 5.3).

Another important question to answer is "*How to combine temperature and wind-speed measurements to compute the wind-chill factor?*". This information is also provided by the domain ontology where it is specified that:

$$wc = (10\sqrt{w} - w + 10.5) \cdot (33 - t) \tag{5.1}$$

where wc denotes the wind-chill concept, w denotes the wind-speed concept, and t denotes the temperature concept. In this case, temperature and wind-speed are the *expansion concepts*. For the formula to provide correct results, temperature should

be in *celsius* and wind-speed should be in *meter/second*. The resulting wind-chill factor unit is $kcal/m^2/hr$.

5.2.2 Mapping

The mapping phase allows the identification of the types of sensors/actuators needed to provide services that can measure/act on the expansion concepts in C_e. In this phase, the needed types of sensors/actuators are extracted into a set $T : \{\tau_1, \ldots, \tau_j\}$, with j being the total number of extracted sensor/actuator types. The mapping between concepts and corresponding sensor/actuator types is specified in the device ontology. Afterwards, the registered instances of services abstracting the types in T are discovered by interacting closely with the Discovery, resulting in a set of addresses, to access service instances, $S : \{s^1_{\tau_1}, \ldots, s^n_{\tau_j}\}$ with n being the total number of available service instances that abstract a sensor/actuator τ_j. The details of the mapping phase are presented in `Algorithm 15`.

The algorithm takes the set of sub-queries generated by the expansion phase as input and returns the set of addresses of service instances to interrogate as output. For each sub-query, it acquires the main concept c_i and the location of interest (Line 5-6). Afterwards the following SPARQL query is sent to the device ontology to extract the types of needed sensors/actuators to measure/act on c_i (Line 7):

```
SELECT
?type
WHERE {
?type
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:measures
<http://www.semanticweb.org/ontologies/2011/7/domain.owl#>:c_i.}
```

Once the types are acquired, the algorithm finds, for each type, the addresses of appropriate service instances to access, at the location of interest, through the discovery functionality, precisely, the look-up functionality called in `Algorithm 16` (Line 11). Once all types have been checked, the algorithm returns the set of found addresses (Line 14).

If the location is an area, `Algorithm 16` calls the area-based look-up (Line 2). If the location is a point, `Algorithm 16` calls the point-based look-up (Line 5). Both

look-up approaches were presented in Chapter 4. The composition process also exploits the grid, constructed during discovery, to be used upon execution when performing data aggregation (Line 6).

Algorithm 15 Mapping Algorithm

Require: *subQueries*
Ensure: S {`the set of registered service instances to access`}
1: $T \leftarrow \emptyset$
2: $S \leftarrow \emptyset$
3: $a \leftarrow \emptyset$
4: **for** $sq \in subQueries$ **do**
5: $c \leftarrow sq.concept$
6: $l \leftarrow sq.location$
7: $\tau \leftarrow getType(c)$ {`queries device ontology for sensor/actuator types`}
8: $T.add(\tau)$
9: **end for**
10: **for** $\tau \in T$ **do**
11: $a \leftarrow findServiceAddresses(\tau, l)$
12: $S.add(a)$
13: **end for**
14: **return** S

Algorithm 16 `findServiceAddresses` method

Require: τ, l
Ensure: a {the set of addresses of service instances}
1: **if** $L.type = area$ **then**
2: $a \leftarrow findSubSetOfServicesInArea(\tau, l)$
3: **else**
4: **if** $L.type = point$ **then**
5: $a \leftarrow findSubSetOfServicesAroundPoint(\tau, l)$
6: $Grid \leftarrow getGrid()$ {`retrieve grid constructed during discovery`}
7: **end if**
8: **end if**
9: **return** a

Complexity analysis. The mapping algorithm, `Algorithm 15`, iterates over two `for` loops that depend on the *subQueries* set and the sensor types set T, which are fi-

nite given that, during expansion, the algorithm ensures to avoid cyclic expansions by checking each concept only once guaranteeing to have no infinite loop and no infinite number of sub-queries. Therefore the algorithm is sure to terminate. `Algorithm 16` requires no complex computation or iteration and is sure to terminate.

The time complexity of `Algorithm 15` depends on the number of sub-queries in *SubQueries* and the number of types in T, in addition to the complexity of the discovery functionality in `findServiceAddresses()` method in `Algorithm 16`. `Algorithm 15` has a time complexity $\mathrm{O}(|subQueries| + |T|)$. In the worst case, the *subQueries* set will contain sub-queries for each formula in the domain ontology, leading to $|subQueries| = |F|$, with $|F|$ being the total number of formulas, and the types set T will contain all sensor/actuator types defined in the device ontology. For the complexity analysis of the look-up methods, we refer the reader to Chapter 4.

The space complexity of `Algorithm 15` depends on four sets: i) the *subQueries* set; ii) the types set T; iii) the set a of addresses of service instances, that should be accessed, for each type in T; and iv) the set S containing addresses of service instances, to access, for all types. The mapping phase has a space complexity $\mathrm{O}(|subQueries| + |T| + |S| + |a|)$. In the worst case, the *subQueries* set will contain a sub-query for each formula in the ontology, T will contain all sensor/actuator types that can measure the concepts of interest, and $|S|$ and $|a|$ will contain all registered services in R. Therefore, the mapping phase has a worst case space complexity $\mathrm{O}(|F| + |T| + |R|)$.

Continuing with Charlie's request, the next question to answer is: "*How to measure wind-chill, temperature and wind-speed?*". The answer is provided by interacting with the device ontology, where the types of sensors needed to measure wind-chill, temperature, and wind-speed are specified. The needed types are: wind-chill sensor, thermometer and anemometer. Once those types are identified, registered service instances that abstract wind-chill sensors, thermometers and anemometers are discovered, as illustrated in Figure 5.4.

5.2.3 Execution

The execution phase takes the set S of service addresses from the mapping phase, and by leveraging the access functionalities, acquires measurements or requests actions from the corresponding service instances. Once all services are accessed, resulting

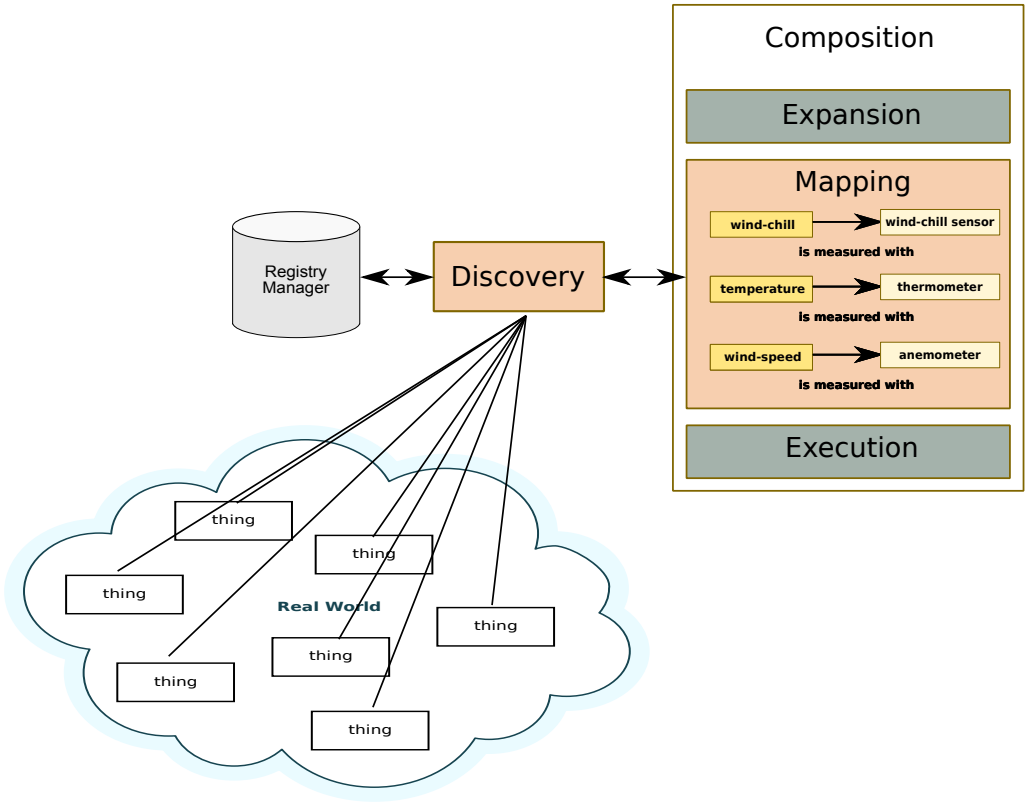

Figure 5.4. An example of the mapping phase.

values are returned in a set $V_{C_e} : \cup_{C_e}(\cup_{v_{c_{e_i}}}(v^1_{ce_i}, \ldots, v^{|a|}_{c_{e_i}}))$, with $|a|$ being the size of the set of addresses of services that measure/act on concept c_{e_i}. During this phase, all compositions, i.e., expansion formulas, are computed before returning the answer to the IoT application. The answer should be a set of measurement values corresponding to the initial concepts in the user query only.

The final outcome of this phase is a set of values $V_C = \cup_C(\cup_{v_{c_i}}(v^1_{c_i}, \ldots, v^j_{c_i}))$, with j being the possible number of individual values (measured or computed) for concept $c_i \in C$. An important requirement is that measurements of each expansion formula be co-located. Collocated measurements are provided by measurements hosted on Things that are within the same cell. The cells are constructed by dividing each square q_i in the grid G into smaller squares. The value of the ratio depends on the real life scenario and the sought after area division scale. Assuming L_g is the set containing the locations l_g corresponding to the centers of the constructed cells, with $l_g \in L_g$, the execution problem can be stated as follows:

Given an expansion formula $f()_i$, an area a, and the set of locations $l_g \in L_g$, estimate the state of c_i in a, based on measurement values $v_{c_{e_i}}$ of different expansion concepts $c_{e_i} \in C_e^{f()_i}$ provided at the different locations $l_g \in L_g$.

To consider that an instance of $f()_i$ can be properly computed within cell g, measurement values for all concepts in formula $f()_i$ should be hosted on Things that are at distance d from l_g with $d =$ (cell width $\frac{w_c}{2}$).

Another important requirement is that the units of the measurement values match the requested units of the input `Parameters` for each expansion formula. We remind readers that a `Parameter` wraps a physical concepts and a specific measurement unit. If the units do not match, conversions are a must. To that end, whenever there is a unit mismatch, the following SPARQL query is sent to the SWEET units ontology that we integrated with our domain ontology:

```
sweet :  <http://sweet.jpl.nasa.gov/2.2/reprSciUnits.owl#>
SELECT ?shift ?scale ?p ?type ?powerOfUnit
WHERE { ?p reprSciUnits:hasBaseUnit ?q.}
FILTER {(
?p = sweet:firstUnit.name) || ?p = sweet:secondUnit.name) &&
?q = sweet:firstUnit.name) || ?q = sweet:secondUnit.name))
OPTIONAL {?p sweet:hasShiftingNumber ?shift .}
OPTIONAL {?p sweet:hasScalingNumber ?scale .}
OPTIONAL {?p sweet:toThePower ?powerOfUnit }
```

The are three types of unit conversions: shifting units, when a number should be added or subtracted; scaling units, when the measurement should be multiplied by a value; and powerOf conversion, when the measurement should be raised to the power of a number to match a new unit. Those operations are not exclusive and can be combined together to convert a unit to another unit. Since it is not known which of the operations apply a priori, they are specified in the OPTIONAL clause. The mapping between two units is unidirectional, therefore it is important to take this into account when extracting the conversion formula. This is accomplished by the FILTER expression which checks whether the initial unit is the input or the output of the conversion formula.

The details of the final answer computations are presented in `Algorithms` 17 to 20.

`Algorithm` 17 takes the set of addresses as input and calls the access functionality to acquire service measurements or ask services to perform some actions. For each provided measurement, the algorithm creates a `Parameter` (Line 5) and links it to the measurement value (Line 6). Afterwards, the `Parameters`, their measurement values, the sub-queries and the initial concepts are sent to a computation function (`Algorithm` 18) that returns the final values for each initial concept (Line 8).

Algorithm 17 Execution Algorithm

Require: $C, subQueries, S, locs$
Ensure: V {`the set of requested values either measured directly or computed`}
1: $V \leftarrow \emptyset$
2: $V_e \leftarrow \emptyset$
3: $V_e \leftarrow access(S)$
4: **for** $v \in V_e$ **do**
5: $param = Parameter(v.type, v.unit)$ {`location information of each measurement is provided in the value object`}
6: $ParamsValues.add(param, v)$
7: **end for** {`expansion functions are computed and units are checked and converted`}
8: $V \leftarrow computeFinalValues(C, ParamsValues, subQueries, locs)$
9: **return** V

`Algorithm` 18 presents our work on data computation. The algorithm takes the estimated locations of hosts providing the measurements, the values of the measurements, the sub-queries, the initial concepts C, and the location of interest as input. It generates an estimate of the value of the initial concepts as output. The algorithm first acquires all the `Parameters` to evaluate (Line 3). Afterwards, for each concept $c_i \in C$, it checks if measurements or actions on c_i have been directly provided by sensing/actuating services and acquires the measurements (Line 5). It follows that the algorithm retrieves all sub-queries, which estimate the state of c_i (Line 6), to then initiate a recursive computation process through the `compute()` method presented in `Algorithm` 19 (Line 7). The result of the `compute()` method is a set of measurement

values that correspond to the initial concept c_i. The algorithm then checks the next concept and so on until all values for all initial concepts have been computed.

Algorithm 18 `ComputeFinalValues` method

Require: $C, parameterValues, subQueries, r_s, locs$
Ensure: $finalResult$
1: $tempSubQueries \leftarrow \emptyset$
2: $finalResult \leftarrow \emptyset$
3: $params \leftarrow parameterValues.params$
4: **for** $c_i \in C$ **do**
5: $m \leftarrow parameterValues.getMeasurements(c_i)$ {`find measurements for` c_i}
6: $tempSubQueries \leftarrow subQueries.get(c_i)$
7: $eMeasurement = compute(subQueries, tempSubQueries, parameterValues, r_s, locs)$
8: $measurements.add(eMeasurement)$
9: $finalResult.add(c_i, measurements)$
10: **end for**
11: **return** $finalResult$

`Algorithm 19` starts by iterating over the list of sub-queries for a concept c_i (Line 6) and retrieves the expansion formula f associated to each sub-query (Line 7) and the input `Parameters` to f (Line 8). For each input `Parameter`, it acquires the corresponding measurement values (Line 9-11) and the expansions for the concept it wraps (Line 12). The recursive computation is called again (Line 13) until all `Parameters` have been checked and their expansions have been extracted. The algorithm calls the `pairWiseComputation()` method presented in `Algorithm 20` to ensure that sets of co-located measurements are computed together. `Algorithm 19` is designed to construct a graph of `Parameters` and their corresponding measurement values. The graph resembles the expansion graph created in the expansion phase. The end goal is to be able to compute the expansion formulas in a bottom-up approach starting from leaf nodes. Leaf nodes represent measurements of `Parameters` provided as input to the bottom level expansions. Bottom level expansions are found in the last round of the expansion phase. The algorithm computes the lowest expansion formulas, then goes one level higher and so on, until the parent nodes representing the `Parameters` wrapping the initial concepts are reached.

As stated earlier, different types of measurements are co-located if they are provided by Things that have a distance $d < \frac{w_c}{2}$ from the center of a cell g. This

Algorithm 19 `Compute` method

Require: $subQueries, conceptSubQueries, params, locs$
Ensure: V
1: $paramMeasurement \leftarrow \emptyset$
2: $computedValues \leftarrow \emptyset$
3: $msrments \leftarrow \emptyset$
4: $pMeasurements \leftarrow \emptyset$
5: $result \leftarrow \emptyset$
6: **for** $csq \in conceptSubQueries$ **do**
7: $f \leftarrow csq.Formula$
8: $parameters \leftarrow csq.InputParameters$
9: **for** $p \in parameters$ **do**
10: **if** $p \notin pMeasurement$ & $p.concept \in subQueries.concepts$ **then**
11: $msrments \leftarrow params.get(p.concept)$
12: $sq \leftarrow subQueries.get(p.concept)$
13: $eMeasurement = compute(subQueries, sq, params, locs)$
14: $measurement.add(eMeasurement)$
15: $pMeasurement.add(param, measurements)$
16: **end if**
17: **end for**
18: $rsl \leftarrow pairWiseComputation(pMeasurement, f, locs)$
19: $pMeasurement \leftarrow \emptyset$
20: $V.add(result)$
21: **end for**
22: **return** V

requirement is accounted for in `Algorithm 20`. The algorithm takes, as input, an expansion formula f. f is specified in terms of `Parameters`, which should be substituted with actual measurement values for the formula to be executable. The `Parameters` are provided as input to the algorithm, along with their expansions, and all the corresponding measurement values. The algorithm generates the final result of the expansion formula as output. The algorithm iterates over each location l_g and checks if the unknown variables of the expansion formula have already been replaced with actual measurement values (Line 6). If so, it computes the formula (Line 7). If not, it iterates over the measurements, and searches for measurements at distance $d < \frac{w_c}{2}$ from l_g (Line 13). If a measurement is found, the algorithm compares its unit to that of the corresponding `Parameter` for correctness. If they do not match, a SPARQL query is sent to the domain ontology to extract and execute the conversion (Line 14).

Algorithm 20 `pairWiseComputation` method

Require: $measurements, f, locs$
Ensure: $result$
1: $result \leftarrow \emptyset$
2: $flag \leftarrow 0$
3: $counter \leftarrow 0$
4: $canCompute \leftarrow false$
5: **for** $loc \in locs$ **do**
6: **if** canCompute = true **then**
7: $v \leftarrow calculate(f)$
8: $result.add(v)$
9: **else**
10: **for** $m \in measurements$ **do**
11: $flag \leftarrow 0$
12: **if** measurement.checked = false **then**
13: **if** $InRange(measurement, loc)$ **then**
14: $m.Value \leftarrow convertUnit(f.Unit, m.Unit, m.Value)$
15: f.setParam(p,measurement.Value)
16: $measurement.checked = true$
{`check the formula to determine if all parameter variables have been replaced with values`}
17: $canCompute \leftarrow f.checkParam$
18: **end if**
19: **if** canCompute = true **then**
20: $v = calculate(f)$
21: $result.add(v)$
22: $break$
23: **end if**
24: **end if**
25: **end for**
26: **end if**
27: **end for**
28: **return** $result$

Afterwards, the algorithm replaces, in f, the `Parameter` corresponding to the current measurement type, with the measurement value (Line 15). The algorithm then checks if all variables have been replaced (Line 17). If so, the boolean canCompute is set to true. If canCompute is true, an instance of f is computed and the algorithm moves to the next location in L_g (Line 19-22), otherwise it checks the next measurement. If

any of the measurements at a location l_g is missing, the whole set of measurements at l_g is ignored. An alternative is to estimate the value of the missing measurements from similar measurements at other locations, which is left for our future work.

Complexity analysis. `Algorithm 17` depends on one `for` loop (Line 4 - 6), which iterates over the set of measurement values provided by accessed services. It also invokes `Algorithms 18 - 20`. `Algorithm 18` depends on one `for` loop over the set C of initial concepts (Line 4 - 9). The set C is finite. `Algorithm 19` iterates over two sets: the *ConceptSubQueries* and the *Parameters* sets, both of which are finite and sure to terminate. Given that we ensured to have no infinite cycle between expansions in the expansion phase, and `Parameters` are only checked once, the recursive aspect of the algorithm will not lead to an infinite loop. `Algorithm 19` also invokes `Agorithm 20`, which iterates over two finite sets: *locs*, the set of locations where measurements should be provided, and *measurements*, containing measurement values for the different `Parameters` of an expansion formula. There is no recursive call and this algorithm is sure to terminate. Consequently, all algorithms are sure to terminate.

The time complexity of `Algorithm 20` depends the number of locations in the *locs* set and the number of measurements in the *measurements* set. The algorithm has a time complexity $\mathrm{O}(|locs| * |measurements|)$. In the worst case, there will be $|R|$ measurements provided by all registered services and the complexity becomes $\mathrm{O}(|R| * |locs|)$.

The time complexity of `Algorithm 19` depends on the number of sub-queries in *ConceptSubQueries* and the number of `Parameters` for each sub-query in the *parameters* set, in addition to the depth of the recursive calls of the function. Consequently, the time complexity of `Algorithm 19` is $O(depth * |conceptSubQueries| * |parameters|)$. In the worst case, the algorithm will go through all `Parameters` in the ontology leading to $|parameters| = |M|$ (assuming there are $|M|$ `Parameters` in the ontology). Given that expansions are only extracted once for each concept and `Parameters` are checked only once in the compute function, depth will be equal to $|M|$ at most. Additionally, *conceptsSubQueries* will contain all formulas in the ontology, leading to $|conceptSubQueries| = |F|$. Therefore, the worst case time complexity is $\mathrm{O}(|M| * |F|)$.

The time complexity of `Algorithm 18` depends on the size of the set C. The time

complexity of `Algorithm 18` is O ($|C|$). $|C|$ depends on the number of concepts in the domain ontology. In the worst case $|C| = |P|$, leading to O ($|P|$).

The time complexity of `Algorithm 17` depends on the number of measurement values, which is equal to the number of service instances to access, and the complexity of the function employed to compute the final result, which includes invocations of `Algorithms 18 - 20`. The complexity of `Algorithm 17` is $O(|S|+|R|*|P|*|M|*|F|)$. In the worst case, all registered services in R should be accessed to provide measurement values leading to $|S| = |R|$. The complexity becomes O ($|R| * |P| * |M|$).

the space complexity of `Algorithm 20` depends on three sets: i) *locs*, containing the locations where the formula instances should be computed; ii) *measurements*, containing the measurement values provided by the selected services; and iii) *result*, containing the computed values for each location. The set *result* depends on the number of locations as, ideally, it should contain a computed value, for the expansion formula, at each location. Therefore, the space complexity is O ($|locs| + |measurements|$). In the worst case, there will be $|R|$ measurements (one measurement for each registered service in R), leading to $|measurement| = |R|$. The resulting space complexity is O ($|R| + |locs|$).

The space complexity of `Algorithm 19` depends on the following sets: i) *subQueries*, containing expansions for all concepts; ii) *conceptSubQueries*, containing expansions for the concept currently being computed; iii) *params*, containing all `Parameters` and their measurement values; iv) *parameters*, containing input `Parameters` for each expansion formula; v) *msrments*, containing measurement values, for each `Parameter`, provided directly by service instances; vi) $pMeasurements$, containing measurement values that can be used to estimate the state of each `Parameter` and provided by expansions services; and vii) V, containing the final results to return. The space complexity of `Algorithm 19` is $O(|subQueries| + |conceptSubQueries| + |params| + |parameters| + |pMeasurement| + |msrments| + |V|)$. In the worst case, there will be a sub-query for each formula in the ontology, leading to $|Subqueries| = |F|$. It is also possible that all expansion formulas in the ontology be defined for the concept currently being evaluated, assuming there are $|F|$ formulas in the ontology, we obtain $|conceptSubQueries| = |F|$. Additionally, $pMeasurements$ and *msrments* will contain measurements from all registered services in R, leading to $|pMeasurements| = |measurements| = |msrments| = |R|$. Finally, V will contain a computed value for each concept in the ontology, i.e., $|V| = |P|$ and *params* will

contain a measurement from each registered service in R (for every `Paramater` in the ontology), leading to $|params| = |M| * |R|$. The resulting space complexity is $O(|M| * |R|)$.

`Algorithm 18` depends on six sets: i) C, containing the initial concepts; ii) *subQueries*, containing the expansion sub-queries; iii) $parameterValues$, containing all `Parameters` of the expansion formulas and their measurement values; iv) *params*, containing `Parameters` to evaluate (similar to the `Parameters` in $parameterValues$); v) *tempSubQueries*, containing sub-queries for each concept; and vi) $finalResult$, containing initial concepts and their measurement values. The space complexity of `Algorithm 18` is $O(|C| + |subQueries| + |parameterValues| + |params| + |tempSubQueries| + |finalResult|)$. In the worst case, the $parameterValues$ set will contain values from all registered services in R, the initial query will contain all concepts in the ontology, leading to $|C| = |P|$, *subQueries* and *tempSubQueries* will contain all formulas in the ontology, leading to $|subQueries| = |tempSubQueries| = |F|$. Additionally, $finalResult$, which contains only one value for each concept in the query, will in the worst case, contain one value corresponding to each concept in the ontology, leading to $|finalResult| = |P|$. Finally, *params* will contain a measurement from each registered service in R for every `Paramater` in the ontology, leading to $|params| = |M| * |R|$. In the worst case, the space complexity becomes $O(|P| + |F| + |R| + |M| * |R|)$.

`Algorithm 17` uses five data sets: i) C the set of concepts; ii) *subQueries*, the set of expansion sub-queries; iii) S, the set of service instances to access; iv) V_e, the set of measurement objects; and v) V the set of final results with one result for each concept $c_i \in C$. The space complexity of `Algorithm 17` is $O(|C| + |subQueries| + |S| + |V_e| + |V|)$. In the worst case, all registered services will be asked for measurements and consequently $|S| = |R|$ and $|V_e| = |R|$ (each service provides one measurement). Moreover, the initial query will contain all concepts in the ontology, leading to $|C| = |P|$, $|V|$ will contain a final result for each concept in the ontology, leading to $|V| = |P|$, and the set *subQueries* will contain all formulas in the ontology, leading to $|subQueries| = |F|$. The worst case complexity becomes $O(|P| + |F| + |R|)$.

The execution phase answers the last question "*how to combine wind-chill, temperature and wind-speed measurement values?*", after which the wind-chill factor is computed and returned to the IoT application. Firstly, the service instances discovered during the mapping phase are accessed and the individual results are returned.

However, Charlie does not care for individual results. Moreover, the application developer is not aware of the internal expansions that took place, i.e., the fact that, in addition to wind-chill measurements, temperature and wind-speed measurements were acquired. The developer will therefore not know how to treat those measurements or comprehend the physics behind our expansions. To alleviate that burden, the execution phase returns only wind-chill values:

- The values directly sensed by a wind-chill sensor.
- The values computed by the different instances of the expansion formula $f(t, w)$, as illustrated in Figure 5.5.

Rules of mathematics and physics are once again of utmost importance in this phase. For instance, in this example, there are two rules that should be taken into account:

1. $average(f(t, w)) \neq f(average(t, w))$. This is assumed by default.
2. Temperature measurements should be provided in *celsius*. Otherwise, they should be converted. Similarly, wind-speed measurements should be provided in *meter/second*. Otherwise, they should be converted.

To satisfy the first condition, the temperature and wind-speed measurements should be aggregated based on their location. For the wind-chill expansion formula $f(t, w)$ to be computed correctly, the values of temperature and wind-speed to provide as input should be co-located. To consider that a temperature measurement value v_t and a wind-speed measurement value v_w are co-located, they should be provided by sensors hosted on Things within distance $d < \frac{w_c}{2}$ from one of the locations $l_g \in L_g$. To satisfy the second condition, the unit of each measurement is compared to the required unit (provided with the `Parameters`, e.g., temperature_data_C, denoting temperature in celisius, to provide as input to the wind-chill expansion formula). If it does not match, it is converted by exploiting the conversion formulas.

Discussion The approach we presented for automatic composition alleviates the burden of specifying complex compositions on developers and end-users as the process is performed separately by a domain expert. Additionally by exploiting the mathematical expansions, the composition approach ensures better efforts are provided to exploit available services of different types to answer a user query. However,

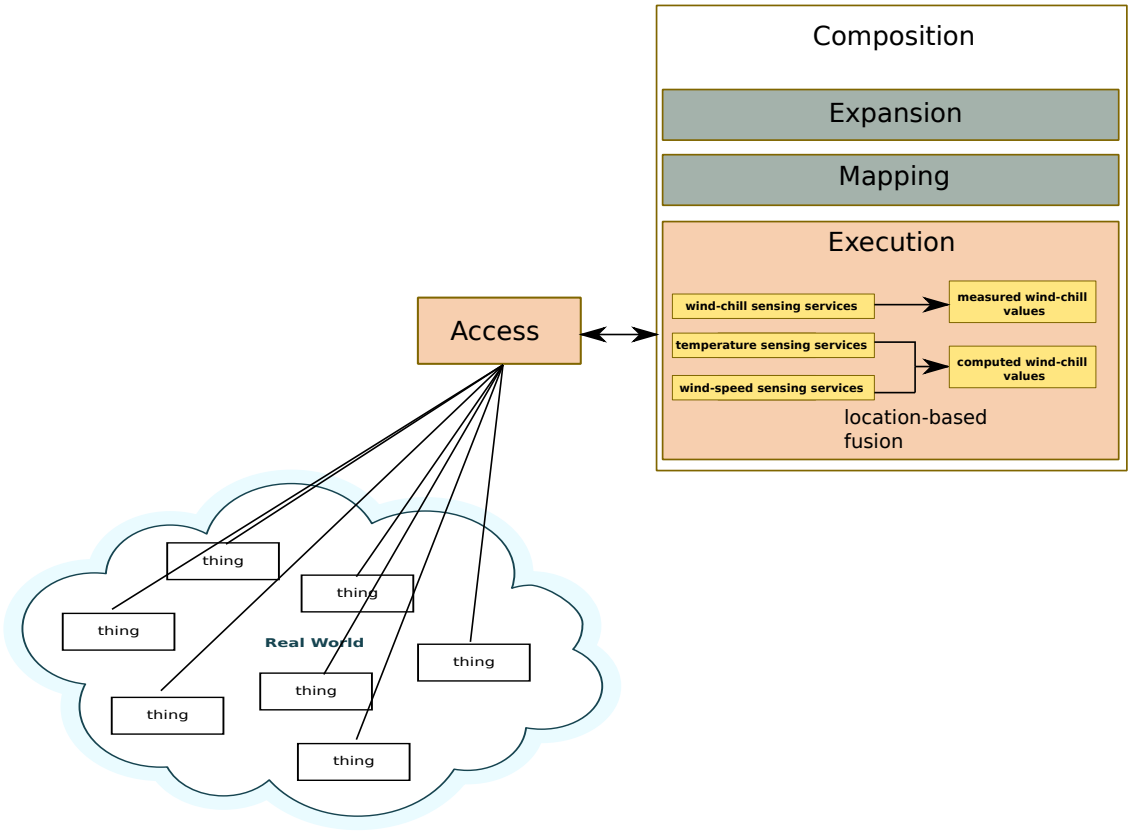

Figure 5.5. An example of the execution phase.

the approach for the final answer computations needs to be further investigated. Firstly, we plan on relaxing the assumptions pertaining to collocation, and impose a requirement for co-located Things to be within a distance $d < 2r_\tau$ away from each other. Evidently, our problem reduces to a geometric intersection problem. It can also be modeled as a nearest neighbor search problem to find the nearest Things with registered services to a Thing q:

> *Given a set P of points in a normed space l_p^d, preprocess P so as to efficiently return a point $p \in P$ for any given query point q, such that the distance $d(q,p) \leq (1+\epsilon)d(q,P)$ where d(q,P) is the distance of q to its closest point in P.*

In our case, d(q,P) should be $2r_\tau$. One particular technique of interest, to specifying Things to partake in the expansion formula computations, is the Locality Sensitive

Hashing technique introduced by Indyk and Motwani [Indyk and Motwani, 1998]. The approach is designed to group points in a search space based on a distance metric, possibly Euclidian distance, and their probability of satisfying the required distance. Although designed to address the issue of highly dimensional data through hashing, it is aimed at handling large volumes of data and increasing search performances, which matches our requirements.

Moreover, the data computation process can be optimized. There is a large plethora of scalable techniques that we can build on to enhance the performance of our solution.

The common approach to scale data processing is to perform in-network data aggregation [Akyildiz et al., 2002] by Things in the network, mostly heads of clusters (of Things) elected either at deployment time or at run time in ad hoc manner. Clustering is commonly used in sensor networks for network scalability [Zhao and Raychaudhuri, 2009], load balancing [Cheng et al., 2009, Heinzelman et al., 2002], data aggregation [Wu et al., 2010] or energy efficiency [Heinzelman et al., 2002]. To the best of our knowledge, to our day, clustering solutions require some stability in the network either by having quasi-static cluster heads, or expecting the cluster formations to be steady for some periods of time [Younis and Fahmy, 2004, Chatterjee et al., 2002, Awwad et al., 2011, Yoo and Park, 2011]. This assumption does not comply with the dynamicity of the IoT environment, which complicates the problem to solve. Consequently, although the existing solutions provide a solid starting point they still have limitations and need to be revisited. We consider the following alternative. Clusters are formed during the look-up phase while creating the selection grid over the area of interest. During this phase, Things in each cell constitute a cluster, and the Thing with the most resources becomes the cluster head and handles, locally, the aggregation of data provided by the cluster Things by injecting the composition graph into the cluster head. Power consumption and communication costs can be handled by adopting, for instance, Bluetooth-based communication for neighboring devices as done in [Yoo and Park, 2011] where authors present a cooperative clustering protocol that exploits nodes with WLAN and Bluetooth interfaces where in-cluster nodes communicate through Bluetooth and the cluster head acts as a gateway with the WLAN.

5.3 Summary

In this chapter, we presented our Thing-based service composition approach, provided by MobIoT. Thing-based service composition revisits the traditional Service-Oriented composition by requiring no involvement from application developers, but only domain experts that specify, independently, compositions and all related knowledge in the domain ontology. It also builds on knowledge of real world sciences as the core of the compositions. The Thing-based service composition is executed automatically and seamlessly through three phases. The first phase performs the expansion, where initial concepts to measure/act on are extracted from the user query. Afterwards, mathematical functions that allow the estimation of the state of the initial concepts are extracted. This is crucial in case services that can directly measure the initial concepts are missing. It also allows to enhance the measurements/actions quality by providing additional options. The functions build on laws of physics modeling how the state of different concepts, expansion concepts, can allow the estimation/modification of the state of the initial concept. Afterwards, the corresponding types of sensors/actuators are identified from the device ontology. Instances of services abstracting the identified sensors/actuators are then discovered through the discovery functionality. This is the mapping phase. Finally, in the execution phase, measurements/actions are requested from the discovered service instances and final values that answer the user query are returned to the IoT application. We proceed to present, in the next chapter, the implementation details of MobIoT along with extensive experiments to evaluate its performance.

Chapter 6

Implementation and Evaluation

In previous chapters, we described the theoretical aspects of our approach towards a scalable, interoperable, and mobile IoT. In this chapter, we present the implementation of MobIoT components, namely `Registration`, `Lookup`, `Composition & Estimation`, and `Access`. Those components were conceived and implemented as part of CHOReOS, a European project for service choreographies in the Future Internet. In addition to concretizing our contributions, the implementation enables us to assess the functionalities of our middleware, especially its semantic support for IoT application development, its usability in real world settings, and its support for automatic transparent composition. Most importantly, it allows us to demonstrate the scalability of our approaches through extensive evaluations, which we detail throughout the chapter.

6.1 MobIoT Prototype Implementation

MobIoT is implemented using Java 1.6 to create two complementary parts, that comprise all components presented in Chapter 3: the *MobIoT Mobile middleware* and the *MobIoT Web Service* illustrated in Figure 6.1. The MobIoT Mobile middleware is deployed on mobile Things (smartphones, tablets and laptops). It wraps the `Query` component, the `Registration` component, and the domain ontology. The MobIoT Web Service wraps the `Registry` component, the probabilistic `Lookup` component, the `Composition & Estimation` component, and both domain and de-

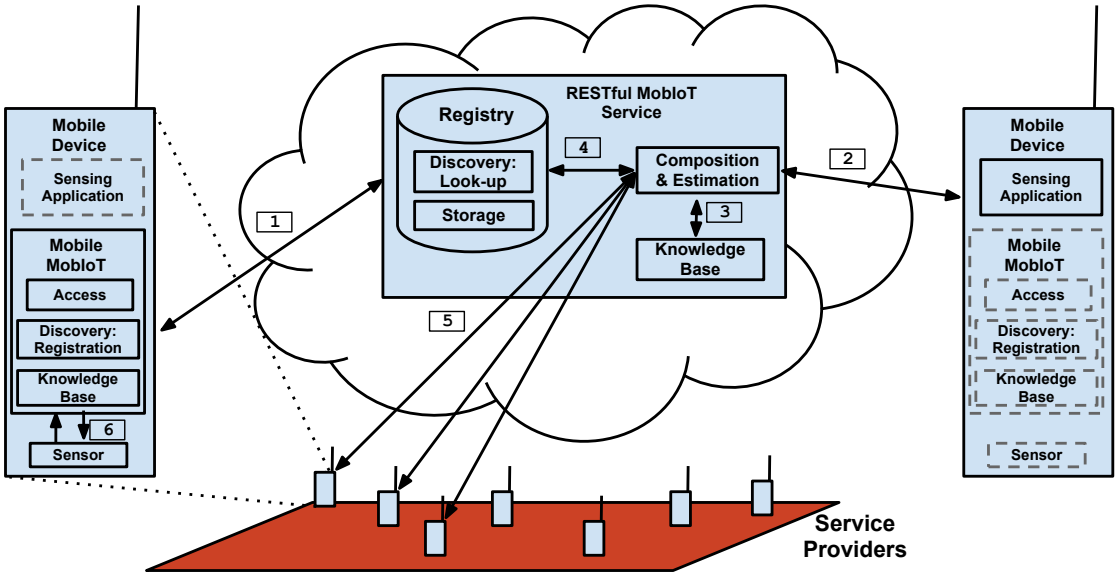

Figure 6.1. The MobIoT middleware implementation.

vice ontologies. Ontologies are accessed using Jena[1], an open source Semantic Web framework for Java, that provides a uniform interface to read/write data from/into RDF graphs and AndroJena[2] to enable access to RDF graphs on smartphones running Android . It is noteworthy to mention that we chose Java as a programming language given its robustness, portability, large suit of available libraries and the widespread community of Java developers. Nonetheless, we acknowledge the fact that Java and related frameworks employed while implementing our middleware solution are mostly suitable for Things with processing and computation capabilities such as smartphones. However, the solution can be ported to other Object Oriented languages (e.g., Python, C++) that are more suitable for constrained Things, which is especially feasible given the fact that the Registry provides a RESTful API that only requires Things to be able to communicate over HTTP. Our code is released as open source and links to the different components can be found at http://choreos.eu/bin/Documentation/IoTS_Middleware.

[1]Jena:http://jena.apache.org/.
[2]Androjenacode.google.com/p/androjena/?.

6.1.1 MobIoT Mobile Middleware

The MobIoT Mobile middleware consists of three components: i) the `Query` component (presented in Chapter 3), which handles queries for sensing/actuating services; ii) the `Registration` component (presented in Chapters 3 and 4), which determines whether or not Things can register their sensing/actuating services, and iii) the `Thing Access Middleware` (presented in Chapter 3), provided by our partners in CHOReOS to enable access to locally embedded sensors/actuators.

The `Query` component provides a uniform interface to enable developers to integrate sensing/actuating services in their mobile application logic. The developer needs only be concerned with specifying the physical concepts his application should measure/modify, the units of measurements his application requires, the data format his application supports, and the location of interest (either specified directly by the developer or by the end-user at run time). Consequently, when end-users require the services of an IoT application, their requests should be refined to filter out the application-specific logic. Afterwards, the resulting requests, matching the MobIoT queries, are forwarded to the Query component (Step 2 in Figure 6.1). The latter builds on an object oriented design where *Concepts*, *Locations* of interest and *Units* are designed as Java objects and extracted from the user query to be forwarded to the `Composition & Estimation` component. The class diagram with all Query objects is presented in Chapter 3.

The `Registration` component generates the decision to allow or prevent Things from registering their sensing/actuating services through the cooperation of three modules:

- A `RegistrationManager`, which implements the Probabilistic Registration Decision algorithm presented in Chapter 4. It receives registration requests from Things and informs them of the registration decision. To compute the registration decision, it requires the assistance of an `ExpansionGenerator` to identify the types of expansion services and a `ProbabilityEstimator` to compute the coverage probability. The component then uses the generated probability to produce the final registration decision (step 1 in Figure 6.1).

- The `ExpansionSetGenerator` identifies all possible expansion services based on information, provided by the `RegistrationManager`, on the initial concept

measured/modified by the registering service. Expansion information is extracted from a local copy of the domain ontology using SPARQL queries.

- The `ProbabilityEstimator` computes the coverage probability based on a mobility model, the estimated path of the Thing provided by the `RegistrationManager`, and information on the availability of similar and expansion service instances provided by the Registry. The computed probability is returned to the `RegistrationManager` to generate its final decision.

The `Thing Access Middleware` enables access to embedded sensors and actuators through the cooperation of two modules: `Thing Mediator` and the `Sensor_Actuator Driver`. A Thing-based service using local sensors/actuators can utilize the `Thing Mediator` for abstracting access to different sensors/actuators attached to the Thing. Individual vendors can contribute `Sensor_Actuator Drivers` to their sensors/actuators, which transparently bind with this mediator and provide access to data through Thing-based services. The `Sensor_Actuator Driver` and `Thing Mediator` are described in more detail bellow:

- `Sensor_Actuator Driver.` The `Sensor_Actuator Driver` defines the API that should be implemented by the sensor/actuator driver developer. It consists of four main modules: i) the `Sensor`, an Interface for sensor drivers, which are the objects that hold the actual logic behind each sensor's operation; ii) the `SensorInfo`, which contains metadata about a sensor; iii) the `Actuator`, an Interface for actuator drivers, which are the objects that hold the actual logic behind each actuator's operation; and iv) the `ActuatorInfo`, which contains metadata about an actuator.

- `Thing Mediator.` The `Thing Mediator` is meant to be accessed by the application developers and it consists of two modules: i) the `Mediator`, and ii) the `MediatorListener`.

  The `Mediator` provides functionality to discover all the sensors/actuators available on the Thing; discover the instance of a specific sensor/actuator, specified by its interface; and trigger an immediate sensing/action by all sensors/actuators.

  Through the `Mediator`, the application can perform basic discovery and sensing/actions with a single method call, abstracting away all the logical details

such as creating separate processes in the OS, synchronization, etc. All of these lower-level issues are handled by the `Mediator` in an OS-independent manner. In addition, all sensors/actuators are treated homogeneously under common APIs.

The `MediatorListener` is the abstract class that contains the functionalities to be implemented by the application developer to be able to retrieve the data from the sensors, or request actions from actuators on the Thing.

6.1.2 MobIoT Web Service

The MobIoT Web service consists of two components: i) the `Registry`, where sensing/actuating services are registered. The `Registry` integrates the probabilistic `Look-up` component (presented in Chapters 3 and 4), which retrieves the services to access; and ii) the `Composition & Estimation` component (presented in Chapters 3 and 5), which receives the query details from the `Query` component, identifies possible compositions, and generates the final answers to the user query.

The `Registry` component is a RESTful Web Service hosted on Apache Tomcat servers with Apache DERBY JDBC database as a backend store. In this setup, incoming requests are received by an Apache HTTP Web server. The `Registry` can be geographically distributed, per city for instance. To identify the proper Registry to contact upon registration and look-up, we intend to further explore geo-fencing, which sets virtual borders over geographical areas and enables the middleware to identify the right Registry based on the coordinates of the registering Thing or the location of interest in the user query and the boundaries of the area covered by each Registry. It is also possible to implement a look-up service that provides the registering Thing or IoT application with the address of the appropriate Registry to contact based on location information. The `Registry` address can also be specified by the developer himself, if known a priori, which is the case in our current prototype implementation. The `Registry` comprises three modules:

- A `Storage` module, which provides read/write functionalities on the data store. Functionalities include saving, deleting, updating, and reading the metadata of registered services.

- A `Look-up` module, which implements the probabilistic look-up algorithms pre-

sented in Chapter 4. It queries the backing store in the `Storage` module to acquire the addresses of services providing the required measurements/actions along with the locations of their hosts. It then determines the ones to select for access (step 4).

- A `RegistrationAssistant` module, which helps the `Registration` component acquire necessary information to generate the registration decision, precisely, the number of registered similar and expansion services at the location of interest, in addition to the spatial distribution of their hosts, through the `ALLFITDIST` method provided by MATLAB.[1] The method has the following signature `ALLFITDIST`$(DATA, PDF, SORTBY)$. `DATA` is the set of X and Y coordinates whose distribution is to be returned. The `SORTBY` parameter is optional. It is used to specify the preferred metric for the goodness of fit. To execute the method and call MATLAB from Java we used `matlabcontrol` which provides a uniform Java interface to evaluate MATLAB methods. Additionally, since Things do not register their services indefinitely, each Thing sends a registration duration as part of the service metadata. If the duration expires, the Thing can inform the `RegistrationAssistant` to keep its service alive. Otherwise the `RegistrationAssistant` removes the service through the delete functionality of the `Storage` component.

Finally, the `Composition & Estimation` component answers users' queries through the cooperation of four modules:

- A `CompositionManager` module that implements the Expansion, Mapping, and Execution algorithms in Chapter 5. It generates expansion concepts and executes expansion formulas. Additionally, it interacts closely with the `Lookup` component to find the addresses of service instances to access.

- The `SPARQLManager` module handles the SPARQL-related tasks. It queries the MobIoT ontologies to extract the expansion formulas and unit conversion formulas (step 3). When formulas are extracted from the ontologies, they are extracted as strings; to convert them to actual mathematical formulas we use Expr4j[2], an expression calculation engine for Java.

[1] MATLAB:http://www.matlab.com.
[2] Expr4j:http://expr4j.sourceforge.net/.

- The `AccessGenerator` module handles the remote access to sensing/actuating services hosted on mobile Things (step 5). Services are implemented, by service developers, as REST services with a unique address each. The remote access component interacts directly with the `Thing Access Middleware` to interrogate sensors/actuators abstracted by the services (in mobile MobIoT).
- The `DataFusionGenerator` module handles all fusion/aggregation tasks to combine measurements provided by different services. It assists the `CompositionManager` module in generating the final answers to user queries. It implements the `PairWiseComputation` algorithm presented in Chapter 5. We extended it to implement an average and count fusion functions. It can be extended to integrate any fusion function.

The service developer and the IoT application developer are only required to implement one method each, provided by the `RegistrationManager` and `QueryManager`, respectively. All computations pertaining to probabilistic discovery, composition and access are handled internally by MobIoT. The methods to implement are: `executeRegistrationQuery()` (Listing 6.1) and `getData()` (Listing 6.2). We illustrate MobIoT usability and the method implementations through a proof-of-concept application in the following section.

Listing 6.1. The `executeRegistrationQuery()` method signature.

```
public boolean executeRegistrationQuery(String deviceId,
    String serviceName, String serviceType, double accuracy,
    String type, String physicalConcept, double range,
    String address, String dataType,
    double locationCoordinateX, double locationCoordinateY,
    ArrayList<Location> pathLocations, String unit);
```

Listing 6.2. The `getData()` method signature.

```
public SensorData getData(Query myQuery, String requestedDataType,
    String fusionFunction);
```

6.1.3 DynaRoute Application

We assess the usability of MobIoT through DynaRoute, an application developed by our industrial partners within CHOReOS [CHOReOS consortium, 2011c], that uti-

lizes smartphone-carried sensors to provide Thing-based services along with business services. DynaRoute is designed to support dynamic personal organizer features. The main storyline is about a travelling citizen, called Collista, who follows a predefined itinerary. The application imports external events on-the-fly, provides utilities (like Point Of Internet liveliness, traffic service, taxi appointment, friend proximity notification), and utilizes them to organize Collista's schedule. We focus on Thing-based services that can be exploited by the application. While visiting Rome, Collista wishes to take the Colosseum tourist tour, with a preference to attend a low-crowded program. DynaRoute can adapt her schedule, based on information being provided by a "Point-Of-Interest (POI) liveliness" service and dynamically modify the initial itinerary. The POI service exploits noise measurement information provided by Things in the landmark to visit.

In more detail, the application, illustrated in Figure 6.2, can be used to:

- Register the sensing services hosted on a mobile Thing; and
- Acquire noise level information before visiting a landmark.

The application, deployed on mobile Things, has the following components:

- A *GUI* allowing users to choose between registering (activating) their sensing services at their current location, or requesting noise estimations at a remote location of interest (e.g., the Colosseum in Rome) (Figure 6.2); and
- The *MobIoT middleware*, through which the composition, probabilistic discovery and access to services take place.

Figure 6.2 shows a map where the user specifies the location of interest by tapping on the screen, the noise level will be checked at this location. The user can choose to register the noise measurement service by checking the contribute box. If the decision is positive, the upload arrow is enabled.

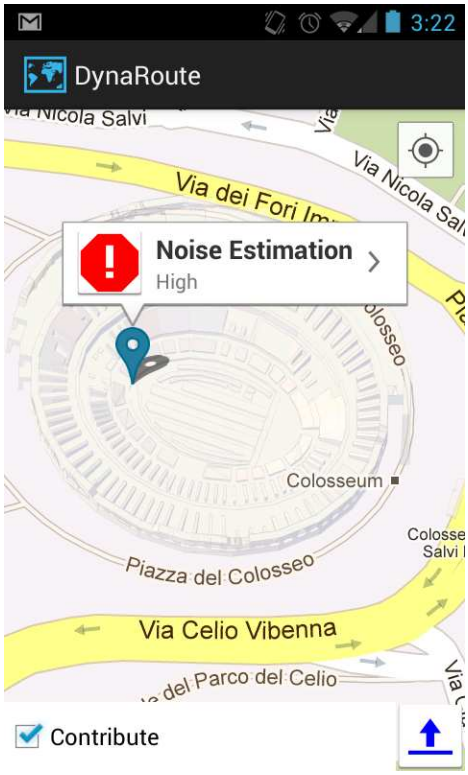

Figure 6.2. The GUI of DynaRoute POI service.

Listing 6.3. The Android `executeRegistrationQuery()` method signature.

```
//called from AsyncTask launched by OnCheckedChangeListener
protected boolean tryToRegister(double accuracyVal, double rangeVal,
        long initiLocationLongVal, long initLocationLatVal) {
  ThingsRegistrationManagerImpl cm = new ThingsRegistrationManagerImpl();
  boolean decision = cm.executeRegistrationQuery(
      myDeviceId, ''getnoiselevel'', ''noiselevel_service'', accuracyVal,
      ''noiselevel_sensor'', ''noiselevel'', rangeVal,
      ''http://Things.vtrip.net:8080/proxy/123456/getnoiselevel'',
      Constants.DOUBLENOISE, initLocationLongVal, intLocationLatVal,
      myPath, ''dB'');
  return decision;}
```

While writing the code for the noise estimation application, the application developer is required to call the `executeRegistrationQuery()` method. The method enables the developer to request a registration decision for the noise measurement service at his location as shown in Listing 6.3. The developer is also required to invoke the `getData()` method shown in Listing 6.4.

Listing 6.4. The Android `getData()` method signature.

```
//called from AsyncTask launched by OnMapViewLongClickListener
protected SensorData getNoiseValue(String locName, long latitude,
  long longitude,String fusionFunction) {
  ThingsQueryManagerImpl qm = new ThingsQueryManagerImpl();
  Query myQuery = new Query(
          new Selector(
            new Concept(''noiselevel'')),
            new Constraint(
              new Where(''noiselevel.unit.equals= dB'')),
            new Location(locName, latitude, longitude));
  return qm.getData(myQuery, Constants.DOUBLENOISE');}
```

Beyond those two methods, all computations, interactions and method calls are performed internally by MobIoT using the knowledge specified in our ontologies, which in this scenario, was sufficient and needed no extension. However, given that we work with extensible RDF graphs, any additional information that needs to be modeled can easily be integrated. By exploiting the knowledge in the ontologies, and although not explicitly specified by the developer, nor requested by the user, any necessary compositions that can be exploited, as presented in Chapter 5, to compute the noise information, are automatically identified, and executed transparently. We proceed in the following to present the experimentation setup and evaluations we performed to assess the benefits of our contributions.

6.2 Experimental Evaluation

We aim with our experiments to demonstrate the validity of our probabilistic registration and look-up approaches and to assess the scalability of MobIoT. The validity of the approaches is evaluated based on the coverage provided by Things hosting registered services or Things hosting retrieved services (to access). The scalability is evaluated based on the times needed for MobIoT functionalities to complete and

the benefits they introduce as compared to *full* registration and look-up approaches. In full registration mode, all Things are allowed to register their services. In full look-up mode, all registered services that match a query are selected for access. In other terms, the *full* approach corresponds to traditional SOA-based registration and look-up. The evaluations of the Composition approach are part of our future work.

For evaluation purposes, we implemented a version of MobIoT, that executes full registration and full look-up. We also implemented a set of simulation components presented below.

6.2.1 Simulation Environment

Our simulation environment comprises five components: the `TrafficGenerator` in charge of simulating mobility of Things; `Mobility Traces` comprising timestamped displacements of Things; `ThingLauncher`, which generates virtual `Things`; `Thing`, which emulates a Thing; and the `QueryLauncher`, in charge of simulating user queries for Thing-based services.

- `TrafficGenerator`. The `TrafficGenerator` generates traffic based on data from a mobility trace, and simulates the mobility of each Thing, represented by a path (list of timestamped coordinates) to follow over time. It generates a launch trace, to be used by the `ThingLauncher`, specifying the times at which virtual `Things` appear and request registration, as well as a query trace to be used by the `QueryLauncher` presenting the times at which virtual user queries are forwarded to MobIoT. The `TrafficGenerator` parses the mobility trace into a time map with timestamps as the keys and Thing IDs with their coordinates at that timestamp as values. Based on information from the map, and a movement rate (λ) provided as input, the `TrafficGenerator` scans the trace to find timestamps that satisfy λ. The `TrafficGenerator` then generates a Poisson arrival process (for Things to show up and request registration of their services). At each Poisson arrival time, represented by a timestamp, the `TrafficGenerator` scans the time map for Things with a matching timestamp. If no result is found, the `TrafficGenerator` executes the interpolation method presented in Chapter 4 to estimate the position of the Thing at the desired timestamp. It then generates, for each Thing, a sequence of timestamps based on λ, and corresponding coordinates based on the interpolation method. This

list represents the path the Thing will follow. Finally the arrival and path information are forwarded to the `ThingLauncher`, to create virtual `Things`, with the following format:

```
Timestamp,Device ID,Longitude,Latitude,Path
```

The `TrafficGenerator`, uses the lowest and highest timestamps of the generated mobility trace and creates a query trace that serves as an input for the `QueryLauncher`. It specifies the query arrival times (between the lowest and highest stamps above), which similar to the Thing arrival times, follow a Poisson process. Each query tuple in the trace has the following format:

```
Timestamp,Concept,Location
```

- `Mobility Traces`. To assess the validity of our solution, it is crucial to rely on real mobility information. Towards that purpose, we use a dataset that provides real mobility traces —in the form of (`taxi id, date, time, longitude, latitude`)— for $10,000$ GPS-equipped taxis in Beijing [Yuan et al., 2011]. We set the length of the area A to 1136.2 km and breadth to 3002.4 km based on the area covered by the mobility traces. We computed the average velocity of taxis (8 km/hr) and set the sensing range of virtual microphones they carry to be 10 m. The value of the sensing range was scaled from the average sensing range of microphones in mobile phones (5 m, a value that we estimated empirically) carried by pedestrians who walk at an average speed of 4.5 km/h to better suit the average velocity (8 km/hr) in the Beijing trace.

  To assess MobIoT's ability to handle registration and look-up requests over a large number of Things, we measure the respective response times for both *full* and *probabilistic* registration and look-up, using the Beijing mobility trace, with no concurrent request. We then simulate concurrent requests (up to $1,000$ requests per second) and use a larger synthetic mobility trace generated with SUMO [Behrisch et al., 2011]. The latter generates individual mobility traces based on real mobility data in the city of Cologne [Uppoor et al., 2013]. Our choice to assess our work with the Beijing trace then the SUMO trace is due to the fact that our first priority is to evaluate the approach's validity with a real

mobility dataset. However, the Beijing dataset provides us with 10000 traces while our second priority is to evaluate the scalability of our middleware, which requires larger datasets. Consequently, as we were unable to find larger real mobility traces, we switch to the SUMO dataset, which provides synthetic but larger traces (15000). It should be noted that it is also possible to integrate any other mobility trace, with larger traces, as long at they comply with the required format presented above.

- `ThingLauncher and Things.` The `ThingLauncher` creates, launches, and stops virtual `Things` based on the arrival times in the launch trace generated by the `TrafficGenerator`. Once the network traffic is generated, virtual `Things` are created to emulate real-world mobile Things running the `Registration` component and hosting services to select and access. Each virtual `Thing` initiates a service registration request and measures its own response time. Each `Thing` is an independent single Java thread. If a `Thing` is permitted to register its service, the registration remains valid until the end of the evaluation.

- `QueryLauncher`. The `QueryLauncher` generates look-up and access queries based on the query trace generated by the `TrafficGenerator`. It is able to launch queries concurrently or sequentially and measure their response time. The `QueryLauncher` produces sensing queries to measure a particular physical concept (e.g., temperature) at a specific geographical location (either an area or a point). It is possible for the queries to require atomic services or composite services, depending on the physical concept. At each query arrival time, the `QueryLauncher` creates a thread. In each thread, a query is generated with the concept and location information from the trace, to be forwarded to the `QueryManager` component.

We deployed the simulation system in a computer cluster, consisting of 40 machines with Scientific Linux 5.5, Intel Xeon $X5650$ dual processors and 48 GB RAM each. The system's elements were distributed inside the cluster to maximize resource utilization. The deployed architecture is depicted in Figure 6.3. When a virtual `Thing` is allowed to register its service, it launches a RESTlet[1], allowing REST access to its service with "access" results being random double values. To evaluate the

[1]RESTlet:`http://restlet.org/.`

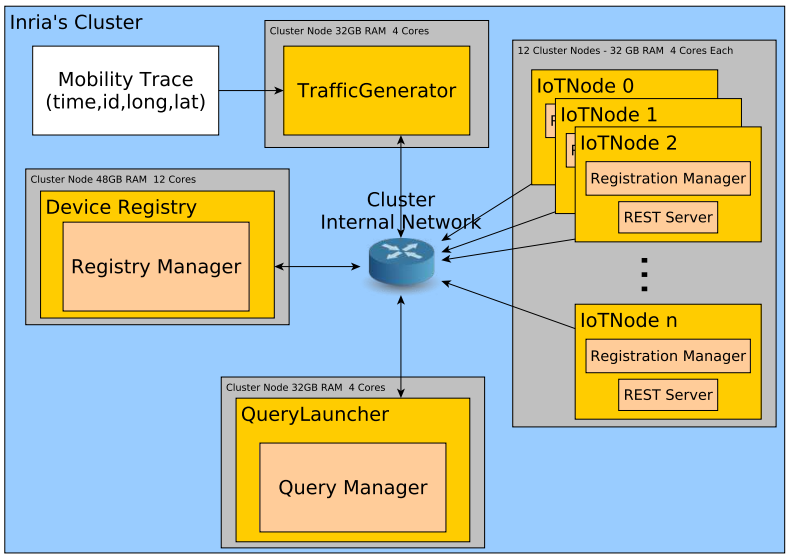

Figure 6.3. Deployment used for performing large scale evaluations.

time needed to compute a registration decision by a mobile Thing, we used Android Samsung Galaxy S3 smartphones with 1GB RAM. Given the large scale we target, a real testbed with thousands of mobile phones was not an option.

6.2.2 Assessment Metrics

We conducted several experiments to evaluate the validity and scalability of MobIoT in general, and the probabilistic registration and probabilistic look-up approaches in particular. Specifically, we are interested in three metrics:

1. The **coverage quality** measures the percentage of the area that is covered. This criterion is essential to assess the validity of our probabilistic approaches and can be presented by one of the following parameters:

   - **Maximum Possible Coverage (MPC)**: Percentage of the area of interest A that is covered by the complete set of Things currently present within A. This value gives us the upper bound of possible coverage.
   - **Actual Area Coverage (AAC)**: Percentage of the MPC of A covered by a subset of Things. During registration, the subset contains the Things that register their services. During look-up the subset contains the Things that are selected to provide their services.

Acronym	Meaning
MPC	Maximum Possible Coverage
AAC	Actual Area Coverage
TLW	Truncated Lévy Walk
RW	Random Walk

Table 6.1. Acronyms used in this chapter.

- **Required Path Coverage**: The desired percentage of the area to cover by each registering Thing throughout its path.

2. The **size of the selected subset** illustrates the benefits of the probabilistic approaches as they decrease the number of participating Things.

3. The **response times** measure the time needed to execute concurrent registration requests or to answer concurrent user queries. They are essential to evaluate the scalability of MobIoT.

Additionally, the probabilistic registration approach imposes that we evaluate the ability of our middleware to handle different mobility models and the gains introduced by choosing TLW over the deterministic registration approach. Acronyms used in this chapter are presented in Table 6.1.

6.2.3 Registration Evaluation Results

As mentioned in Chapter 4, the probability computations underwent a set of simplifications, notably when substituting the coverage circles with squares. We first present the effects of those modifications, followed by an illustration of the validity and viability of our registration approach, based on the Beijing traces, in terms of three criteria presented below:

1. How coverage varies as we shift from full registration to probabilistic registration. Full registration refers to the case where all Things can register their services, i.e., equivalent to SOA registration.

2. The appropriateness of TLW, to which our computations are directly related. Consequently it is paramount to compare it to other well established mobility models.

3. the benefits of the probabilistic computations with respect to the deterministic version presented in Chapter 4, in terms of the final coverage and final number of registered services.

In this set of experiments, the registration simulation executes sequentially, in the sense that when a Thing shows up in the network it directly requests registration and no two Things show up at the same time, therefore no concurrent requests take place.

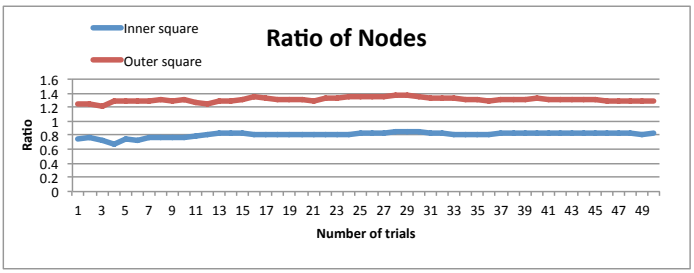

Figure 6.4. The ratio of the number of Things in the inner and outer squares to the number of Things in the sensing circle after five steps.

Simplification effect. To evaluate the effect of replacing the circle $\zeta_{l_i,r}$ by the largest square within $\zeta_{l_i,r}$, as discussed in Chapter 4, we computed the ratio of the number of Things within the largest inner and smallest outer squares to $\zeta_{l_i,r}$ after the Things take 5 and 10 steps. We generated Lévy Walks (with a scale factor of 10 and 1 for the step-size and pause-duration respectively [Sadiq and Kumar, 2011], maximum allowed pause time $\tau_p = 1$ hr, and maximum allowed displacement length $\tau_\xi = 5$ km) for 100 Things and repeated this process 50 times. We find that the number stabilizes around a value of 0.8 for the inner square and 1.2 for the outer square (Figure 6.4). We use this inner square to circle ratio, (0.8), as a scaling factor for n_i as it is better to be conservative and assume fewer Things are covering the Thing's path, thus leading to fewer false positives.

Coverage quality. To evaluate how *coverage* varies as we shift from full to probabilistic registration, we computed the MPC while the 10,000 Things show up sequentially. The MPC when all 10,000 Things register their services is 0.003% of the total area, which is very low, but we were unable to find larger real datasets. This is

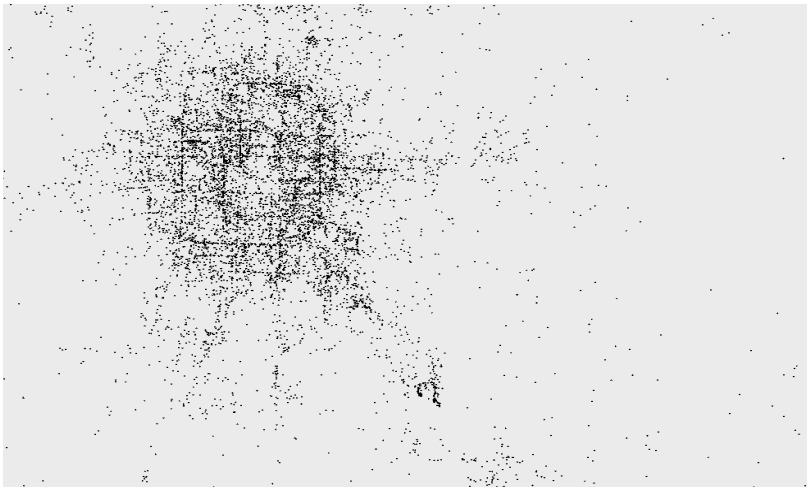

Figure 6.5. A plot of the locations of the 10,000 Things in Beijing.

depicted in Figure 6.5 where the black dots represent the MPC of 10,000 Things in Beijing.

Figure 6.6(a), shows the AAC for 0.8 and 0.6 required thresholds, along with the registration curves for each of the thresholds as Things show up. Note that the AAC values in Figure 6.6(a) are misleading. In spite of being high, they are only a percentage of the MPC, which is very low. Moreover, although, presented at a very small visual scale, Figure 6.5 reflects the sparsity of the Things and the large portion of the area left uncovered. Consequently, due to this low density of Things and their sparse distribution, they will rarely cross paths. This explains their continuing registration, depicted by the linearly increasing registration curves for 0.6 and 0.8 thresholds in Figure 6.6(a).

We remind readers that AAC is a percentage of the MPC and not A. This is also the reason why, at the beginning of the evaluation, AAC curves start high and then decrease until they stabilize. More precisely, when the first Thing shows up, it will register, regardless of the threshold value, leading to: $AAC = MPC$ for a subset size of 1. Similar results will occur for the second Thing and so on until the thresholds start to affect the registration decisions (when the sensing areas of Things start to overlap).

To better show the benefits of our approach, which are most relevant in highly dense networks, it is important to reach a MPC of 100% of the area before all 10,000 Things register. However, instead of adding phantom traces to increase the MPC

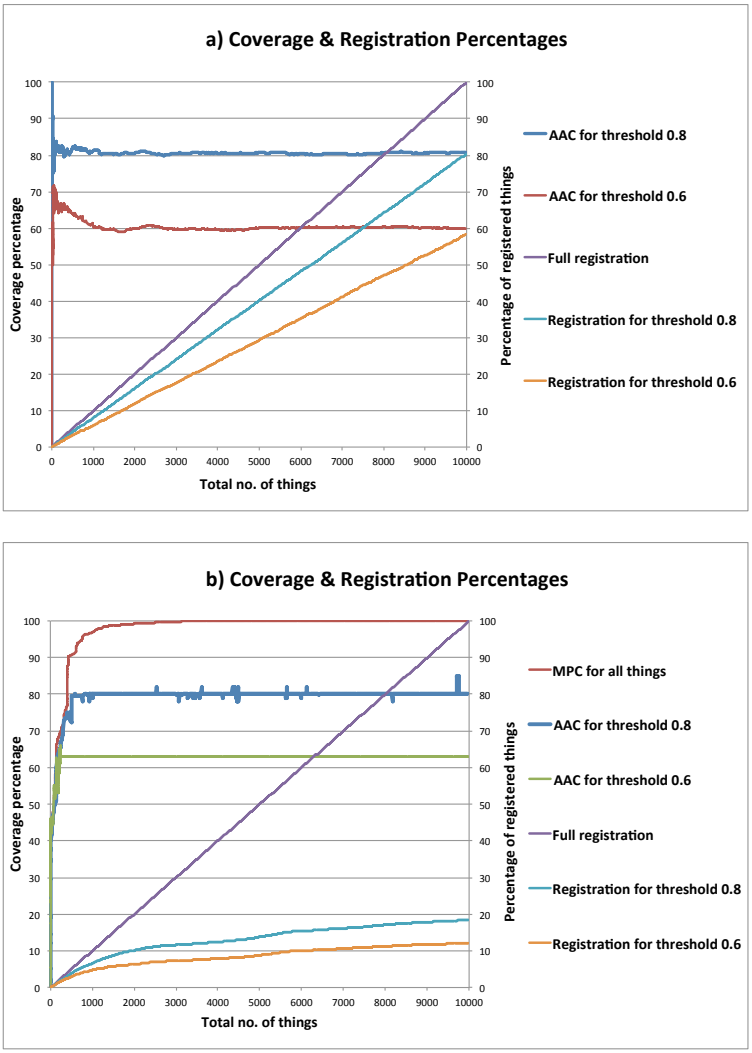

Figure 6.6. The resulting coverage and registration percentages as the required coverage threshold decreases from 1 to 0.6 a) for a radius of 10 m b) for a radius of 10 km.

of the set of Things, we decided to increase the sensing range. Since our goal was only to find a suitable scenario with the above properties, we decided to graphically and rapidly render the possible options, and chose the first to match our constraints by visual confirmation. Results showed that a sensing range of 10 kilometers best addresses the sparsity issue between Things in this dataset, spread over a very large area. We are aware that this modification is not an accurate reflection of the real world, but in the absence of better publicly available large scale real mobility traces,

it was necessary to evaluate if our probabilistic registration approach decreases the number of registered services while still maintaining the requested coverage threshold. In fact, larger ranges may indeed be applicable for sensing, say, air quality. The resulting MPC is shown in Figure 6.6(b), which illustrates how our approach successfully prevents Things from registering. The registration curves for thresholds of 0.8 and 0.6 show that the resulting subset sizes of Things that register are less than 2000, while still meeting the coverage requirements, giving us, at least, 80% gain.

What can be concluded from the set of experiments presented above is that our probabilistic registration approach functions correctly in both sparse and dense infrastructures. In the former, although our approach decreases the registration rate, it enables Things to register their services throughout the whole experiments as they remain needed. However, the approach is most beneficial in dense networks where it strongly decreases the registration rate, which stabilizes around 20% of the complete set size while still satisfying area coverage requirements.

Mobility Model evaluation. We evaluate in this section the ability of our registration approach to support various mobility models, and illustrate the correlation between the quality of the results and the chosen model. To that end, we compared the TLW-based results to Random-Walk (RW) based results, by substituting our probability computations with RW-based computations [Bai and Helmy, 2004, Roy, 2011]. Note that RW is also a commonly adopted model for estimating displacements of mobile entities due to its simplicity. The Random Walk model has the following characteristics:

- Things change their speed and direction after each time interval, and there is no pause time between changes.
- Each Thing can select a speed within $[v_{min}, v_{max}]$ following a uniform distribution or gaussian distribution at every new time interval. We consider an average constant speed v that is similar for all mobile Things.
- Each Thing selects a new direction $\theta(t)$ uniformly between $[0, 2\pi)$.
- The model is memoryless and does not keep knowledge of past directions and all velocity values are independent.

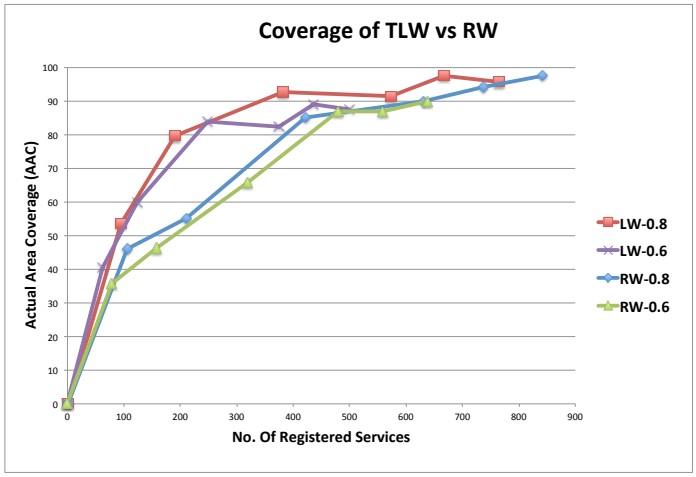

Figure 6.7. Coverage/registration percentage for TLW-based and RW-based registration.

In addition to the elements presented above, and given the characteristics of a Random Walk, we assume that the displacement angles and lengths are independent. Each displacement D_i is represented as $D_i = \xi_i * e^{i\theta_i}$ [Stadje, 1987]. Things are initially deployed based on a 2D Poisson process with characteristics presented below and the deployment maintains a 2-D Poisson distribution after fixed movement time intervals with the same rate λ [Serfozo, 1999, Wimalajeewa and Jayaweera, 2010].

- n (the total number of Things) is a Poisson variable.
- The distance between the nodes is an exponential variable.
- $\lambda = n/A$ is the density of the Poisson distribution (A is the total area).

Therefore, P_{cov} in Chapter 4, Equation 4.17, is substituted with:

$$P_{cov} = (1 - e^{(-\lambda\pi r^2)})^{|L|} \tag{6.1}$$

with L being the total number of locations.

We repeated the same set of evaluations presented in the previous section to compare RW- and TLW- based computations. Results show that both models decrease the registration of Things while satisfying coverage requirements. However TLW outperforms RW as it leads to fewer Things participating (Figure 6.7). The figure shows the AAC of TLW- and RW-based registrations and the number of registered services.

As can be seen from the curves, the former leads to 760 Things registering their services while the latter leads to 840 for a required threshold of 0.8 and 499 Things versus 639 for a threshold of 0.6. Therefore, TLW leads to at least 10% fewer Things registering their services. Given the large scale we anticipate, this is an important gain. It can also be seen that TLW-based registration reaches the required coverage threshold faster.

Consequently, our evaluations prove that the probabilistic registration approach is not restricted to one model alone. Any appropriate model can be easily plugged in, as illustrated by the straightforward substitution of the TWL model with the RW model. Moreover, the results of our evaluations —where TLW outperforms RW— illustrate the correlation between the quality of the registration decision and the mobility model exploited by the middleware demonstrating that the latter does not perform blindly regardless of the employed model.

Probabilistic versus deterministic registration. We also compared our probabilistic registration approach to the deterministic registration approach presented in Chapter 4 (with a similar setup as above). Our goal is to verify, based on real mobility traces, if indeed the probabilistic approach requires less computation efforts while satisfying the coverage threshold. Unlike the probabilistic registration where the threshold varies based on a utility function (Chapter 4), the deterministic registration decision is pessimistic. The coverage threshold remains unchanged regardless of how many neighboring Things are around the new Thing. Moreover, as long as Things with registered services will not, for certain, cross path with the new mobile Thing and provide sufficient coverage, the latter is requested to register its service. We use the term certain loosely as it is still based on path predictions. Although the deterministic version of the solution leads to fewer Things registering in general, it registers almost 30% more Things than the probabilistic version.

In the deterministic approach, it takes up to $1,000$ ms (Figure 6.8) and up to 800 ms (Figure 6.9) for a decision to register and not register, respectively, to be taken while the $10,000$ Things request registration sequentially. Figure 6.8 shows the time needed to generate a positive registration decision and the number of already registered services. Figure 6.9 shows the time needed to generate a negative registration decision and the number of services already prevented from registering. The computation time for the deterministic approach increases with more Things having

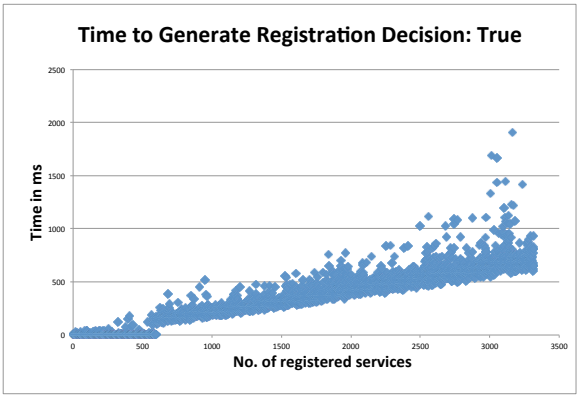

Figure 6.8. Time needed by the Registry to allow registration.

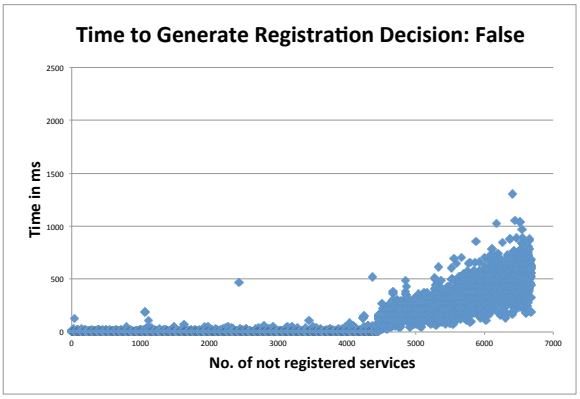

Figure 6.9. Time needed by the Registry to prevent registration.

registered their services because the Registry has to iterate over a larger number of services before it finds enough coverage of the current Thing's path. If so, it stops and determines that coverage is sufficient. It is also possible for the Registry to go through the whole list of services and determine that coverage is insufficient, in which case the registration decision is positive. This is especially highlighted in Figure 6.9 where the time of the decision to not register increases strongly when more services start to register. The time increase can be noticed after 4500 Things have been prevented from registering their services. Figure 6.10, presenting the number of Things allowed and prevented from registering their services as they show up, and the times needed for the decision to be generated, proves our claim. It shows that the number of registering services starts to increase strongly after 5000 Things have showed up (among which only 500 Things have registered their services). The increasing regis-

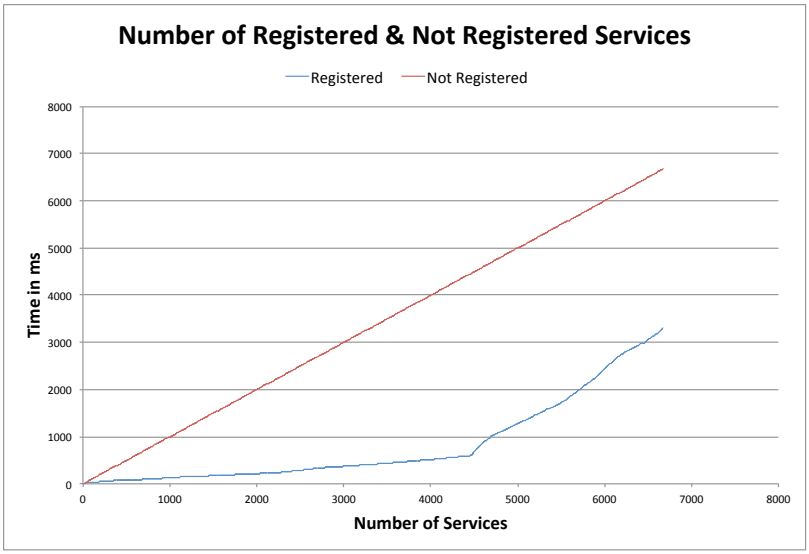

Figure 6.10. The number of Things allowed and prevented from registering their services.

tration trend, can also be seen in Figure 6.8 after almost 500 Things have registered their services. Contrary to the expectations for the registration rate to decrease, the new Things are appearing in less crowded areas and even though many Things have already registered, they do not sufficiently cover the new comers and therefore the Registry allows them to register.

In the probabilistic approach, the time needed to compute the decision probabilistically on an Android phone is constant and on average equal to 151.5 ms (Figure 6.11) regardless of the number of Things that registered their services already.

Unlike the deterministic registration approach which requires computation times that are proportional to the number of registered services, its probabilistic counterpart requires more or less constant computation times that are independent of the size of the registered set, which is a sign of better scalability. Thus, it can be concluded that our probabilistic registration approach is indeed an appropriate solution to handle the anticipated IoT scale. Moreover, the fact that computations on an Android phone require on average 150 ms for the registration decision to be computed illustrates that the probabilistic registration approach is not too complex to be computed locally on smart Things.

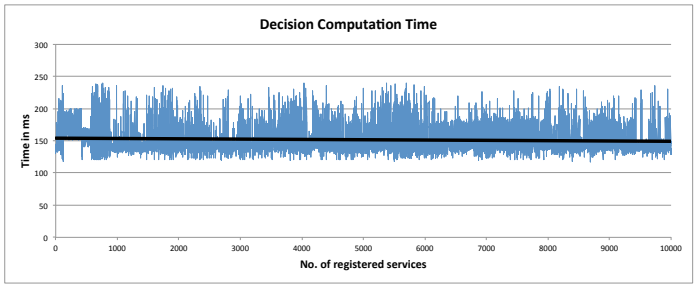

Figure 6.11. The time needed by an Android phone to compute the registration decision.

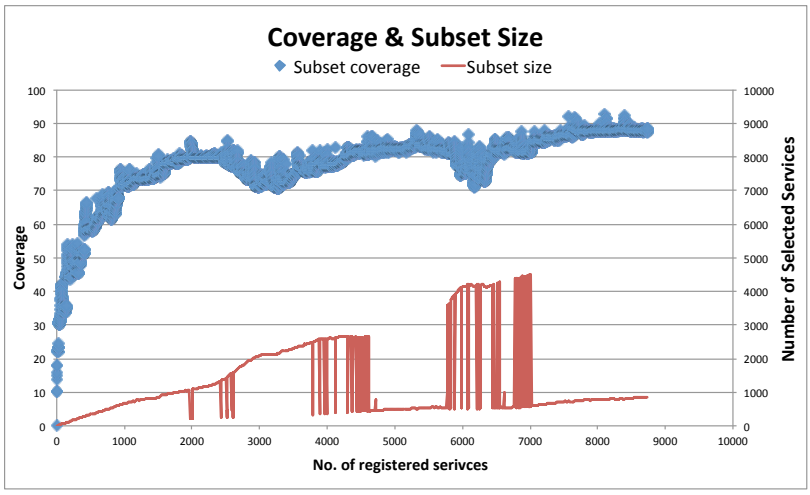

Figure 6.12. Coverage and set size of probabilistically selected subset of registered services.

6.2.4 Look-up Evaluation Results

Similar to probabilistic registration evaluations, the validity of the probabilistic look-up approach should be evaluated with respect to coverage and resulting subset size. For this set, we chose to evaluate look-up with full registration to keep the available set of services as large as possible. We also set the area of interest to be A. In this set of experiments, the look-up simulation executes sequentially, i.e., when a Thing shows up in the network it directly requests registration, followed by a user query being forwarded to MobIoT. No two Things show up at the same time and no two queries are created at the same time, therefore no concurrent request takes place.

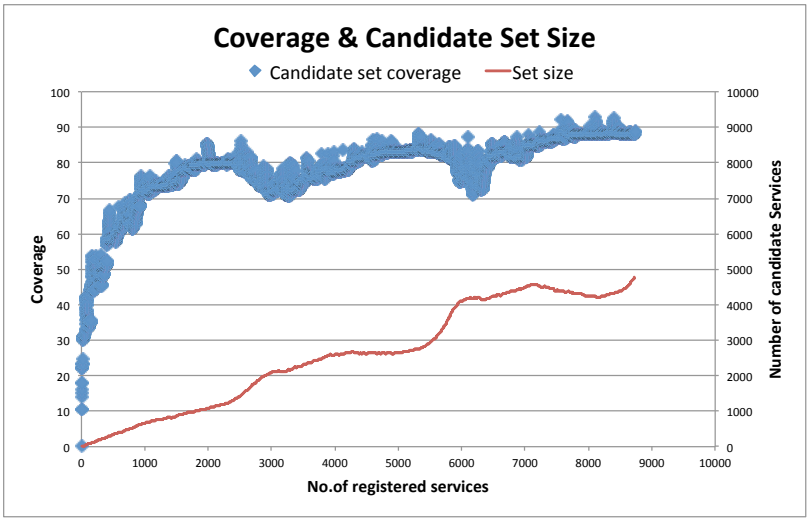

Figure 6.13. The coverage and set size of the candidate set of services.

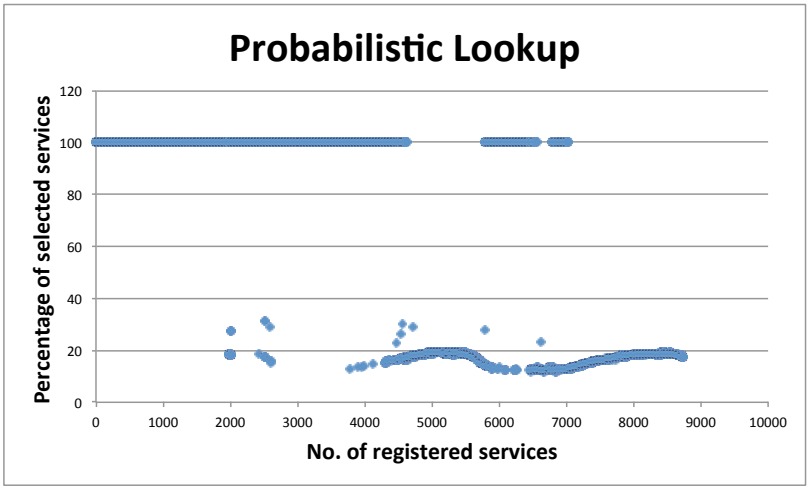

Figure 6.14. Subset selection by the uniform distribution-based look-up.

Coverage quality. The experiments were conducted for subset selections are based on an 80% required coverage and a uniform spatial distribution. The outcome of our experiments is presented in Figures 6.12 - 6.14. Candidate services are services that match the user query, estimated (through interpolation) to be at the location of interest depicted the complete set of appropriate services. It should be noted that we evaluated look-up for atomic services only (with no composition). The coverage provided by the probabilistically selected subset of services along with the corresponding subset size are shown in Figure 6.12 and the coverage provided by the set of candidate

services along with the corresponding set size are shown in Figure 6.13. In fact, even if the full look-up was to be adopted, not all registered services would have been selected, but only the ones estimated to be at the location of interest, i.e, the ones in the candidate set. When coverage is not satisfied, the complete set of candidate services should be selected, which explains why, in many locations in the graph of Figure 6.14 —which shows the percentage of services selected probabilistically from the set of candidate services— the selected services curve is 100% of the candidate services set. When coverage becomes sufficient, the subset selection process is initiated and the selected subset size decreases strongly to less than 20% of the candidate services set size. This information is depicted in Figure 6.14, where the lower pattern of the curve in the graph is around 20% of the complete candidate set. Consequently, 80% fewer services are selected while not hindering the coverage quality. This information, is also illustrated in Figure 6.12 by comparing the lower pattern of the selected number of services curve, when coverage is satisfied, to the upper pattern of the same curve when coverage is less than required, leading the size to increase fourfold. Moreover, Figures 6.12 and 6.13 show that the area coverage of the candidate set and the coverage of the probabilistically selected subset are almost identical although the size of the latter is much smaller than the size of the former.

The experiments in this section validate our claim that, in dense networks, it is sufficient to select some of the available services, rather than all of them. Furthermore, they illustrate the benefits introduced by employing probabilistic look-up as the selection rate decreases fourfold when coverage requirements are satisfied. The decreased selection rate illustrates in turn the scalability of our approach, which, in spite of the large number of available Things, can successfully employ the needed ones only without burdening the system, exploiting our middleware, with redundant communications and access to services that provide no or little additional information.

In addition to evaluating each approach separately, it is crucial to evaluate the global performance of MobIoT based on the Query Response times (QRT)s of the probabilistic approaches. We assess the results by comparing them to QRTs of full registration, look-up and access, presented in the following.

6.2.5 MobIoT Evaluation Results

In order to assess the scalability of MobIoT, we conducted a set of experiments that can be divided into two categories: i) the first set evaluates the performance of MobIoT's probabilistic registration, and ii) the second set compares MobIoT's probabilistic look-up and access to full look-up and access approaches. The performances are evaluated with respect to one metric: the Query Response Time (QRT). The QRT for registration is the time taken for MobIoT to compute the registration decision and if allowed, register the Thing's service. The QRT for look-up and access is the time needed for MobIoT to execute a user query, including finding and accessing services (with no composition). This set of experiments requires concurrent requests. For both the registration and user-query execution experiments, we varied the input rate to MobIoT from 100 requests per second up to $1,000$ requests per second. The results show averages from 20 runs. As input, we used our `TrafficGenerator` to generate traffic from the most populated sections of the original Cologne trace [Uppoor et al., 2013].

Registration scalability. In the first set of experiments, we measured the response times for probabilistic registration with a coverage threshold of 80%. When the registration goes through the full approach, response times are expectedly lower than its probabilistic counterpart since no calculations are involved and the Thing is always allowed to register its service. Nonetheless, the overhead for the probabilistic registration is acceptable as the response times are still reasonably fast and this is a fair price to pay for the faster look-up and access times shown in the next evaluations.

By looking deeper at the probabilistic registration, especially the *Find* operation, depicted in Figure 6.15, we conclude that it is the bottleneck of the probabilistic approach. The operation presents the phase where the registry determines the number of similar and expansion services at the location of interest. Note that the response time of the other registration operations, namely probability computation and actual registration operations, is somewhat stable with respect to the input rate (around 200 ms). The probability computation represents the computation of the coverage probability to generate the registration decision. The registration operation represents the attempts to register a Thing in the Registry assuming a positive decision was generated. The response time of the *Find* operation grows as the input rate

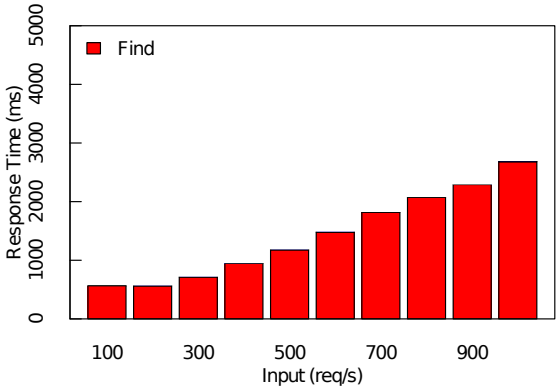

Figure 6.15. Time analysis of the *Find* operation for different input rates.

grows and as the number of registered services grows. This behavior is mainly due to two different reasons. First, as the input rate grows, the number of Things concurrently accessing the Registry grows as well, which increases the response times. Note that this is not a simple INSERT operation but a READ operation that requires the Registry to perform several computations before returning the result to each request. Second, as the number of already registered services grows, the time needed for retrieving the information from the Registry backend store also grows. However, this delay is dependent solely on the search algorithms implemented by the chosen database technology.

To measure networking overhead in a realistic mobile environment, we evaluated the time needed to register a service if requested by an Android smartphone after a positive registration decision is taken, i.e., the time needed to establish a connection to the registry and save the service metadata in the backend store, for an individual request. The former takes, on average, 120 ms, while the latter takes 20 ms regardless of the number of already registered services. The results are averaged over 100 runs using Wi-Fi connection.

Look-up and access scalability. We also measured the response times for concurrent queries with both full registration, look-up and access and probabilistic registration, look-up and access with a coverage threshold of 80%. The benefits of our probabilistic look-up are clearly illustrated in Figure 6.16 depicting the distribution of response times for both approaches with $1,000$ concurrent user queries. Figure 6.16(a) shows that most occurrences of QRT for the full approach are between 0 and 10 sec-

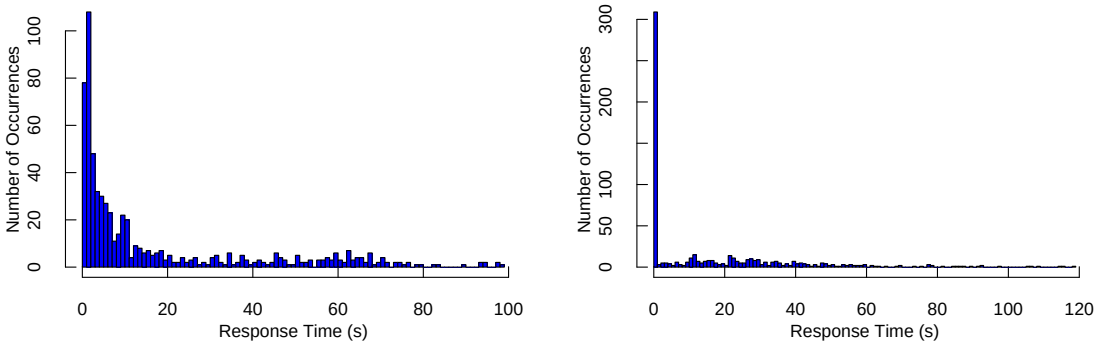

Figure 6.16. The time distribution of QRT for 1,000 concurrent queries with a) full registration, look-up b) probabilistic registration and look-up.

onds while Figure 6.16(b) shows that most occurrences of QRTs for the probabilistic approach are between 0 and 3 seconds. On the one hand, when using the full approach, MobIoT retrieves the full set of registered services at each query request and attempts to access them all in order to provide a result. On the other hand, for the probabilistic approach, response times are lower since the system retrieves only a small subset of registered services to access.

In conclusion, the experiments performed in this section illustrate the true benefits of employing our probabilistic discovery. Firstly, response times, needed to answer a user query for real time measurements/actions, are strongly decreased with respect to the traditional SOA (full discovery) counterpart. Secondly, the results highlight the scalability of MobIoT with the relatively low response times in spite of the large number of available Things and its ability to transparently manage their mobility, which is integrated in the solution by design, by successfully satisfying the coverage requirements in the whole area of interest.

6.3 MobIoT In The Future Internet

We presented in this thesis our work for the large scale mobile IoT. In addition to this perspective, our contribution took part within CHOReOS[1], with a vision towards the Future Internet, which encompasses the IoT along with an Internet of Services and an Internet of Content. CHOReOS is an FP7-ICT research project

[1]CHOReOS:http://www.choreos.eu/.

for large scale choreographies of adaptable, QoS-aware and heterogeneous services in the Future Internet. The latter is a direct evolution of today's Internet comprised of an ultra large number of Things and services. The CHOReOS Project addresses the anticipated scalability, interoperability, mobility, and adaptability issues of the Future Internet through novel approaches within a middleware solution that builds on the Service-Oriented Architecture. We proceed in the following to give an overview of the CHOReOS middleware architecture, depicted in Figure 6.17, while positioning our contribution with respect to the overall middleware solution.

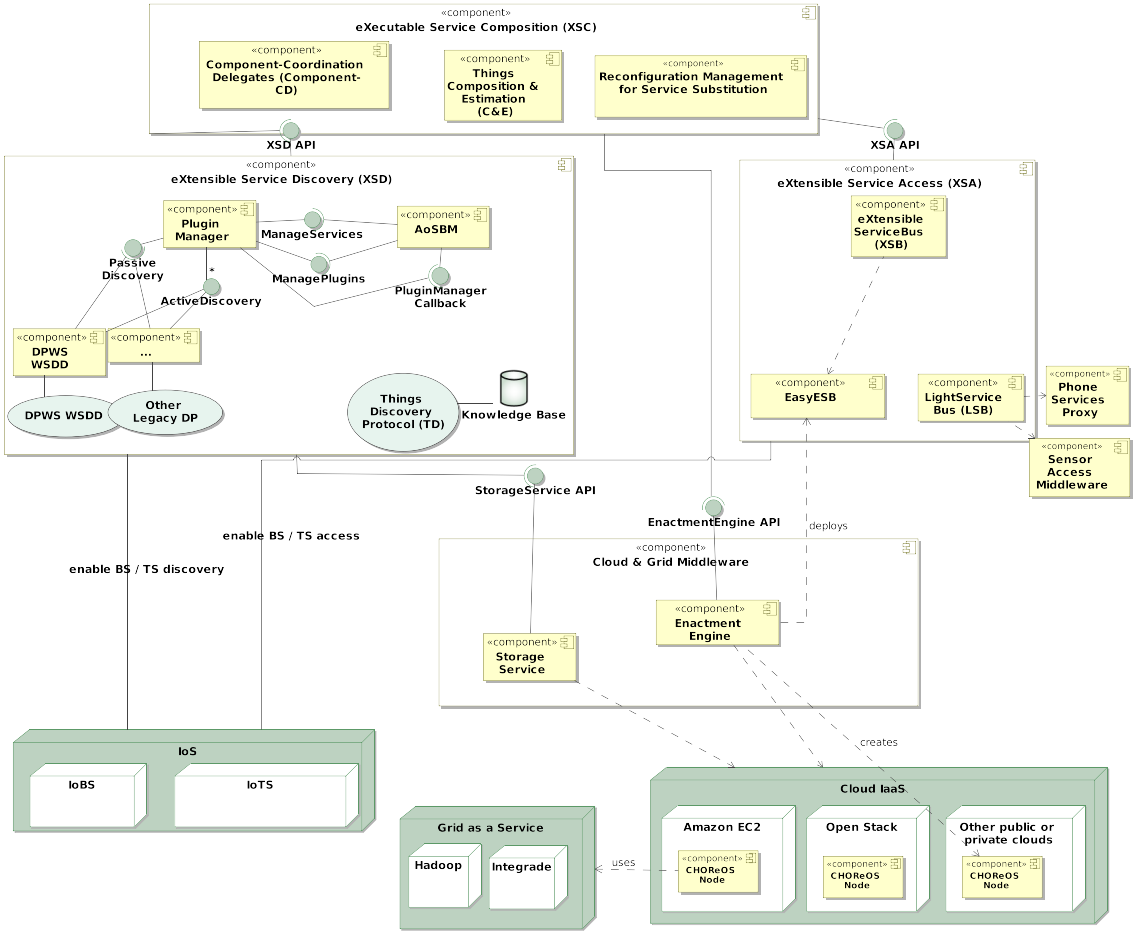

Figure 6.17. The CHOReOS middleware architecture.

6.3.1 CHOReOS Middleware Architecture

The CHOReOS middleware, is composed of four components that cooperate to enable large scale choreography deployments and executions. The components are: i) the *eXtensible Service Discovery* (XSD), which retrieves services that can take part in the choreography execution. Services can be either Business services or Thing-based services. XSD also indexes and stores the services in a service registry; ii) the *eXecutable Service Composition* (XSC), which takes as input and executes a service choreography; iii) the *eXtensible Service Access* (XSA) component, which enables runtime access to the services in the choreography; and iv) the *Cloud&Grid* component, which assists CHOReOS with handling the ultra large scale of services and data in the Future Internet. Each component is presented in the following sections.

6.3.1.1 eXtensible Service Discovery

The XSD component enables the discovery, i.e., registration and retrieval of heterogeneous business services and Thing-based services through the cooperation of three components; Discovery Plugin Framework, Abstraction-oriented Service Base Management (AoSBM), and our Thing-based service Discovery component presented in Chapters 3 and 4. It also requires the functionalities of the Registry component we presented in Chapter 3.

For scalable and interoperable discovery of business services, the Discovery Plugin Framework enables multi-protocol service discovery where, firstly, services advertise themselves by using a legacy protocol to be discovered by the appropriate plugins, which translate their descriptions to a common representation [CHOReOS consortium, 2012a]. Afterwards, the resulting service descriptions are passed to the AoSBM [CHOReOS consortium, 2011d]. The AoSBM exploits clustering mechanisms to group services based on their functional/non-functional properties and create corresponding functional/non-functional abstractions.

To enable scalable interoperable and dynamic discovery of Thing-based services, the XSD fully relies on our Thing-based service discovery approach, presented in Chapter 4, by employing our probabilistic registration and look-up functionalities implemented within MobIoT, along with the Registry we conceived to hold the service metadata and perform look-up computations. By utilizing our discovery approach, CHOReOS can better control the participation of an ultra large number of Things

while not jeopardizing the quality of the results.

6.3.1.2 eXecutable Service Composition

The eXecutable Service Composition (XSC) is the key component for composing business services and Thing-based services through the cooperation of three main components: Component coordination delegates (Component-cd), Choreography Adaptation component, and the Composition & Estimation component presented in Chapters 3 and 5.

To compose business services, the Component-cds coordinate the global interaction behavior of services taking part in a choreography, in order fulfill the specified choreography goals. In addition, to support the dynamic nature of the Future Internet, choreographies must be adaptable , i.e., adapt to available service instances at runtime. To that end, the Choreography Adaption Component enables transparent substitution of discovered services without interrupting the choreography by hiding the discovered services behind functional abstractions that are invoked instead of the actual services.

To compose Thing-based services, the XSC fully relies on the Composition & Estimation (C&E) component (Chapter 3) that concretizes the semantic and automatic composition approach, presented in Chapter 5. By relying on C&E, the CHOReOS middleware can take advantage of various sensing/actuating services and overcome the dynamic nature of the mobile environment. To deal with the scale issue of the IoT, the XSC also employs the services of the Thing-based service discovery.

6.3.1.3 eXtensible Service Access

The XSA consists of four cooperating components that enable uniform access to heterogeneous business and Thing-based services i) eXtensible Service Bus (XSB) with extensible binding components; ii) Easy Enterprise Service Bus (EasyESB); iii) the Remote Access component presented in Chapter 3; and iv) a Light Service Bus (LSB) wrapping the Thing Access Middleware presented in Chapter 3.

To enable access to business services, The XSB is based on a three-layered abstraction. Firstly, heterogeneous middleware platforms (e.g., DPWS, JMS, Java Spaces) are abstracted to connector types, representing the middleware-specific interaction paradigms (CS, PS and TS). Connector types are then abstracted into a uniform

connector type, known as Generic Application (GA). The XSB also provides a three-layered "concretization" to transform GA connector types to CS, PS, or TS connector types to then reach concrete middleware platforms. To guarantee scalability, the XSB is deployed over EasyESB, an evolution of the Bus paradigm designed for cross-paradigm interoperability.

To enable access to Thing-based services, CHOReOS firstly requires the functionalities of the Remote Access component presented in Chapter 3, in order to seamlessly request access to sensor/actuator services hosted on remote mobile Things, on the end user's behalf. the Remote Access component then interacts with the Light Service Bus (LSB) that handles the dynamics and resource constraints of Things hosting the services. The LSB employs the Thing-Access Middleware to interact with heterogeneous sensors and actuators embedded in Things, in particular, through a component of the Thing Access Middleware known as the Thing Mediator [CHOReOS consortium, 2011b], concretized by the implementation presented in Section 6.1.1. The Thing Mediator abstracts interactions with sensors/actuators and assumes vendors provide drivers that can bind with the Mediator seamlessly. The Thing Mediator then provides access to actual sensor/actuator data/actions through Thing-based services.

6.3.1.4 Cloud & Grid Support

The Cloud & Grid component is exploited for large scale infrastructure support by CHOReOS and by MobIoT as it provides various resources according to need. Resources include CHOReOS nodes, which are virtual machines hosting CHOReOS components; Storage nodes enabling elastic storage capacities; and Grid support for complex computations. The Cloud & Grid are exploited by MobIoT to provision the resources needed by the Registry to hold and store the metadata of registering services and perform computations associated with the look-up functionalities.

6.4 Summary

We presented in this chapter the implementation details of MobIoT followed by a set of experiments to assess the validity of our probabilistic registration and look-up approaches, leaving composition for our future work. To assess the validity of our approaches, we computed the difference in the coverage provided by the full set and

the subset of registered/retrieved services and the difference in the resulting set sizes. We also assessed the scalability of MobIoT by computing the Query Response Times (QRTs) of up to $1,000$ concurrent queries to register or find and access services in the probabilistic registration/look-up approach. The QRTs are then compared to the QRTs of the full registration/look-up approaches. We conclude from the set of experiments we conducted that our approaches successfully enhance user query response times and decrease the number of Things needed to participate while still providing good coverage quality.

Nonetheless, there is one operation in particular that still requires our attention: the *Find* operation. The operation is executed by the Registry and depends on the adopted storage technology where the search implementation, specific to the used technology, can slow down the search process. Furthermore, the operation should be optimized to overcome the need to repeatedly search for, possibly, the same information whenever a query over the number of available services is received. This can be accomplished by caching information and sharing it with new incoming Things for some duration before performing the search again. Another possible solution is to index the locations of services based on various known geographical regions they belong to, which decreases the search space. The count of available services in each location can be continuously updated through a background process. A key start is by investigating spatio-temporal databases. Caching and indexing this information introduces new concerns among which we identified the need to determine the acceptable duration for which cached information remains valid; the additional cost the continuous indexing process will introduce; and the potential gain of adopting those optimizations.

It is noteworthy to mention that our evaluation environment presents a limitation, which comes from the maximum number of TCP ports available at an IP address. Even if the host machine had unlimited memory resources, and the Operating System was configured to support unlimited processes for a user, each `Thing` needs to bind to a TCP port to provide access to its service. Therefore, the maximum number of available TCP ports imposes a limit on the number of `Things` that can be run concurrently on a single machine, thus limiting the scale at which we could perform our evaluations.

In conclusion, our evaluations demonstrate that our Thing-based Service-Oriented Architecture, concretized by MobIoT, can better handle the anticipated IoT scale and

its mobility by controlling the participation of Things without strongly compromising the quality of the outcome. Additionally, our approaches can successfully decrease communication overload by requiring the involvement of and access to fewer services. Finally, by exploiting the Cloud, our Registry can handle any number of services and perform all needed complex computations.

Chapter 7

Conclusion

A few years ago, the Internet of Things was merely a vision, where the physical environment becomes connected to the Internet and becomes smarter by exploiting context-aware objects with computation capabilities, referred to as *Things*. Today, the vision is already a reality, with context-awareness accomplished by sensors and actuators seamlessly embedded within phones, vehicles, home appliances, and even fabric.

Moreover, a considerable portion of those Things have the ability to move, either autonomously or assisted by other Things/humans, which introduces several advantages, yet comes with unprecedented challenges. The anticipated challenges stem from the fact that mobility is becoming the norm rather than the exception. Although the Internet of Things is not expected to be solely mobile, humans are increasingly becoming dependent on their mobile gadgets, i.e., mobile Things, which are increasing in types, numbers, and sensing/actuation capabilities, i.e., sensing/actuation services. In particular, the most critical challenges are handling the heavily increasing number of Things, which is only aggravated by their different types, and most importantly, by their mobility. In other terms, a scalable, interoperable and dynamic system is crucial to successfully abstracting the physical world into a mobile Internet of Things.

To that end, we presented in this thesis a middleware solution that revisits the commonly adopted Service-Oriented Architecture in order to enable any application, in such a setting, referred to as IoT application, to perform properly in spite of the above challenges. We proceed in the following with a summary of our specific

contributions followed by an overview of our short term research goals to optimize our solution and long-term research goals to build on the work done in the thesis and the expertise we acquired.

7.1 Summary of Contributions

Our main contribution lies in conceiving MobIoT, a Thing-based Service-Oriented middleware that offers novel probabilistic service discovery and composition approaches, and wraps legacy access protocols to be seamlessly executed by the middleware. The middleware exposes two levels of abstractions: abstracting a Thing as a service (on the service provider side); and abstracting Things measurements/actions as a service (on the service consumer side). Our contributions can be summarized in the following.

Semantic Support for Application Development: Although our goal is not to provide a middleware for IoT application development, we posit that leaving the required expertise to the application developer is not a feasible option, especially with the complexity of sensing/actuation tasks. This is partly due to their highly specialized domains (signal processing, estimation theory, robotics, etc.), which demand application programmers to assume the role of domain experts. To that end, we encapsulate all needed knowledge in a set of ontologies that assist developers in creating high-level applications while leaving the sensing/actuation specific logic to be executed by our middleware. In particular, we provide two ontologies: a **Device** ontology that models Things, sensor, actuators, and their internal functionalities; and a (physics) **Domain** ontology, that models the real world features to measure/act on, i.e., physical concepts, and comprises physical/mathematical models pertaining to sensing/actuation tasks.

Scalable Probabilistic Thing-based Service Registration: The load on the network, especially at the communication and storage levels, resulting from the ultra large number of Things willing to provide their services, is addressed upon registration (publishing) of those services, which is now probabilistic. The proposed registration approach controls the participation of Things and enables only a portion of those Things to register their services, depending on the benefit they introduce. Precisely, the decision is based on whether or not other registered services with similar sensing/actuation capacities are sufficient, in particular, in terms of the coverage they

provide.

Scalable Probabilistic Thing-based Service Look-up: With the anticipated scale, even if only a portion of willing Things are allowed to register their services, the resulting number remains high and so does the load on the communication network. As a solution, we provide a probabilistic service retrieval (look-up) approach that is executed seamlessly by the middleware, without requiring any involvement from the end user or the developer. The probabilistic look-up returns only a subset of services based on the type of the event to measure/act on. Our solution relies on the fact that in many case, a point by point coverage of an area is not always a critical requirement and it is deemed sufficient to sample Things that follow a certain distribution in space, which depends on the nature of the event.

Semantic Thing-based Service Composition: Thing-based service composition no longer needs to be specified by the end user or by the developer. All compositions are specified independently of the IoT application, by a domain expert, within the domain ontology. Moreover, the specifications are modeled as mathematical expressions that represent the physics behind sensing/actuation tasks, which ensures that the composition process takes the specificities of those tasks into account. Consequently, compositions are executed automatically and transparently, by exploiting the knowledge in the ontology, without requiring any involvement from consumers and developers who most likely do not possess the needed expertise.

Transparent Thing-based Service Access: Although we adopt legacy access protocols, we ensure that access to service providers is executed transparently in a timely manner by exploiting the probabilistic discovery approach we introduced, which guarantees faster response times, a key requirement to satisfactory user experience.

In addition to the novel middleware architecture, we also provide a prototype implementation released as open source. Both theoretical and implementation efforts were conceived as part of CHOReOS, a research project with efforts directed towards the realization of the Future Internet vision. We then exploited the prototype implementation to validate our approaches as well as evaluate the scalability of the overall middleware solution. Our results demonstrate that MobIoT can successfully control the participation of Things while not compromising the quality of the results (pertaining to sensing/actuation requests). Additionally, our experiments demonstrated that MobIoT can drastically enhance query response time, i.e., the time needed for an IoT application to satisfy a user request for Thing-based services.

In addition to the scientific contributions above, our first high-level contribution in this thesis, where we provide a coherent approach that enables systems to manage the anticipated ultra large number of mobile Things at low costs, is to take the large-scale mobile IoT vision one step forward towards its realization by devising a realistic solution, designed for real-life applications and scenarios. Our second contribution is to provide a product that is in a mature stage and provides a solid foundation to be transformed into an end-user product that can be exploited to efficiently assist developers of IoT applications in their development endeavors. Finally, we consider that our work in this thesis and contributions above can pave the way towards further research efforts to extend our solution in order to support more comprehensive IoT applications that cover a larger diaspora of real life scenarios and more importantly incorporate more constrained Things as first class actors.

7.2 Future Work

In addition to the contributions we provided in this thesis, we identified several technical optimizations that can be exploited within each approach, to enhance the quality of our solution. We also identified general optimizations to be exploited regarding the high-level logic of each approach. In addition to those short-term goals, we present our future work from a long term perspective that builds on the expertise we acquired and research interests we gained during this thesis.

7.2.1 Short-term Perspective

Throughout the evolution of the work in this thesis, we made several assumptions about the nature of the physical environment, in which MobIoT operates. Our future short-term research goals are to address those assumptions, related in particular, to the mobility of the Things and their physical locality.

Mobility estimations. In particular, when performing our probabilistic computation that exploits mobility models, we assumed independence between the displacement of mobile Things on the X-axis and the Y-axis, therefore leaving the correlation of the displacements, on those two axis, unaccounted for. We plan on relaxing this assumption by exploiting polar-based representations of displacements as suggested

by Sadiq *et al.* in [Sadiq and Kumar, 2011]. Additionally, given that human-carried Things are major players in our architecture, we consider an important improvement to the mobility estimations, to be by accounting for the correlations between the displacements of individuals, especially that in most cases, individuals travel in groups and follow repetitive patterns.

Composition computations. While conceiving our approach to Thing-based service composition, we assumed Things in each other's vicinity (within the same cell of some grid) to be collocated. Collocation of sensor/actuator measurements/actions is an important requirement when composing different types of measurements/actions to estimate/modify the state of a feature of interest at a location of interest. Consequently, we plan on relaxing this assumption and further investigating fusion techniques for spatially-correlated sensor data. The revisited approach should be further evaluated to assess its quality and performance. In particular, our aim is to compare the quality of results provided through composition to the quality of results provided directly by atomic services, in the same setting.

In addition to the relaxed assumptions above, we are interested in a more comprehensive solution that requires several optimizations, to be applied, on a higher level, in both Thing-based service discovery and composition approaches.

Utility-based probabilistic discovery. In its current state, the probabilistic discovery approach is solely dependent on area-coverage requirements, which can be restrictive. Our future work consists of conceiving a multi-criteria model that accounts for energy consumption, and service quality. Given the context, service quality is mapped to sensor/actuator accuracy, sensor/actuator confidence in the measurements/actions they provide, and finally service response time. Sensor confidence reflects how certain a sensor is of the correctness of the measurements it provides.

Data estimation. By exploiting probabilistic confidence models provided by sensors themselves, and the fact that the value of a concept can be estimated instead of it being measured, we plan on integrating estimation techniques to enable the middleware to estimate the most likely true value of the data at a desired spatial point.

Approximately-optimal composition. Given that scalability is a central focus in our thesis, and although scalability requirements are successfully ensured by our contribution on probabilistic discovery, we are interested in providing an independently scalable Thing-based service composition. In more detail, rather than adopting a brute-force philosophy to retrieve all possible compositions, we deem it more scalable to adopt an approximately-optimal composition which revisits the expansion and mapping phases. The former should return only candidates (concepts to measure/act on) with the highest likelihood of having corresponding service types. The latter should no longer attempt to find all possible mappings (to identity needed sensor/actuator types), but instead, pick a small subset of feasible mappings based on the number of available running services that abstract the sensors/actuators.

7.2.2 Long-term Perspective: Better World With Social Urban Monitoring

During my Master work, my research revolved around mobile social networking, with a focus on privacy and access control. My interest in the social aspect of mobile computing has not faded away. With the empowerment of users, who are now data producers rather than consumers, and the proliferation of sensors, both embedded in Things and deployed statically within large sensor networks, social and physical data about the environment is becoming widely available. Bearing this in mind, I am confident that the coming era is that of publicly accessible information that will be exploited to better understand and manage our world. As such, my research interests have evolved throughout my thesis towards mobile sensing that is in particular directed towards empowering people, with not only social data generation, but also the generation of scientific data about their environments, noise information being an example, that can be exploited to enhance their daily lives.

Consequently, with MobIoT's extensible design and operational prototype, the middleware along with my acquired expertise can be leveraged in a long term future plan to enable large-scale environmental monitoring with the end goal to assist users in better understanding the ambient conditions that strongly affect their lives and enable them, individually and collectively, to enhance their quality of life and well-being by better caring for and eventually protecting their surrounding environment (for instance with respect to noise, pollution, etc.). This goal can be achieved by ex-

ploiting information users acquire through applications —that can employ MobIoT's functionalities— hosted on their mobile Things (e.g., tablets or smartphones). Yet, for the acquired information to be truly useful, especially when provided by/targeted towards the common population, with no scientific expertise, a learning/inference mechanism must be put in place to extract higher level knowledge from the raw data provided by sensors. Additionally, for the latter to be feasible, instantaneous access to discrete data, as done in this thesis, becomes insufficient and new protocols are to be devised to manage large volumes of real-time and historical data streams.

Another requirement relates to the social sensing aspect, where information should be provided by end users themselves based on their own perspectives regarding their environment, which in many cases can be more meaningful than a physical sensor's numerical readings. Additionally, new protocols should be devised to seamlessly integrate the social feedback with the sensor provided measurements.

Last but not least, when talking about mobile sensing in general, and social sensing in particular, privacy concerns should always be addressed with high priority, which is another topic of interest for which I acquired expertise during my previous research. As such, as part of my future research, I intend to partially focus on this topic and possibly integrate access control and privacy protection mechanisms within MobIoT to be exploited when users/Things providing their data deem it more suitable to limit access to their information.

Bibliography

Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *International Conference on Mobile Data Management*, pages 198–205, 2007. 20

Asaad Ahmed, Keiichi Yasumoto, Yukiko Yamauchi, and Minoru Ito. Distance and time based node selection for probabilistic coverage in people-centric sensing. In *Proceedings of the 8th Annual IEEE International Conference on Sensor, Mesh and Ad Hoc Communications and Networks, (SECON)*, pages 134–142, 2011. 58

Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002. 128

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. 14

Samer AB Awwad, Chee Kyun Ng, Nor K Noordin, and Mohd Fadlee A Rasid. Cluster based routing protocol for mobile nodes in wireless sensor network. *Wireless Personal Communications*, 61(2):251–281, 2011. 128

Naba Aziz, Ammar W Mohemmed, and Daya Sagar. Particle swarm optimization and voronoi diagram for wireless sensor networks coverage optimization. In *International Conference on Intelligent and Advanced Systems, (ICIAS)*, pages 961–965. IEEE, 2007. 58

Fan Bai and Ahmed Helmy. A survey of mobility models. *Wireless Ad hoc Networks. University of Southern California, USA*, 206, 2004. 149

Xiaole Bai, Santosh Kumar, Dong Xuan, Ziqiu Yun, and Ten H. Lai. Deploying wireless sensors to achieve both coverage and connectivity. In *Proceedings of the*

7th ACM International Symposium on Mobile Ad Hoc Networking and Computing, pages 131–142. ACM, 2006. 44

Mohamed Bakillah and Steve HL Liang. Discovering sensor services with social network analysis and expanded SQWRL querying. In *Web and Wireless Geographical Information Systems*, pages 221–238. Springer, 2012. 105

Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003. 24

Aaron Beach, Mike Gartrell, Sirisha Akkala, Jack Elston, John Kelley, Keisuke Nishimoto, Baishakhi Ray, Sergei Razgulin, Karthik Sundaresan, Bonnie Surendar, et al. Whozthat? evolving an ecosystem for context-aware mobile social networks. *Network, IEEE*, 22(4):50–55, 2008. 27

Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. SUMO - Simulation of Urban Mobility: An Overview. In *The Third International Conference on Advances in System Simulation, (SIMUL)*, Barcelona, Spain, 2011. 142

Farshid Hassani Bijarbooneh, Pierre Flener, Edith C-H Ngai, and Justin Pearson. An optimisation-based approach for wireless sensor deployment in mobile sensing environments. In *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, pages 2108–2112. IEEE, 2012. 93

Colin Blackman, Ian Brown, Jonathan Cave, Simon Forge, Karmen Guevara, Lara Srivastava, Motohiro Tsuchiya, and Rafael Popper. Towards a Future Internet interrelation between technological, social and economic trends. Technical report, 2010. http://www.internetfutures.eu/wp-content/uploads/2010/11/TAFIFinal-Report.pdf. 5

Carsten Bormann, Angelo P Castellani, and Zach Shelby. CoAP: An Application Protocol for billions of tiny Internet nodes. *Internet Computing, IEEE*, 16(2): 62–67, 2012. 21

Mike Botts, George Percivall, Carl Reed, and John Davidson. OGC® Sensor Web Enablement: Overview and high level architecture. In *GeoSensor networks*, pages 175–190. Springer, 2008. 16

Dirk Brockmann, Lars Hufnagel, and Theo Geisel. The scaling laws of human travel. *Nature*, 439(7075):462–465, 2006. 59

Andrew T Campbell, Shane B Eisenman, Nicholas D Lane, Emiliano Miluzzo, Ronald A Peterson, Hong Lu, Xiao Zheng, Mirco Musolesi, Kristóf Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *Internet Computing, IEEE*, 12(4):12–21, 2008. 3

Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys (CSUR)*, 44(1):1, 2012. 105

CASAGRAS, EU. FP7 Project, RFID and the inclusive model for the Internet of Things. Technical report, 2012. `www.grifs-project.eu`. 1

Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of human mobility on opportunistic forwarding algorithms. *IEEE Transactions on Mobile Computing*, 6(6):606–620, 2007. 59

Ali Chamam and Samuel Pierre. On the planning of wireless sensor networks: energy-efficient clustering under the joint routing and coverage constraint. *IEEE Transactions on Mobile Computing*, 8(8):1077–1086, 2009. 93

Supriyo Chatterjea and Paul Havinga. An adaptive and autonomous sensor sampling frequency control scheme for energy-efficient data acquisition in wireless sensor networks. In *Distributed Computing in Sensor Systems*, pages 60–78. Springer, 2008. 58

Mainak Chatterjee, Sajal K Das, and Damla Turgut. WCA: A weighted clustering algorithm for mobile ad hoc networks. *Cluster Computing*, 5(2):193–204, 2002. 128

Maggie Cheng, Xuan Gong, and Lin Cai. Joint routing and link rate allocation under bandwidth and energy constraints in sensor networks. *Wireless Communications, IEEE Transactions on*, 8(7):3770–3779, 2009. 128

Kyungmin Cho, Younghyun Ju, Sungjae Jo, Yunseok Rhee, and Junehwa Song. SATI: A scalable and traffic-efficient data delivery infrastructure for real-time sensing applications. *Computer Networks*, 55(1):241–263, 2011. 23

Wook Choi and Sajal K Das. CROSS: A probabilistic constrained random sensor selection scheme in wireless sensor networks. *Performance Evaluation*, 66(12):754–772, 2009. 86, 92

CHOReOS consortium. CHOReOS Perspective on the Future Internet and Initial Conceptual Model (D1.2). Technical report, 2011a. URL http://www.choreos.eu/. 11

CHOReOS consortium. CHOReOS Middleware Specification (D3.1). Technical report, 2011b. URL http://www.choreos.eu/. 11, 163

CHOReOS consortium. DynaRoute Architectural Design (D8.2). Technical report, 2011c. URL http://www.choreos.eu/. 137

CHOReOS consortium. CHOReOS Dynamic Development Model Definition (D2.1). Technical report, 2011d. URL http://www.choreos.eu/. 161

CHOReOS consortium. Initial Architectural Style for CHOReOS Choreographies (D1.3). Technical report, 2012a. URL http://www.choreos.eu/. 11, 161

CHOReOS consortium. CHOReOS Middleware First Implementation (D3.2.2). Technical report, 2012b. URL http://www.choreos.eu/. 11

CHOReOS consortium. Integrated CHOReOS Middleware - Enabling large-scale, QoS-Aware Adaptive Choreographies (D3.3). Technical report, 2013. URL http://www.choreos.eu/. 11

Benoit Christophe. Managing massive data of the Internet of Things through cooperative semantic nodes. In *IEEE Sixth International Conference on Semantic Computing, (ICSC)*, pages 93–100. IEEE, 2012. 37

Benoit Christophe, Vincent Verdot, and Vincent Toubiana. Searching the 'Web of Things'. In *Fifth IEEE International Conference on Semantic Computing, (ICSC)*, pages 308–315. IEEE, 2011. 37

Cisco Visual Networking Index Cisco. Global mobile data traffic forecast update, 2010-2015. *Cisco white paper*, 2011. 4

Michael Compton, Holger Neuhaus, Kerry Taylor, and Khoi-Nguyen Tran. Reasoning about sensors and compositions. In *SSN*, pages 33–48, 2009. 16, 104

Michael Compton, Payam Barnaghi, Luis Bermudez, Raul García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012. 16, 17, 37

Mathieu d'Aquin and Natalya F Noy. Where to publish and find ontologies? a survey of ontology libraries. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11:96–111, 2012. 17

Tathagata Das, Prashanth Mohan, Venkata N Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. PRISM: platform for remote sensing using smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pages 63–76. ACM, 2010. 30

Suparna De, Tarek Elsaleh, Payam Barnaghi, and Stefan Meissner. An Internet of Things platform for real-world and digital objects. *Scalable Computing: Practice and Experience*, 13(1), 2012. 17, 37

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 23

Santpal Singh Dhillon and Krishnendu Chakrabarty. Sensor placement for effective coverage and surveillance in distributed sensor networks. In *Proceedings of IEEE Wireless Communications and Networking Conference*, 2003. 93

Hui Ding, Goce Trajcevski, and Peter Scheuermann. Efficient maintenance of continuous queries for trajectories. *Geoinformatica*, 12(3):255–288, 2008. 69

Aida Ehyaei, Eduardo Tovar, Nuno Pereira, and Björn Andersson. Scalable data acquisition for densely instrumented cyber-physical systems. In *IEEE/ACM International Conference on Cyber-Physical Systems, (ICCPS)*, pages 174–183. IEEE, 2011. 23

Shane Eisenman, Nicholas Lane, Emiliano Miluzzo, Ronald Peterson, Gahng-Seop Ahn, and Andrew Campbell. MetroSense project: People-centric sensing at scale. In *Workshop on World-Sensor-Web (WSW), Boulder*. Citeseer, 2006. 29

Martin Erwig, Ralf Hartmut Gu, Markus Schneider, Michalis Vazirgiannis, et al. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999. 69

Heitor Ferreira, Sérgio Duarte, and Nuno Preguiça. 4Sensing–decentralized processing for participatory sensing data. In *IEEE 16th International Conference on Parallel and Distributed Systems, (ICPADS)*, pages 306–313. IEEE, 2010. 23

Antoine Gallais, Jean Carle, David Simplot-Ryl, and Ivan Stojmenovic. Ensuring area k-coverage in wireless sensor networks with realistic physical layers. In *5th IEEE Conference on Sensors*, pages 880–883. IEEE, 2006. 44

Shravan Gaonkar, Jack Li, Romit Roy Choudhury, Landon Cox, and Al Schmidt. Micro-blog: sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 174–186. ACM, 2008. 27

Marta C Gonzalez, Cesar A Hidalgo, and Albert-Laszlo Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, 2008. 59

Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. Spatio-temporal data handling with constraints. *GeoInformatica*, 5(1):95–115, 2001. 69

Tao Gu, Hung Keng Pung, and Da Qing Zhang. A Service-Oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, 2005. 7

N. Guarino, M. Carrara, and P. Giaretta. An ontology of meta-level categories. In *4th International Conference Principles of Knowledge Representation and Reasoning*, pages 270–280. Citeseer, 1994. 16

D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-Based Internet of Things: Discovery, query, selection, and on-demand provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3): 223–235, 2010. 7, 16, 19, 57, 105

Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*, pages 97–129. Springer, 2011. 20

Sara Hachem, Thiago Teixeira, and Valérie Issarny. Ontologies for the Internet of Things. In *Proceedings of the 8th Middleware Doctoral Symposium*, page 3. ACM, 2011. 11, 36

Sara Hachem, Animesh Pathak, and Valérie Issarny. Probabilistic registration for large-scale mobile participatory sensing. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications, (Percom)*, mar. 2013a. 11, 60

Sara Hachem, Animesh Pathak, and Valerie Issarny. Service-Oriented middleware for large-scale mobile participatory sensing. *Pervasive and Mobile Computing*, 2013b. 11, 60

Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. The Internet of Things in an enterprise context. In *Future Internet–FIS 2008*, pages 14–28. Springer, 2009. 2

Wendi B Heinzelman, Anantha P Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, 2002. 128

Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 125–138. ACM, 2006. 29

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 604–613. ACM, 1998. 128

Valérie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Françoise Sailhan, Rafik Chibout, Nicole Levy, and Angel Talamona. Developing ambient intelligence systems: A solution based on Web Services. *Automated Software Engineering*, 12(1):101–137, 2005. 7

Valerie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Pano Vassiliadist, Maraco Autili, Marco A. Gerosa, and Amira Ben Hamida. Service-Oriented middleware for the Future Internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011. 11

Swaroop Kalasapur, Mohan Kumar, and Behrooz A Shirazi. Dynamic service composition in pervasive computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):907–918, 2007. 7

Thomas Karagiannis, J-Y Le Boudec, and Milan Vojnovic. Power law and exponential decay of intercontact times between mobile devices. *IEEE Transactions on Mobile Computing*, 9(10):1377–1390, 2010. 59

Stamatis Karnouskos, Domnic Savio, Patrik Spiess, Dominique Guinard, Vlad. Trifa, and Oliver Baecker. Real-world service interaction with enterprise systems in dynamic manufacturing environments. *Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management*, pages 423–457, 2010. 57

Gabriel Y Keung, Qian Zhang, and Bo Li. The delay-constrained information coverage problem in mobile sensor networks: single hop case. *Wireless Networks*, 16(7): 1961–1973, 2010. 58

Wazir Z. Khan, Yang Xiang, Mohammed Y. Aalsalem, and Quratulain Arshad. Mobile phone sensing systems: A survey. *IEEE Communications Surveys Tutorials*, 15(1):402–427, 2013. 26, 27, 28

Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. Atlas: A Service-Oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Proceedings of 31st IEEE Conference on Local Computer Networks*, pages 630–638. IEEE, 2006. 7

Nicholas D Lane, Shane B Eisenman, Mirco Musolesi, Emiliano Miluzzo, and Andrew T Campbell. Urban sensing systems: opportunistic or participatory? In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications*, pages 11–16. ACM, 2008. 26

José Antonio Cotelo Lema, Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46(6):680–712, 2003. 69

Jize Li, Kejie Li, and Wei Zhu. Improving sensing coverage of wireless sensor networks by employing mobile robots. In *IEEE International Conference on Robotics and Biomimetics, (ROBIO)*, pages 899–903. IEEE, 2007. 58

Valerie Loscri, Enrico Natalizio, Tahiry Razafindralambo, and Nathalie Mitton. Distributed algorithm to improve coverage for mobile swarms of sensors. In *IEEE International Conference on Distributed Computing in Sensor Systems, (DCOSS)*, pages 292–294, 2013. 58

Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, pages 165–178. New York, NY, USA, 2009. 27

Hong Lu, Nicholas D Lane, Shane B Eisenman, and Andrew T Campbell. Bubble-sensing: Binding sensing tasks to the physical world. *Pervasive and Mobile Computing*, 6(1):58–71, 2010. 30

Samuel R. Madden, Micheal J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems, TODS*, 30(1):122–173, 2005. 48

Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010. 5

Peter Mell and Timothy Grance. The NIST definition of Cloud computing. *NIST special publication*, 800(145):7, 2011. 24

Richard Mietz, Sven Groppe, Kay Römer, and Dennis Pfisterer. Semantic models for scalable search in the Internet of Things. *Journal of Sensor and Actuator Networks*, 2(2):172–195, 2013. 23

Daniele Miorandi. The impact of channel randomness on coverage and connectivity of ad hoc and sensor networks. *IEEE Transactions on Wireless Communications*, 7(3):1062–1072, 2008. 44

Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of Things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012. 2

Nathalie Mitton, Symeon Papavassiliou, Antonio Puliafito, and Kishor S Trivedi. Combining Cloud and sensors in a smart city environment. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):1–10, 2012. 24

Mirco Musolesi, Mattia Piraccini, Kristof Fodor, Antonio Corradi, and Andrew T Campbell. Supporting energy-efficient uploading strategies for continuous sensing applications on mobile phones. In *Pervasive Computing*, pages 355–372. Springer, 2010. 58

D. Papadimitriou. Future Internet–The Cross-ETP Vision Document. *European Technology Platform, Alcatel Lucent*, 8, 2009. 3, 5

M.P. Papazoglou. Service-Oriented Computing: concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering.*, pages 3 – 12, dec. 2003. 6

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web services vs. big'Web services: making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, pages 805–814. ACM, 2008. 19

Nikos Pelekis, Babis Theodoulidis, Ioannis Kopanakis, and Yannis Theodoridis. Literature review of spatio-temporal database models. *The Knowledge Engineering Review*, 19(03):235–274, 2004. 69

Charith Perera, Arkady Zaslavsky, C Liu, Michael Compton, Peter Christen, and Dimitrios Georgakopoulos. Sensor search techniques for sensing as a service architecture for the Internet of Things. 2013. 58

Charith Pereral, Arkady Zaslavsky, Peter Christen, Ali Salehi, and Dimitrios Georgakopoulos. Capturing sensor data from mobile phones using Global Sensor Network middleware. In *IEEE 23rd International Symposium on Personal Indoor and Mobile Radio Communications, (PIMRC)*, pages 24–29. IEEE, 2012. 4

A Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *Proceedings of 13th International Conference on Data Engineering*, pages 422–432. IEEE, 1997. 69

Rajib Kumar Rana, Chun Tung Chou, Salil S Kanhere, Nirupama Bulusu, and Wen Hu. Ear-phone: an end-to-end participatory urban noise mapping system. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 105–116. ACM, 2010. 23

D. Raychaudhuri and M. Gerla. *Emerging Wireless Technologies and the Future Mobile Internet*. Cambridge University Press, 2011. 45

Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, and Song Chong. On the Levy-Walk nature of human mobility. In *Proceedings of the 27th IEEE Conference on Computer Communications, (INFOCOM)*, pages 924 –932, april 2008. 59, 72

Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, Seong Joon Kim, and Song Chong. On the Levy-Walk nature of human mobility. *IEEE/ACM Transactions on Networking, (TON)*, 19(3):630–643, 2011. 59, 79

Leonard Richardson and Sam Ruby. *RESTful Web services*. O'Reilly, 2008. 19

Radhika Ranjan Roy. Random Walk Mobility. In *Handbook of Mobile Ad Hoc Networks for Mobility Models*, pages 35–63. Springer, 2011. 149

Zheng Ruan, Edith C-H Ngai, and Jiangchuan Liu. Wireless sensor deployment for collaborative sensing with mobile phones. *Computer Networks*, 55(15):3224–3245, 2011. 58

David J Russomanno, Cartik R Kothari, and Omoju A Thomas. Building a sensor ontology: A practical approach leveraging ISO and OGC models. In *IC-AI*, pages 637–643, 2005. 16, 37

U. Sadiq and M. Kumar. Proximol: Proximity and mobility estimation for efficient forwarding in opportunistic networks. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, pages 312–321. IEEE, 2011. 71, 146, 171

Loïc Schmidt, Nathalie Mitton, David Simplot-Ryl, Roudy Dagher, and Roberto Quilez. DHT-based distributed ALE engine in RFID Middleware. In *IEEE International Conference on RFID-Technologies and Applications, (RFID-TA)*, pages 319–326. IEEE, 2011. 23

SENSEI consortium. Highly Scalable Architecture Framework (D3.6). Technical report, 2011. URL http://www.ict-sensei.org/index.php?option=com_content&task=view&id=14&Itemid=31. 24

Richard Serfozo. *Introduction to stochastic networks*, volume 44. Springer Verlag, 1999. 150

Minho Shin, Cory Cornelius, Dan Peebles, Apu Kapadia, David Kotz, and Nikos Triandopoulos. AnonySense: A system for anonymous opportunistic sensing. *Pervasive and Mobile Computing*, 7(1):16–30, 2011. 29

Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of predictability in human mobility. *Science*, 327(5968):1018–1021, 2010a. 45

Zhexuan Song, Alvaro A Cárdenas, and Ryusuke Masuoka. Semantic middleware for the Internet of Things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010b. 21

W. Stadje. The exact probability distribution of a two-dimensional Random Walk. *Journal of Statistical Physics*, 46(1):207–216, 1987. 150

Jeffrey Star and John Estes. *Geographic Information Systems*. Prentice-Hall, 1990. 69

Peter Stuckmann and Rainer Zimmermann. European research on Future Internet design. *IEEE Wireless Communications*, 16(5):14–22, 2009. 5

Thiago Teixeira, Sara Hachem, Valerie Issarny, and Nikolaos Georgantas. Service Oriented Middleware for the Internet of Things: A Perspective. *Servicewave*, 2011. 11, 46

I. Toma, E. Simperl, A. Filipowska, G. Hench, and J. Domingue. Semantics-driven interoperability on the Future Internet. In *IEEE International Conference on Semantic Computing (ICSC)*, 2009. 5

Vlasios Tsiatsis, Alexander Gluhak, Tim Bauge, Frederic Montagut, Jesus Bernat, Martin Bauer, Claudia Villalonga, Payam Barnaghi, and Srdjan Krco. The SENSEI real world Internet architecture. *Towards the Future Internet-Emerging Trends from European Research*, pages 247–256, 2010. 57

Sandesh Uppoor, Oscar Trullols-Cruces, Marco Fiore, and Jose M. Barcelo-Ordinas. Generation and analysis of a large-scale urban vehicular mobility dataset. *IEEE Transactions on Mobile Computing*, 99(PrePrints):1, 2013. ISSN 1536-1233. 142, 157

Ovidiu Vermesan and Peter Friess. *Internet of Things-Global Technological and Societal Trends From Smart Environments and Spaces to Green ICT*. River Publishers, 2011. 5

Claudia Villalonga, Martin Bauer, Vincent Huang, Jesús Bernat, and Payam Barnaghi. Modeling of sensor data and context for the real world Internet. In *8th IEEE International Conference on Pervasive Computing and Communications Workshops, (PERCOM Workshops)*, pages 1–6. IEEE, 2010. 37

M. Von Kaenel, P. Sommer, and R. Wattenhofer. Ikarus: large-scale participatory sensing at high altitudes. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 63–68. ACM, 2011. 28

W3C. Web Services Architecture. Technical report. URL `http://www.w3c.org/TR/ws-arch`. 18

Bang Wang. Sensor coverage model. In *Coverage Control in Sensor Networks*, pages 19–34. Springer, 2010. 44

Jianping Wang. Exploiting mobility prediction for dependable service composition in wireless mobile ad hoc networks. *IEEE Transactions on Services Computing*, 4 (1):44–55, 2011. 86

Wei Wang, Suparna De, Ralf Toenjes, Eike Reetz, and Klaus Moessner. A comprehensive ontology for knowledge representation in the Internet of Things. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, (TrustCom)*, pages 1793–1798. IEEE, 2012. 37

Xue Wang, Sheng Wang, and Jun-Jie Ma. An improved co-evolutionary particle swarm optimization for wireless sensor networks with dynamic deployment. *Sensors*, 7(3):354–370, 2007. 58

Yi Wang, Jialiu Lin, Murali Annavaram, Quinn A Jacobson, Jason Hong, Bhaskar Krishnamachari, and Norman Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, pages 179–192. ACM, 2009. 58

Roy Want. Enabling ubiquitous sensing with RFID. *Computer*, 37(4):84–86, 2004. 22

Roy Want. An introduction to RFID technology. *Pervasive Computing, IEEE*, 5(1): 25–33, 2006. 22

Thakshila Wimalajeewa and Sudharman K. Jayaweera. Impact of mobile node density on detection performance measures in a hybrid sensor network. *IEEE Transactions on Wireless Communications*, 9(5):1760–1769, 2010. 150

Yanwei Wu, Xiang-Yang Li, YunHao Liu, and Wei Lou. Energy-efficient wake-up scheduling for data collection and aggregation. *IEEE Transactions on Parallel and Distributed Systems*, 21(2):275–287, 2010. 128

Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 70–84. ACM, 2001. 83, 88

Fan Ye, Haiyun Luo, Jerry Cheng, Songwu Lu, and Lixia Zhang. A two-tier data dissemination model for large-scale wireless sensor networks. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 148–159. ACM, 2002. 83, 88

Jong-Woon Yoo and Kyu Ho Park. A cooperative clustering protocol for energy saving of mobile devices with WLAN and Bluetooth interfaces. *IEEE Transactions on Mobile Computing*, 10(4):491–504, 2011. 128

Ossama Younis and Sonia Fahmy. Heed: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *Mobile Computing, IEEE Transactions on*, 3(4):366–379, 2004. ISSN 1536-1233. 128

Jing Yuan, Yu Zhen, Xing Xie, and Guang Sun. Driving with knowledge from the physical world. In *In The 17th ACM SIGKDD International cConference on Knowledge Discovery and Data Mining, (KDD)*, 2011. 142

Suli Zhao and Dipankar Raychaudhuri. Scalability and performance evaluation of hierarchical hybrid wireless networks. *IEEE/ACM Transactions on Networking*, 17(5):1536–1549, 2009. 128

Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. From today's Intranet of Things to a Future Internet of Things: a wireless-and mobility-related view. *IEEE Wireless Communications*, 17(6):44–51, 2010. 6