Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

uDiscovery: An Urban-Centric Model for Service Discovery in Smart Cities

Christian Humberto Cabrera Jojoa

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science) in the

School of Computer Science & Statistics
Trinity College Dublin, The University of Dublin

June 2020

Declaration

Christian Humberto Cabrera Jojoa

June 2020

Abstract

Cities offer services to their citizens to improve their overall quality of life (e.g., education, or health care services) and these services are frequently supported by digital information (e.g., library opening hours or hospital status) on the Internet. IoT devices have potential to further enhance city services as they can provide additional digital information from direct interaction with local urban spaces. These devices enable the collection of city and citizens' context, which may be useful to infer citizens' needs. For example, a citizen at a bus stop is more likely to need information about transportation city services. Service-oriented architectures (SOAs) are commonly-used to manage the services provided by IoT devices and include processes to register, discover, compose, execute, and monitor services. This thesis focuses on the discovery process, which is challenging for IoT services because they are fundamentally different from web services in the traditional Internet. In particular, the expected number of services is much greater, which will impact discovery efficiency, and they work in dynamic environments where distributed and adaptive architectures are more appropriate.

Existing research on IoT and service-oriented computing (SOC) improves discovery efficiency by reducing search spaces, which is achieved by organising services into different groupings. Different approaches organise services according to different service attributes, such as location, or domain, which are captured in structures like overlays or hierarchies. However, approaches that organise services based on their attributes still create large search spaces where there are a large number of services, which is the case in smart cities. Moreover, the supporting structures do not provide enough information to drive the discovery process. For example, in a location-based approach, a request $r1$ might not get a response because service $s1$, which is relevant to $r1$, is in a different geographic area. In addition, overlays or hierarchical structures are static, but cities are dynamic and require continuous adaptation of their information systems. There are adaptive service discovery approaches that react to changes in service properties or the network topology, but they do not consider changes in the real-world environments with which services interact. Finally, current approaches to service composition are limited when constituent services need to be discovered from large IoT environments. Conversation-based approaches to service composition have good discovery accuracy but require high human intervention to define composition plans in advance. Interface-based approaches avoid human intervention but use expensive processes that affect discovery latency and accuracy.

This thesis introduces *uDiscovery*, a distributed urban-centric model to support adaptive service discovery, designed to be efficient and adaptable in smart city environments. *uDiscovery* organises service descriptions based on urban-context. Gateways in a city environment recognise their surrounding places and create search spaces only relevant for these places. This urban context also drives service discovery by forwarding requests to gateways where they are most likely to be solved. *uDiscovery* adapts the service organisation as the city evolves. Each gateway recognises different city events as they occur and reacts to them by moving services from other gateways. This self-adaptive organisation puts the right service at the right place at the right time, in preparation for discovery. *uDiscovery* also includes a planner that searches for services that constitute compositions. The planner uses consumers' feedback to drive a progressive search that improves both discovery latency and accuracy.

uDiscovery has been evaluated using a city simulation and an IoT test bed. Evaluation metrics include the discovery success rate, discovery accuracy, discovery response time, and the network overhead under varying number of services, and different mobility scenarios. Results present both the strengths and limitations of the proposed service discovery model. In general, *uDiscovery* outperforms baselines as it solves more requests with good accuracy and latency, at the cost of higher network overhead.

Acknowledgements

This thesis was made possible thanks to the support of many people. First, I would like to thank my supervisor, Professor Siobhán Clarke, for her guidance and for all the valuable advice in the research process. I would like to thank my colleagues Fan Li, Andrei Palade and Gary White. Thank you for all your help over these past years and the learning that we shared in the SURF project. Many thanks to all DSG members for their help and support. Thanks to CIT people for their useful advice. I am grateful for the financial support from Science Foundation Ireland, which has offered me the opportunity to study a Ph.D. in Dublin. I would like to acknowledge the Trinity Centre for High-Performance Computing (TCHPC) department funded by eINIS, for providing me with computational resources to run the experiments for this thesis.

Thanks a lot to all my family and friends. In particular, I would like to thank with my heart to my grandma Ligia, my mom Adela, my dad Humberto, and my siblings Catalina, Carolina and Camilo for all their support and love. I would like to thank my beloved Viviana for all the encouragement and support that I receive from her everyday. Last but not least, I would like to thank the responsible for the universe for everything.

Christian Humberto Cabrera Jojoa

University of Dublin, Trinity College

June 2020

List of Publications

Christian Cabrera, Andrei Palade, Gary White, and Siobhán Clarke. "An Urban-driven Service Request Management Model". In the International Conference on Pervasive Computing and Communications (PerCom 2020). IEEE, 2020.

Christian Cabrera, and Siobhán Clarke. "A Self-Adaptive Service Discovery Model for Smart Cities". IEEE Transactions on Service Computing, (TSC 2019).

Christian Cabrera, Gary White, Andrei Palade, and Siobhán Clarke. "Services in IoT: A Service Planning Model Based on Consumer Feedback". In the International Conference on Service-Oriented Computing (ICSOC 2018). Springer, 2018.

Christian Cabrera, Andrei Palade, Gary White, and Siobhán Clarke. "The Right Service at the Right Place: A Service Model for Smart Cities". In the International Conference on Pervasive Computing and Communications (PerCom 2018). IEEE, 2018.

Andrei Palade, Christian Cabrera, Fan Li, Gary White, M. A. Razzaque, and Siobhán Clarke. "Middleware for Internet of Things: An Evaluation in a Small Scale IoT Environment." Journal of Reliable Intelligent Environments. Springer, 2018.

Christian Cabrera, Andrei Palade, and Siobhán Clarke. "An Evaluation of Service Discovery Protocols in the Internet of Things". In the Symposium of Applied Computing (SAC 2017). ACM, 2017.

Christian Cabrera, Fan Li, Vivek Nallur, Andrei Palade, M.A. Razzaque, Gary White, and Siobhán Clarke. "Implementing Heterogeneous, Autonomous, and Resilient Services in IoT: An Experience Report". In the International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM 2017). IEEE, 2017.

Contents

List of Figures

List of Tables

List of Abbreviations

6LowPAN	IPv6 over Low-Power Wireless Personal Area Networks
AAL	Ambient Assisted Living
ACO	Ant Colony Optimisation
APFs	Artificial Potential Fields
ASPs	Autonomous Service Providers
BPMN	Business Process Model and Notation
CCN	Content Centric Networking
CoAP	Constrained Application Protocol
DHT	Distributed Hash Table
DL4J	Deep Learning for Java
DNS	Domain Name Server
DOI	Digital Object Identifier
DQN	Deep Q-Network
HTTP	HyperText Transfer Protocol
I/O	Input/Output
ICN	Information-centric Network
ICO	Internet Connected Object
IoT	Internet of Things
JSON	JavaScript Object Notation
MANETs	Mobile Ad-hoc Networks

MDP	Markov Decission Process
MQTT	Message Queuing Telemetry Transport
NDN	Named Data Networking
OSGi	Open Service Gateway Initiative
OSM	Open Street Map
OWL	Web Ontology Language
P2P	Peer to Peer
PHT	Prefix Hash Table
QA	Question Answer
QoS	Quality of Service
REST	Representational state transfer
RL	Reinforcement Learning
SOA	Service Oriented Architecture
SOC	Service Oriented Computing
UDDI	Universal Description, Discovery, and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
WSNs	Wireless Sensor Networks
XML	Extensible Markup Language

Chapter 1

Introduction

Smart cities are urban spaces, which offer advanced and innovative services to improve cities' sustainability and citizens' quality of life [Piro et al., 2014, Borgia, 2014]. The Internet of Things (IoT) has the potential to integrate a large number of IoT devices with urban spaces to realise the smart city vision. Such IoT devices can sense data and perform actions to support city services (e.g., a multi-modal transport service), and citizens' daily activities (e.g., a visit to a tourist place) [Al-Fuqaha et al., 2015]. The interaction between IoT devices and the physical world creates cyber-physical systems, which have enormous potential in different domains such as mobility, tourism, energy, safety, environment, and health care [Petrolo et al., 2016b]. Cities must manage the large number of IoT devices to realise the potential and offer efficient, highly available, reliable, and secure applications. Service oriented architectures (SOAs) are commonly used to manage IoT devices by abstracting devices' capabilities as IoT services [Teixeira et al., 2011]. Services are self-contained pieces of software that offer a functionality with a specified output (e.g., a service that reports the wind speed in the city centre). SOAs include processes to register, discover, compose, execute, and monitor services, enabling the development of flexible applications [Papazoglou and Georgakopoulos, 2003]. Service discovery is a key process in SOAs that locates relevant services for a given request, based on a description of their functional and non-functional requirements [Klusch, 2014]. Service discovery is also a fundamental requirement in IoT platforms because such platforms must locate devices and their capabilities before using them [Datta et al., 2015].

Service discovery has been widely explored in the web services domain. However, service discovery in smart cities presents different challenges. The IoT encapsulates millions of devices that communicate and enable new forms of interaction among things and people [Cirani et al., 2018]. This large scale generates enormous search spaces where efficient discovery is difficult. In addition, cities are continuously changing, which affects the interaction between different entities (e.g., an unforeseen closed street that affects pedestrians' mobility) [Tran et al., 2017]. The discovery process must maintain good performance over time, but it is challenging because cities are continuously changing, and a large number of IoT devices are deployed in large geographic areas. Citizens' requests are also likely to be complex and may require the combination of different services (i.e., service composition) [Lemos et al., 2016]. It is difficult to efficiently discover services that constitute a composition in large environments because there are likely to be multiple possible services combinations. In addition, the accurate discovery of such services combinations requires a considerable input from consumers, which is not feasible in large-scale IoT scenarios [Wang and Chow, 2016].

1.1 Challenges

IoT environments have significant challenges for service discovery [Fathy et al., 2018, Pattar et al., 2018, Tran et al., 2017]. In particular, service discovery in smart cities faces:

Challenge 1: A large number of services

The IoT connects millions of devices, which augurs well for the future of smart cities. However, cities must manage the large number of services that encapsulate these devices' capabilities [Fathy et al., 2018]. In particular, a discovery process will need to efficiently search for services in large environments [Pattar et al., 2018].

Challenge 2: Dynamic environments

Cities evolve over time because of the interactions between different entities (e.g., citizens, city places, etc.). Information systems that support a city must adapt to its evolution. Service management, including service discovery, must be adaptive to satisfy cities and consumers' changing needs [Fathy et al., 2018, Tran et al., 2017].

Challenge 3: Complex consumer requirements

Service consumers might have complex requirements that require service composition (i.e., a combination of existing services). It is difficult to discover the most appropriate set of services to constitute such compositions because of the potentially large number of available services [Pattar et al., 2018].

Challenge 4: Limited consumer input

Consumers can provide rich request descriptions as input to the discovery process in traditional environments. This input might help to improve the search accuracy by driving a more informed process. However, consumers are not likely to provide rich inputs in IoT as they will have limited knowledge about all available services in large environments [Wang and Chow, 2016]. Service discovery in IoT must be efficient with limited input from consumers.

1.2 Existing Solutions

Approaches to service discovery in IoT have proposed semantic methods to offer accurate discovery. Logic techniques are used to describe and match services [Perera et al., 2014b, Zhao et al., 2015, Quevedo et al., 2016]. Although these solutions offer good accuracy, latency is negatively impacted because ontologies must be queried to compare services and requests. Non-logic approaches [Cassar et al., 2014] offer better latency, but accuracy is affected because syntactic matching can introduce false positives. There is a trade-off between search accuracy and latency that needs to be balanced to offer an efficient service discovery. IoT environments also require methods to discover services that constitute compositions. Service planning mechanisms have been proposed to discover services as constituents of a service composition [Georgievski and Aiello, 2017]. However, current planning approaches either need high input from consumers to offer accurate search, or have performance issues when automating the process. Strategies to address the identified trade-off and current service planning approaches are discussed below.

1.2.1 Service Organisation in IoT Environments

Previous research has tried to balance the trade-off between search accuracy and performance by reducing the size of search spaces through sensible organisation of services [He et al., 2013, Fredj et al., 2014, Dasgupta et al., 2014a, Wang et al., 2015]. Such organisations create overlays or hierarchies, which are centred on service attributes such as service location [Andreini et al., 2011, Fredj et al., 2013, Wang et al., 2015, Lunardi et al., 2015], service type [Paganelli and Parlanti, 2012, Ebrahimi et al., 2017, Lee and Lee, 2018], or service domains [Dasgupta et al., 2014a, Bharti et al., 2018, Sivrikaya et al., 2019]. Requests are solved according to these structures, which are smaller search spaces than the whole set. These approaches are designed for smaller scenarios where scalability issues are not as significant as in a city. For example, a city can have a large number of services that belong to the weather domain, which means that large search spaces are still generated if services are organised by domain. Moreover, approaches based on service attributes do not provide enough information to drive the discovery process. For instance, in a location-based approach, a request $r1$ might not get a response because service $s1$, which is relevant to $r1$, is in a different geographic area. Finally, these approaches assume static environments that do not change. They produce an organisation of services' information that gets outdated, which then impacts discovery accuracy and performance. For example, an unforeseen flooding event might increase the demand for emergency services in a given city area. But, static architectures might not have information about emergency services in the flooded area. The complexity of searching for services in large environments lies in the strategies to organise services in distributed structures [Tran et al., 2017]. Approaches that organise services based on service attributes are limited and do not exploit the contextual information offered by cities. Solutions that use the city and citizens' information can offer more efficient service discovery [Zhou et al., 2016]. The discovery of services should also respond to a city's evolution, adapting the organisation of services information around the city [Tran et al., 2017]. In particular, the use of the spatial and temporal data of the city is a promising direction [Pattar et al., 2018].

Self-adaptive approaches have been proposed to handle dynamic IoT scenarios. Self-adaptation has been triggered by changes in the "thing" properties (e.g., battery level) [Albalas et al., 2017, Guerrero-Contreras et al., 2017] or network topology [Ebrahimi et al., 2017, Yuan et al., 2018]. These approaches do not consider the changes of the environments with which IoT services interact (e.g., cities). Self-adaptive discovery mechanisms are needed to support smart cities. These mechanisms should create IoT networks of devices that collaborate with each other to exchange data and services [Fathy et al., 2018].

1.2.2 Service Planning in IoT Environments

A composition process transforms consumer requests into the execution of a set of services [Lemos et al., 2016]. A service plan defines the services to be executed but the discovery of constituent services is challenging in IoT because of the large number of possible services in the environment [Cabrera et al., 2017]. Current planning approaches in IoT can be classified as conversation-based, or interface-based [Urbieta et al., 2017]. Service conversations use process models [Thiagarajan et al., 2002] to represent service plans, which are manually defined by consumers [Klein and Bernstein, 2004]. Such manual definition is not feasible in IoT because consumers cannot know about all available services. Interface-based approaches automate the creation of service plans, but incorrect services can appear, affecting search accuracy [Urbieta et al., 2017]. For example, a service $s1$ that provides readings about wind speed and a service $s2$ that consumes data of buses speeds are likely to match because the output type of $s1$ is equivalent to the input type of $s2$ (i.e., both are speeds). Search latency is also impacted by large search spaces and exhaustive exploration of all possible services combinations. Novel planning mechanisms for IoT should include historical data in progressive searches, with sensible ranking methods [Pattar et al., 2018].

1.2.3 Observations and Research Gap

There are two observations that drive this research. First, traditional approaches organise services according to their attributes in static structures. These structures do not provide enough information to support efficient service discovery and they can easily get outdated in dynamic IoT environments. Second, current service planning approaches are not suitable for smart cities because large environments affect their performance, and they require high input from consumers or have search accuracy issues. The efficient discovery of IoT services in smart cities remains an open challenge. Service discovery models for the IoT in large and dense geographic areas such as cities, should provide:

1. *Smart organisation of services.* Services should be organised in distributed architectures closer to the edge to provide wide coverage [Tran et al., 2017]. Such architectures should exploit city context to manage and adapt such massive IoT architectures [Zhou et al., 2016, Pattar et al., 2018, Fathy et al., 2018].

2. *Efficient Service Planning.* Service planning processes should minimise consumers' input and still provide good search accuracy. Service planning should include historical data in progressive searches [Pattar et al., 2018].

1.2.4 Research Questions and Hypothesis

Considering the large scale of IoT services in smart cities, the dynamic nature of cities, and the likely complex consumers' requests, this thesis explores *the question of to what extent can the use of city and citizens' context support an efficient service discovery in dynamic and large smart city environments.*

This thesis proposes two complementary hypotheses to respond to this question:

H.1 Urban context improves service discovery efficiency when it drives an adaptive and distributed organisation of services in large geographic environments.

H.2 Service planning based on consumers' feedback improves both search accuracy and latency.

These hypotheses lead to more specific research questions as follows:

RQ.1 To what extent can the use of urban context to organise services' information improve service discovery efficiency in the presence of a large number of services in smart cities environments?

RQ.2 To what extent can the use of urban context to adapt the services' organisation maintain or improve service discovery efficiency *over time* in dynamic smart cities environments?

RQ.3 To what extent can the use of consumers' feedback to search for services improve service discovery efficiency, and minimise human input when responding to complex consumer's requirements?

1.3 Thesis Approach

The following objectives are proposed based on the hypotheses and the state of the art analysis:

O.1 The organisation of services reduces search spaces at discovery time. Previous approaches use service attributes to group services but they do not exploit cities or citizens' information. This work aims to identify and formalise the contextual data that SOAs can use to support efficient service discovery in smart cities.

O.2 The inclusion of city and citizens' information requires architectures with capabilities to capture, process, and exploit this data to manage services based on this new context. This work's goal is to design a service model that uses the identified context to organise services information and drive an efficient discovery.

O.3 Citizens might have requests that require service composition. The discovery process must search for services that constitute such compositions. The large number of services in IoT environments negatively impacts the accuracy and latency of current approaches. This work aims to design a search model to support service composition in IoT environments based on a progressive search.

Assumptions

1. Cities are entities that offer services to citizens (e.g., transport services). These services are offered in different places and can be supported by software services that deliver useful information (e.g., bus station schedule).

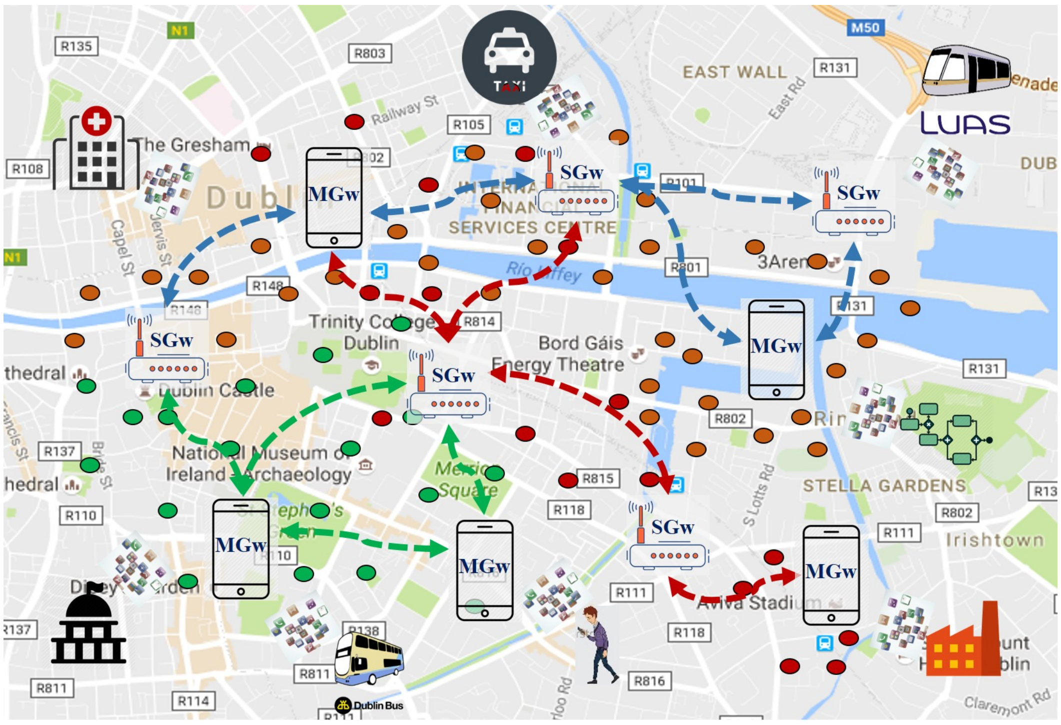


Figure 1.1: Smart City Environment.
MGW: Mobile Gateway, SGw: Static Gateway

2. Citizens are mobile through the city and can request software services to support their activities (e.g., a citizen may want to know which is the best transport mode to go home from the city centre).

3. Software services must be discovered and composed to meet user requests and integrate different providers. A network of IoT gateways covers the city and manages services and requests in a distributed fashion (Figure 1.1).

4. Gateways can be static or mobile. Static gateways (e.g., a dedicated device installed in a city point) have Ethernet and WiFi interfaces to communicate with other gateways, service providers, and service consumers. Mobile gateways (e.g., a device installed in a bus) have a WiFi interface to communicate with other gateways, service providers, and service consumers. Each gateway stores services descriptions in a local registry, and has access to information about its surrounding city places.

5. Services can be registered from different providers (i.e., web services, WSN services, and autonomous services[1]) and their descriptions are defined as $s_{desc} = \langle id, I, O, D \rangle$, consisting of a service identifier, inputs' types, outputs' types, and domains. For exam-

[1]We consider Autonomous Service Providers (ASPs) as providers that can decide when to act as providers (e.g., a smart phone configured by a user to offer noise level information about the environment, in a given time period), and that are likely to be mobile.

ple, a service that calculates a tourist path belongs to the tourist domain, consumes a string of tourist interests and produces a tourist path in XML format.

6. Each gateway can receive requests from consumers and search for services in the local registry to solve them. A consumer request is defined as $r = \langle I, O, D \rangle$, consisting of request inputs' types, outputs' types and domains. For example, a request for a tourist path calculator should include the tourist domain, string as input type and XML as output type.

1.4 Thesis Contribution

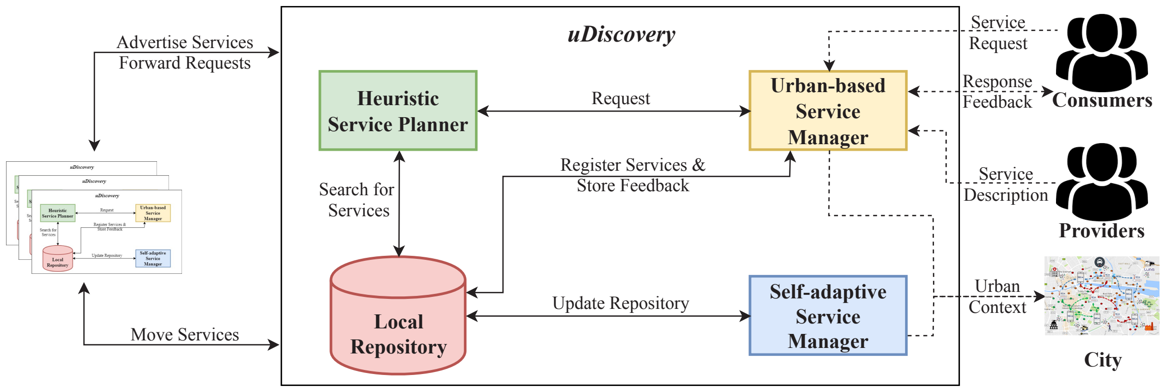


Figure 1.2: *uDiscovery* - High Level Architecture.

This thesis proposes *uDiscovery*, a distributed urban-centric model to support efficient service discovery in smart cities. *uDiscovery* enables gateways in the city to recognise their surrounding places and formalise their information as urban context. *uDiscovery* drives an adaptive discovery process by forwarding requests to search spaces where they are most likely to be solved. Figure 1.2 introduces the architecture of *uDiscovery*, which is composed by four main components. The service manager receives service descriptions from providers and either stores services information in the local registry or advertises it to other IoT gateways based on the urban context captured from a city. This component also receives requests from consumers. It tries to solve requests using a service planner that searches for services in the local repository based on consumers' feedback. If requests are solved locally, *uDiscovery* returns the response to consumers and waits for their feedback, which is stored in the local registry. Otherwise, *uDiscovery* forwards requests to other IoT gateways in the network. The self-adaptive service manager updates the local repository by moving services between IoT gateways according to city events.

This research contributes to the body of knowledge as follows:

- **Urban-based service discovery model**

  Existing work in service discovery does not exploit information offered by cities to manage services and drive the discovery process. They organise services based on simple services attributes such as location or domain, which might not provide enough information to support efficient service discovery in large IoT environments. *uDiscovery* uses urban information to support service discovery in smart cities. It puts information about the right service in the right place, in preparation for discovery. Service information is distributed according to urban places and their meaning in the city (i.e., urban context). The idea is that citizens' requests are influenced by where they are. For example, it is more likely that people near to a college ask for educational services. Then, gateways on or near to the college should store services for educational services. This model also allows smart replication of services information to improve its availability. For example, two colleges in different locations of the same city could share educational services information. The urban context is managed in a knowledge model that includes smart city concepts and is used to build semantic overlays of gateways at registration, and discovery time. This knowledge model is independent of the service discovery domain and can be used to solve different smart city issues.

- **Support for self-adaptive service discovery**

  Existing research organises services in structures that do not respond to changes in the environment where they work. Environment changes negatively impact discovery efficiency because they cause outdated structures that no longer have the required services information. *uDiscovery* responds to the city's evolution, adapting the distribution of services around the city according to city events. *uDiscovery* proposes a novel self-adaptive discovery model for smart cities that offers the right service in the right place, at *the right time*. Service information distribution evolves according to city events which may be unforeseen (e.g., flash flooding), foreseen (e.g., a cultural event), or periodic (e.g., city peak hours). *uDiscovery* formalises the temporal dimension of the city in the knowledge model and proposes mechanisms to identify and react to city events by exchanging services between IoT gateways.

- **Service planning based on consumers' feedback**

  Current approaches that search for services and support service composition in IoT either need high level of consumer's input or reduced knowledge implies the discovery process is likely to take longer in large IoT environments. Such environments require efficient planning methods with minimal input from consumers because consumers can not know about all available services. *uDiscovery* proposes a heuristic service planner that uses consumer feedback to improve search efficiency, minimising the level of consumers' input. Consumer feedback (i.e., historical data from previous searches) determines if a discovered plan was correct or incorrect and is used to improve search accuracy. The latency of the process is reduced using two strategies. First, the model explores only the most promising plans (i.e., search space reduction). Second, the model avoids wasting time on the less promising plans (i.e., progressive search).

1.5 Thesis Scope

uDiscovery uses urban context (i.e., city places, and city events information) to support adaptive service discovery. IoT environments that do not have the required urban context are out of the scope of this research. For example, indoor IoT scenarios such as smart buildings [Wang et al., 2015, Bovet and Hennebert, 2014] or ambient computing environments [Mokhtar et al., 2010, Görgü et al., 2017], and smaller networks of sensors (i.e., WSNs) [Butt et al., 2013, Perera et al., 2014b, Fredj et al., 2014]. However, *uDiscovery's* service planner can be used in domains other than smart cities to discovers services, which constitute compositions.

uDiscovery manages service which are described following a specific format. In particular, the management of heterogeneous services is outside the scope of this thesis. There are two alternatives that could enable *uDiscovery* to manage heterogeneous services: the development of modular mechanisms that support different data formats [Kovacevic et al., 2010, Görgü et al., 2017], and the semantic enrichment of services descriptions using machine learning techniques [Cassar et al., 2014].

uDiscovery searches for services following a goal-driven approach, which generates service plans that specify the set of services to be executed and its order. The actual execution and provision of these services are outside the scope of this research. Different approaches [Georgievski and Aiello, 2017, Palade et al., 2018] can support the execution and provision of service plans that *uDiscovery* creates. Alternative approaches to compose services [Boubiche et al., 2018] such as clustering or aggregation are also outside the scope as *uDiscovery* follows the goal-driven approach.

uDiscovery selects services from a functional perspective and does not take into account non-functional requirements (i.e., QoS attributes). These QoS attributes might also be variable and require special management, which is outside the scope of this work. Current research on QoS management and prediction [White et al., 2017] could enable *uDiscovery* to manage QoS information in IoT environments.

uDiscovery creates an open environment where different providers and consumers can expose sensible data. Security and privacy concerns with regard to these open environments [Kozlov et al., 2012] are out of the scope of this thesis. Different security mechanisms[Ammar et al., 2018] can be used to protect providers and consumers data. Moreover, novel policies from authorities are also needed to regulate such IoT environments.

1.6 Thesis Structure

The remainder of this thesis is organised as follows:

State of the art Chapter 2 analyses how state of the art service discovery approaches address the challenges of large-scale of services in dynamic environments. In particular, this chapter analyses a) how large number of services are managed, b) how this management responds to dynamic environments, c) the efficiency of the discovery process, and d) how consumers' requests are addressed when they require service composition.

Design Chapter 3 describes the design objectives, system model, and design decisions of this thesis according to the service discovery challenges outlined in Chapter 1. It explains how urban-context is extracted and managed. Then, the chapter presents how a network of gateways is initialised, maintained, and used to drive both the organisation of services and the discovery process. It also shows how the organisation adapts to the city simulated environment. Finally, the chapter provides details with regard to the planner that discovers service plans to support service compositions.

Implementation Chapter 4 describes the implementation of *uDiscovery* according to the design outlined in Chapter 3. It presents the structure, behaviour, and interactions of $uDiscovery's$ components.

Evaluation Chapter 5 evaluates how well *uDiscovery* achieves its objective of efficient service discovery in smart cities. It first describes the experimental set-up of the evaluation. The second part of the chapter presents and analyses of the results showing that *uDiscovery* is a suitable alternative for service discovery in smart cities.

Conclusion Chapter 6 concludes this thesis. It discusses $uDiscovery's$ contributions and limitations, and highlights areas for future work.

Chapter 2

State of the Art

This chapter reviews current research that explores service discovery in large and dynamic environments. Existing surveys analyse service discovery in web services [Klusch et al., 2015, Lemos et al., 2016], and IoT domains [Zhou et al., 2016, Bröring et al., 2016, Georgievski and Aiello, 2017, Tran et al., 2017, Pattar et al., 2018, Fathy et al., 2018]. This chapter extends these surveys by analysing to what extent existent work meets efficient service discovery in large and dynamic environments such as a smart city. This review follows the structure described in the Figure 2.1, which is based on the challenges identified in Section 1.1. This chapter reviews how large numbers of services are managed by analysing approaches' architectures and data management strategies. It then assesses how the works address consumers' requests and how they support service composition. Finally, this chapter reviews how approaches respond to dynamic environments.

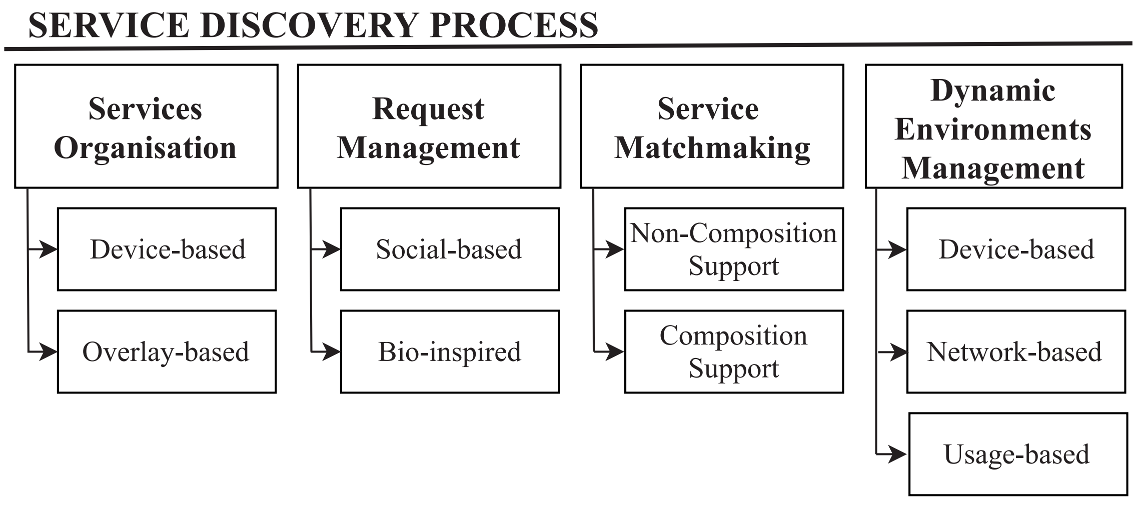


Figure 2.1: Structure of the State of the Art Review.

2.1 Services Organisation

How approaches manage services information has an important influence on the service discovery efficiency. Sensible organisation of service descriptions has the potential to enable efficient discovery by creating reduced and relevant search spaces, under a determined criteria [Zhou et al., 2016]. First approaches in web service discovery used centralised and decentralised directories (e.g., UDDI) to store and search services using WSDL or REST descriptions [Walsh, 2002, Klusch, 2014]. IoT environments challenges these web service discovery by creating large and dynamic environments, where the organisation of services plays a key role to offer efficient service discovery. Current research has proposed to organise service descriptions based on devices' attributes, or network overlays. Services with similar values for a given attribute in the service description are grouped, when services are organised based on devices' attributes (e.g., spatial attributes) [Fathy et al., 2018]. Approaches based on overlays create P2P networks based on distributed hash tables (DHTs). A DHT maps the search space to a numeric range and then allocates directories to parts of that range [Bröring et al., 2016]. Fully distributed architectures can organise services by creating communities according to nodes social properties (e.g., nodes' interactions) [Atzori et al., 2011]. These social-based approaches are analysed later in Section 2.2, as the data dissemination and forwarding are the key concerns in such approaches. There are approaches that organise services in adaptive structures that react to changes in devices' properties or network topology. These approaches are analysed in Section 2.4.

2.1.1 Device-based Organisation

Fredj et al. [Fredj et al., 2013, Fredj et al., 2014] propose a model to cluster and aggregate services according to the device's location. This approach relies on a hierarchical network of gateways that represent different smart spaces in indoor environments and host semantic service descriptions. Gateways are connected in a tree topology, where the lowest level represents spaces containing physical connected devices (e.g., a room). Upper-levels represent spaces that include lower-levels (e.g., floors or buildings). This hierarchy is used to create routing tables that determine requests' forwarding at discovery time. Consumers send requests that are solved by gateways at the lowest hierarchy level. COBASEN is a context based search engine for industrial IoT [Lunardi et al., 2015]. COBASEN organises devices according to their properties (i.e., location, type, purpose, status, and measurement units) using an inverted index to improve search response time. COBASEN searches for services in a centralised repository using key words. Consumers filter, configure, and define aggregations between IoT resources after they are discovered. SMARTSPACE [Dasgupta et al., 2014b] is a distributed multi-agent middleware. This middleware is distributed into a system of federated registries where a service discovery algorithm searches for services. This algorithm creates clusters of services that have similar functional features by comparing their signatures (i.e., input and outputs parameters). SMARTSPACE encodes each cluster in a hierarchical structure that agents use to forward requests through registries and reduce search spaces. Wang et al. [Wang et al., 2015] propose a geospatial index for sensor discovery. Sensors deployed on buildings are grouped using binding boxes and R-tree indexes. GeoNames and indoor location ontologies are used to semantically describe sensors. Consumers send queries, which include consumers' requirements and spatial information. This spatial information is used to select gateways that contain information about sensors that meet consumers' queries. Then, queries are forwarded to the selected gateways, which perform a semantic search and send the response to consumers. Jo et al. [Jo et al., 2015] propose to distribute services information in a hierarchical structure managed through bloom filters to support mobile IoT environments. Bloom filters reduce the configuration cost and the number of exchanged messages to update information about mobile services. A hierarchy for each capability in the environment is built that starts from high level tasks in upper levels to resources capabilities in low levels. Bloom filters encode each level in the hierarchy in a bit array that is used to discover services.

Fathy et al. [Fathy et al., 2017] present a spatial indexing approach to manage data in the IoT. This model organises IoT resources in a distributed architecture by encoding their location on geohashes that group resources that are geographically close. Geohashes are one-dimensional representations of two-dimensional spatial coordinates, which create a hierarchical structure that divides the geographic areas recursively into bounding boxes until the required resolution is achieved. Adjacent locations share a similar prefix in their geohashes, which simplifies searching by spatial attributes. Hoseinitabatabaei et al. [Hoseinitabatabaei et al., 2018] propose an indexing approach for IoT lookup. Each IoT source attribute (e.g., sensor type or sensor location) has a mathematical representation that is easy to maintain and provides enough information for the search process. Indices support exact queries by type (e.g., get temperature) and location (i.e., coordinates). The search process retrieves the list of gateways that manage sensors of the required type in the specified location. Lee and Lee [Lee and Lee, 2018] propose a model to cluster IoT services. This model groups similar services based on their attributes. Four criteria are applied to classify services: the sensor properties criterion (i.e., power, transmission, and operation), the data management criterion (i.e., pre-processing, data store, transmission and trust), the data processing criterion (i.e., parallel, analysis models, and manipulation models) and the execution criterion (i.e., post-processing). This model has a graphic interface where users specify the values for each criteria to classify services. Bharti et al. [Bharti et al., 2018] propose a resource discovery model that groups contextual information into clusters to support efficient search. This model organises services based on resources metadata that includes object identification, availability and computation capability. It applies different similarity measures coefficients (i.e., Cosine similarity, Dice similarity, Euclidean similarity, and Jaccard similarity) to create disjoint sets of similar objects. The approach analyses requests to extract input parameters, match them, and determine the list of similar clusters where to search. The Internet of Smart City Objects (ISCO) project [Sivrikaya et al., 2019] proposes a framework to connect smart objects from different domains through an open and extensible platform. This platform includes a distributed service directory that organises services in an information-centric networking (ICN) overlay. Each directory contains smart city objects which are semantically described by city domains, devices' types, and non-functional attributes such as location. This semantic model aims to reduce the amount of returned objects for discovery and planning operations, by grouping objects according to their attributes. This overlay is used at registration and discovery time to replicate service descriptions, solve requests.

2.1.2 Overlay-based Organisation

Paganelli et al. [Paganelli and Parlanti, 2012] present a DHT-based approach for service discovery. This approach distributes services according to their functionality using a prefix hash tree (PHT). This structure relies on the DHT lookup operation to insert and retrieve service descriptions. This approach handles multiattribute and range queries, which are solved by exact matching. Chord4S [He et al., 2013] is a P2P approach that supports QoS-aware service discovery. Chord4S distributes services' information according to their functionality. Similar services are stored in different successor nodes in the chord ring. Queries for a given service are forwarded to the relevant set of successors nodes. Services that meet QoS requirements are selected once they are retrieved from the ring.

Nguyen et al. [Nguyen et al., 2017] propose a discovery model based on Chord to manage heterogeneous IoT resources. This model manages heterogeneous services by creating a structure composed of multiple Chord rings. Each ring represents an IoT context that organises services' descriptions. Each node in the ring represents a gateway that manages a group of sensors. Lookup operations are used to locate devices in a ring. Multiple rings can share nodes to enable search between rings. Tanganelli et al. [Tanganelli et al., 2017] propose a service lookup approach based on a DHT that supports multiattribute and range queries. It organises services in independent DHT clusters where each cluster represents one specific attribute. This approach forwards queries to the respective cluster according to the search attribute. Each node performs exact matching to solve queries.

2.1.3 Assessment

Approaches that organise services based on their attributes (e.g., location, or domain) create reduced search spaces in small IoT environments such as smart buildings. However, these search spaces can be still large in IoT environments. For example, a city can have a large number of services that belong to the weather domain, or provide the same functionality. Moreover, approaches based on service attributes do not provide enough information to drive the discovery process. For instance, in a location-based approach, a request r_1 might not get a response because a service s_1, which is relevant to r_1, is in a different geographic area. IoT environments such as a smart city provide rich contextual information that might be used to support more efficient service management. Service discovery architectures should exploit city and citizens' spatial and temporal information to manage massive and dynamic IoT environments [Zhou et al., 2016, Pattar et al., 2018, Fathy et al., 2018]. The inclusion of such context has potential to enable more informed architectures, where digital entities can make better decisions to create relevant search spaces and drive the discovery process.

Approaches based on overlays rely on distributed hash tables properties to provide highly scalable and dynamic architectures. They organise services in DHT nodes according to different service attributes and propose multiple rings to describe services from different perspectives. Service discovery is based on the look-up function provided by DHT implementations such as Chord [Stoica et al., 2003] or Kademlia [Maymounkov and Mazieres, 2002]. This look-up function is limited to exact string matching where consumers know in advance the name of the key [Nguyen et al., 2017] to retrieve information. This look-up mechanism does not suit IoT environments where consumers only know their needs, but ignore the services that can satisfy them because consumers must specify the desired key. For example, look-up mechanism cannot support goal-oriented scenarios where consumers specify what they expect (i.e., needed data as outputs), and what they have (i.e., available data as inputs). Moreover, searches by exact matching can affect accuracy, by introducing false positives in search outputs.

2.2 Request Management

Service discovery architectures should exploit the organisation of services to offer efficient discovery. It is impractical to scan all the service descriptions of all service repositories in a distributed environment [Zhou et al., 2016]. Consumers' requests must be forwarded to search spaces where they are most likely to be solved. WSNs and MANETs domains have been widely studied with regard to data dissemination. Broadcast, multicast, flooding, and gossiping-based approaches have been proposed for small networks of sensors or devices [Chakraborty et al., 2006, Nedos et al., 2009, Chen and Clarke, 2014]. However, the scale of IoT environments require strategies that cover wide geographic areas, and minimise the resources usage (e.g., network bandwidth and devices' battery) [Zikria et al., 2018]. Current research on IoT service discovery has enriched WSNs and MANETs mechanisms, by creating architectures that manage consumers' requests based on social structures or bio-inspired methods. Approaches based on social structures use the social relationships between IoT devices to forward discovery messages to communities where they can be solved [Atzori et al., 2011]. Bio-inspired approaches take ideas from nature to drive the requests' resolution. They take advantage of the inherent self-managed and optimised behaviour of different natural systems to address IoT challenges [Hamidouche et al., 2018].

2.2.1 Social-based Forwarding

Loser et al. [Loser et al., 2007] introduce a routing algorithm that considers each node in the network as a person in a social network. Each peer has information about other peers in the network according to previous interactions. This information is used to forward queries to relevant peers when they cannot be solved locally. Girolami et al. [Girolami et al., 2015b] propose CORDIAL, an algorithm for service discovery in mobile social networks. CORDIAL creates a distributed social network of smart phones according to human periodic movements and interests. CORDIAL forwards requests and advertises services in communities that are identified based on these movements and interests. It relies in the assumption that nodes with similar interests tend to meet more frequently than nodes with non-overlapping interests. TSSD is a model for service discovery in mobile networks that exploits the temporal-spatial correlation between nodes [Li et al., 2017]. This model creates communities based on the contact time and the frequency of contacts between nodes. These communities evolve over time to reflect nodes interests. TSSD forwards requests in these communities with an epidemic routing that spreads queries in intermittent mobile networks.

Hussein et al. [Hussein et al., 2017] propose an approach for dynamic service discovery in the social internet of things. This approach creates a temporal-social structure that combines users, objects and services. A reasoning mechanism uses this structure to discover services that can meet users goals.Li et al. [Li et al., 2017] present a decentralised semantic-based service discovery framework that uses social links between IoT entities to forward services or recommendations. This model propagates recommendations based on a "usefulness score" that represents the reputation of the recommendation and the advertiser (i.e., trust), the closeness of the recommendation with the target node (i.e., interest similarity), the reliability of the recommendation (i.e., QoS), and the freshness of the recommendation. Corbellini et al. [Corbellini et al., 2017] propose to mine web service repositories to support service discovery. This approach creates clusters that group services according to users' interests. This model calculates similarity between services using neighbour-based metrics, which assume that two nodes are more likely to link if they share neighbours. This structure is used to forward requests to the more relevant web servers. Xia et al. [Xia et al., 2019] propose a social and semantic service discovery mechanism that mimics human-like social behaviour. This approach implements an adaptive strategy that forwards requests to a selected subset of devices. This subset is selected according to a correlation degree that reflects how relevant is a device to a given query. The correlation degree calculates the semantic similarity between services and requests based on an ontology tree.

2.2.2 Bio-inspired Forwarding

Ebrahimi [Ebrahimi et al., 2015] present an approach that clusters sensors' information using an ant-based algorithm. This approach creates a semantic overlay based on sensors' type. Ants walk through this semantic structure and create clusters based on sensors' context (i.e., QoS attributes). This approach extracts required attributes from consumers' queries and forwards them to the most relevant clusters according to users' priorities. Rapti et al. [Rapti et al., 2016] propose a decentralised service discovery approach based on artificial potential fields (APFs). Each service provider has an APF, and its strength depends on the percentage of requests that the provider solved in the past. Each provider applies attraction or repulsion forces to consumers' requests according to the APF to mimic electrically charged particles. These forces drive requests to the most promising providers.

Wanigasekara et al. [Wanigasekara et al., 2016] introduce a discovery approach based on usage patterns to support service composition. Service discovery is modelled as a contextual bandit problem. Services are the set of bandits, the reward is based on the service usage (i.e., positive reward if the user selects the service), and the expected pay off represents how many times the service was successful when selected. The approach maximises the expected pay-off by recommending relevant IoT resources based on crowd-sensed information. Yuan et al. [Yuan et al., 2018] propose a self-organised social network with a swarm-based service discovery. They build an immutable social overlay according to the interaction of the nodes in the network (i.e., friendship). A score is calculated for the associations between nodes, which is updated through the swarm mechanism according to historical searches, to reflect changes in the network. The swarm mechanism provides an adaptive request forwarding that uses the scores to discover the shortest paths to a desired service.

2.2.3 Assessment

Service discovery based on social networks create architectures that suit mobile IoT environments because they take advantage of interactions between devices. These architectures mimic human social features to create communities based on devices interests, reputation, and mobility patterns. These approaches use this information to advertise services and forward requests in dynamic peer to peer networks by using flooding-based strategies. These strategies reflect temporal and spatial dimensions from human behaviour, but fail to capture more complex social relationships that are still difficult to identify, manage, and measure [Girolami et al., 2015a]. It impacts the dissemination efficiency because flooding-based strategies with limited knowledge are likely to fail when delivering messages in dynamic networks, where sources and targets might never have a direct link between them. Requests need more informed and efficient diffusion strategies to be forwarded to relevant devices in short time. More information about the environment (e.g., city context) provide knowledge to make better decisions when forwarding requests. Efficient mechanisms must exploit such knowledge even in environments when it is limited.

Bio-inspired approaches address IoT complexity by taking inspiration from natural systems that also work in complex environments [Hamidouche et al., 2018]. The bio-inspired strategies to propagate information in distributed networks are promising mechanisms to explore with regard to request forwarding in the service discovery problem. These approaches can propagate information to desired destinations in an effective fashion even when nodes have a limited knowledge about other network participants. Such approaches should be integrated with the contextual information from IoT environments to create mechanisms that can make well informed decisions. This thesis uses a bio-inspired method together with the city context to drive the discovery process and forward requests, where they are most likely to be solved.

The proposed approach takes advantage from the properties that enable efficient propagation of information in large and dynamic networks, and provides more knowledge to these properties by adding city context.

2.3 Service Matchmaking

Service discovery latency and accuracy depends on the matchmaking mechanism. This mechanism is responsible for comparing requests with services' descriptions to select the set of services that satisfy consumers' needs [Klusch, 2014]. Service matchmaking mechanisms can be classified according to their ability to discover services that constitute composition of services (i.e., no-composition support or composition support). Matchmaking mechanisms do not support service composition when they search for services by a given attribute (e.g., sensor type). These mechanisms use either non-logic (i.e., syntactic) or logic (i.e., semantic) methods to select atomic services. Service matchmaking mechanisms support composition when they discover the services that constitute the composition [D'Mello et al., 2011]. These mechanisms can be classified as non-automated or automated [Urbieta et al., 2017]. Non-automated approaches need high human input to define composition plans at a high level of abstraction (i.e., service conversations). Automated approaches discover chains of services that meet user requests without human intervention [Klusch et al., 2016].

2.3.1 Non-Composition Support

Non-logic based approaches

Mobile Digcovery [Jara et al., 2014] is a distributed discovery platform that interacts with heterogeneous services using different technologies such as IPv6, 6LowPAN, EPCGlogal, Digital Object Identifier (DOI), and Legacy Devices. Mobile Digcovery uses the elasticsearch search engine to offer scalable search and includes contextual information in queries (e.g., service coordinates, service type). The search engine uses a string-based matching to compare services' descriptions with queries. Cassar et al. [Cassar et al., 2014] propose a matchmaking method based on machine-learning techniques. Probabilistic Latent Semantic Analysis and Latent Dirichlet Allocation are used to extract latent factors from semantic service descriptions and search for services in a latent factor space. This space contains heterogeneous service descriptions represented as a probability distribution over latent factors. Simurgh [Khodadadi et al., 2015] is a framework for service discovery in the IoT. JSON documents are used to describe services. These documents include information about location, service signature, and domains. Simurgh implements keyword comparison between consumers' requests and service attributes to search for services.

Petrolo et al. [Petrolo et al., 2016a] propose a mechanism to discover Internet Connected Objects (ICOs) in the Cloud of things. Each ICO is described by position, observed phenomena, and type. The discovery uses this description to match consumers' requests by performing syntactic comparisons. Stolikj et al. [Stolikj et al., 2016] propose a discovery protocol for resource constrained environments that store information about sleeping nodes in more powerful nodes, which makes their services available when they are off-line. Context tags describe services with additional information. This approach uses syntactic matchmaking between consumers' requests and the contextual data to find the relevant services. Han and Crespi [Han and Crespi, 2017] propose a model for service provisioning of smart objects in 6LowPAN platforms. This model integrates these platforms with the traditional internet by using CoAP severs that provide resource descriptions, which can be accessed through HTTP requests. Service discovery is also based on CoAP and compares contextual information such as sensor type, and coordinates.

Logic-based approaches

CASSARAM is a context aware search model that selects IoT sensors [Perera et al., 2014b]. CASSARAM uses an ontology to describe sensors' properties such as location, accuracy, reliability, and battery life. Consumers' requests include functional and non-functional requirements that are matched with services' descriptions using the ontology model. CASSARAM ranks selected sensors using similarity metrics between queries and sensors data. CADDOT is a service model that integrates 'things' with cloud-based IoT solutions [Perera et al., 2014a]. CADDOT describes sensors using an ontology, which includes the sensor's capabilities and communication technologies. CADDOT uses this information to discover and configure heterogeneous services. Zhao et al. [Zhao et al., 2015] propose a multidimensional model to describe IoT sensors. Each dimension uses an ontology that represents a perspective from where resources are described (e.g., measurement principle ontology, location ontology, and domain ontology). A semantic matchmaker uses these ontologies to calculate resource similarity from each dimension. Users aggregate results from each dimension according to their preferences, to select relevant resources.

Quevedo et al. [Quevedo et al., 2016] propose a mechanism to support flexible service discovery based on Named Data Networking (NDN) and Content Centric Networking (CCN). NDN and CCN propose a communication model driven by consumers' information. Consumers send interests (i.e., requests) which includes semantic descriptions of desired services. A service broker forwards interests to a matchmaker that estimates the semantic measurement between interests terms and available service descriptions. This semantic matchmaker also includes matching based on syntactic methods (i.e., cosine similarity and Jaccard index). Rubio et al. [Rubio et al., 2016] propose an approach to discover subsystems in smart cities (e.g., smart homes or smart buildings). Subsystems and services are described using ontologies that allow the expression of context, profile, and process information. Subsystems and services can be discovered using the ontologies and SPARQL queries.

2.3.2 Composition Support

Non-automated approaches

Lee et al. [Lee et al., 2007] propose a smart space middleware to handle service composition in small dynamic environments. This middleware offers an interface where users can specify composition plans according to services available in the environment, and a plumber service keeps the composition optimal at execution time, utilizing OSGi's support. Tzortzis and Spyrou et al. [Tzortzis and Spyrou, 2016] propose a semi-automatic approach for semantic IoT service composition. The user starts specifying a service request with the expected outputs. The approach shows the list of available services according to the search parameters (i.e., expected outputs) and the user selects the best option. The process is repeated until the composite service meets the input parameters. Ciortea [Ciortea et al., 2016] model things as agents to support decentralised, responsive and flexible composition of service mash ups. Agents offer capabilities and interact between them to pursue a given goal. Relations between the agents, as well as the set of goals and sub goals to achieve are manually defined by users as pre-compiled plans. Huber et al. [Huber et al., 2016] present a framework to discover composed services using PROtEUS. This approach extends the PROtEUS process meta-model to express models for dynamic composition in Ambient Assisted Living (AAL) environments. Consumers specify a service conversation using this metamodel and SPARQL queries to get the set of services that realise the conversation. Baek et al. [Baek and Ko, 2017] present an approach to discover composed services in the user's vicinity to provide good user experience and a stable communication between services. Functional requirements are provided by users as a task that defines the services to compose. Available services are selected according to the input and output relations defined in the task.

Deng et al. [Deng et al., 2017] propose an architecture to optimise mobile composite services, when consumers and providers are dynamic. Consumers specify the composed plan as a set of tasks and relations (i.e., a service conversation). This plan is used to select services to achieve compositions with the shortest response time. Baker et al. [Baker et al., 2017] propose a service composition algorithm for cloud-based applications, to minimise energy consumption. Cloud-providers must define service conversations for each offered service, using the BPMN language. Consumers specify the desired functionalities (i.e., input and output parameters), which are matched with the service conversations to find the relevant one, with the minimal energy consumption. Urbieta et al. [Urbieta et al., 2017] present a context-aware service composition model for smart cities based on wEASEL [Urbieta et al., 2015]. wEASEL is an abstract model to represent services and users tasks as conversations. Consumers use this language to define conversations and the composition model searches for the available services that realise each conversation task. Semantic methods are used to match services and tasks by their input and output parameters. Zhao et al. [Zhao et al., 2017b] introduce a service composition approach based on user preferences. Consumers must define the set of tasks to be composed as a service conversation. The approach selects the appropriate service for a given task in the conversation based on previous learning about user preferences. These user preferences are not defined by users, but extracted from past service selection history.

Automated approaches

Zisman et al. [Zisman et al., 2013] propose a model to support web service composition at run time. This approach searches for services that replace services in the composition that are no longer available or fail to satisfy consumer requirements. It proposes two modes to discover services at run time. The pull mode searches for services when a replacement is needed. The push mode proactively identifies candidate services that can replace a service in a given composition. Service matchmaking compares QoS attributes of services by using non-logic methods in both modes. Rodriguez-Mier et al. [Rodriguez-Mier et al., 2016] propose a web service discovery and composition framework based on input and output parameters. This approach generates a graph-based composition, which contains the set of services that are semantically relevant for a given request. It also includes an algorithm that selects the composition, which minimises the number of services. A composition graph is created by exploring semantic relations between the parameters (i.e., inputs and outputs) of available services. Liu et al. [Liu et al., 2016] propose an agent-based approach to discover services in smart cities. Management Agents (MA) maintains a list of available Services Agents (SA). When a MA receives a request, it executes a local service discovery process, searching for SAs that can meet the request in the services' list. This search is based on non-logic approaches that calculate the similarity between the query, and services. If the MA does not have SAs to solve the request, the MA forwards the query to neighbours according to an estimation matrix that represents the historical availability of SAs, and MAs. CASCOM is a Context-Aware Sensor Configuration Model to simplify IoT middlewares' configuration [Perera and Vasilakos, 2016]. This platform includes a sensor selection process according to user requirements and contextual information. Users' requirements are gathered using a Question-Answer (QA) model. The selection uses an input and output string-based matchmaking to compose services from individual sensors. Selected services are ranked according to contextual information (i.e., sensor location, and battery life).

Chen et al. [Chen et al., 2016] present GoCoMo, a goal-driven service composition approach for pervasive computing. This approach supports an adaptive service composition and execution based on a distributed overlay. Service discovery is based on a backward planning algorithm where each node in the overlay performs a semantic matchmaking based on service parameters that are defined in the consumer request. Zhao et al. [Zhao et al., 2017a] introduce an energy-aware service composition mechanism for WSNs. Sensor nodes are represented as WSN services, which are grouped into service classes according to their functionalities. These service classes are chained using a forward planning algorithm to fulfil user goals (i.e., input and output parameters). The resulting composition plan is used to search individual WSN services according to energy, spatial, and temporal constraints.

2.3.3 Assessment

Service matchmaking is the core process of discovery approaches, as it is responsible for the actual search by comparing consumers' requests with services' information in the repositories. Non-logic based approaches have a low latency because they compare strings using syntactic methods. However, these methods can introduce false positives and impact search accuracy [Cassar et al., 2014]. Logic-based approaches offer a good accuracy because search processes have more information (i.e., semantic models) to select relevant services. However, these methods have a high latency as the comparison of services is complex because of the inclusion of these semantic models. There is a trade-off between non-logic and logic matchmaking methods which needs to be balanced in IoT environments where service discovery must provide accurate responses in short time.

IoT environments also demand the creation of added-value services, from existing services [Lemos et al., 2016]. Approaches that support service composition without automation increase search accuracy, at the cost of requiring that queries and services are modelled in a formal way. A consumer request must include a composition plan in advance, and tasks in the plan are compared against available services to select the relevant set [Klein and Bernstein, 2004, Mokhtar et al., 2007]. These approaches are time-consuming, error-prone, and require high human intervention [Zhao et al., 2017b]. It is not feasible to assume that users will know about all the available services and all their possible combinations in large-scale environments such as that expected for the IoT. Automatic approaches avoid human intervention. However, search precision can be affected negatively by the retrieval of incorrect services because the information used to perform the matchmaking is minimal [Urbieta et al., 2017]. Incorrect services can be easily introduced in services plans causing false positives in the search. For example, a service s_1 that provides readings about wind speed and a service s_2 that consumes data relating to bus speeds are likely to match because the output type of s_1 is equivalent to the input type of s_2 (i.e., both are speeds). Current matchmakers cannot distinguish between two services that can appear to be related because of their I/O parameters, but belong to unrelated domains. Both non-automated and automated approaches have performance problems in terms of response time as they perform complex processes in full search spaces and look for all the possible paths even when they are incorrect [Chen et al., 2016, Urbieta et al., 2017]. Novel composition mechanisms for IoT should include historical data in progressive searches, with sensible ranking methods [Pattar et al., 2018]. Knowledge about previous searches, and sensible rankings have the potential to drive more efficient search processes, by limiting the services exploration to the most promising sets.

2.4 Dynamic Environments Management

Service discovery architectures in the IoT work in dynamic environments that are changing all the time because of the interaction between different entities. There are service discovery approaches that respond to these changes and adapt their behaviour accordingly [Tran et al., 2017]. They can be classified as device-based, network-based, or usage based approaches. Device-based approaches respond to changes in devices' attributes (e.g., battery level). Network-based approaches respond to changes in the network properties. Usage-based approaches respond to changes in the users' demand.

2.4.1 Device-based Adaptation

Guerrero-Contreras et al. [Guerrero-Contreras et al., 2017] propose an architecture to support availability of services deployed in mobile and dynamic environments. It adapts the availability of services in mobile clouds environments according to the energy level of each node. Replicas of nodes are created when a given node is not reachable or does not have enough power, and nodes are hibernated when their services are not used. Albalas et al. [Albalas et al., 2017] propose a protocol based on CoAP to update resource directories according to battery consumption. Each sensor sends updating messages to the directories periodically. The period between messages is adaptable as time passes and the level of battery decreases, following the Fibonacci series. This approach is focused on battery consumption optimization rather than on sensor search efficiency.

Wu et al. [Wu et al., 2015] propose an adaptive multilevel index for service discovery in disaster zones. This solution stores services in a centralised repository using different index levels to improve search latency. These indices organise services by functionality computing the equivalence between their input and output parameters. The index structures are adapted according to the number of services in the repository (i.e., fewer services, fewer indices). Sikri [Sikri, 2019] proposes a self-managed web service framework to discover services in enterprise set-ups. This framework updates QoS attributes related to redundant services according to QoS measurements from services and consumers. This framework selects services with minimal cost based on QoS constraints specified by consumers.

2.4.2 Network-based Adaptation

Trendy [Butt et al., 2013] is a discovery protocol that groups services according to their location. It has a Directory Agent (DA), which maintains the registry and receives user requests. Group Leaders communicate with the DA to register and update information about services. The DA has a timer that determines how often the status of services must be updated. This timer adapts according to service demand. This adaptation process reflects requests' behaviour, to improve network efficiency. Cirani et al. [Cirani et al., 2014] propose a self-configuring architecture to provide automated service discovery. This architecture is based on a DHT structure that covers large geographical areas. Services are organised according to their location and the DHT structure adapts when a gateway joins or leaves the network. Each gateway uses CoAP-SD and DNS-SD to discover services locally. CoAP automates the registration of services through advertisements.

del Val et al. [del Val et al., 2014] propose an agent-based self-organised approach to manage and discover atomic services. Each agent in the system offers a service and can trigger adaptation processes under two circumstances. First, an agent can change the network topology (i.e., relations with other agents) according to the request's resolution and forwarding. Second, an agent can change the population of agents by cloning itself when there are many requests for its services. Ebrahimi et al. [Ebrahimi et al., 2017] propose a context-aware model that clusters sensors into semantic overlays. It groups sensors according to their domain and QoS attributes, using an ant-based algorithm. Ants calculate the similarity between two services and the probability of belonging to the same cluster. Clusters are adapted when the network topology changes. New sensors are assigned to the most similar cluster and the clustering process is triggered when a given number of sensors disappear. Yuan et al. [Yuan et al., 2018] propose a self-organised social network with swarm-based service discovery. They build an immutable social overlay, according to the interaction of the nodes in the network (i.e., friendship). This approach calculates a score for the associations between nodes, which is updated through the swarm mechanism according to historical searches. Changes in the network, when nodes appear and disappear, are reflected.

2.4.3 Usage-based Adaptation

Kumar and Satyanarayana [Kumar and Satyanarayana, 2016] propose a self-adaptive service discovery model for web services. It is based on a semantic classification that uses web logs to categorise services according to usage patterns. Web logs are analysed off-line in a learning phase, and the identified patterns are used to annotate services. The approach uses these semantic annotations to compute the relevance of web services for user queries. This relevance changes according to historical usage of services. Athanasopoulos [Athanasopoulos, 2017] proposes a web service organisation schema based on service functionalities. This schema adapts over time according to the pragmatics data (i.e., historical usage of services). Services are organised into hierarchical groups based on their descriptions (i.e., providers' perspective). This structure adapts according to the historical usage (i.e., consumers' perspective) which is used to calculate the similarity between services.

2.4.4 Assessment

Self-adaptation in current approaches is triggered by changes in service properties, the network topology or services' usage. They do not consider external changes to the context where approaches work. Changes in IoT environments influence consumers' needs and therefore must be considered when adapting service discovery architectures. For example, an accident in a city street might cause an unexpected increment in requests for emergency services. Service oriented architectures must be prepared to respond to such requests in an efficient fashion. Service-oriented adaptive models based on web services usage need previous training (i.e., off-line learning) to identify and exploit usage patterns. However, off-line mechanisms do not suit IoT environments, because, unlike web services, IoT environments are highly dynamic and require on-line adaptation to respond to city changes in short time. Moreover, web service adaptive models work in centralised architectures which represent an unique point of failure and scalability issues in IoT.

The adaptation of service information should respond to citizen needs and a city's evolution in a smart city [Zhou et al., 2016]. A new self-adaptive service model that formalises and responds to the dynamic nature of the city and their citizens should be developed. This model should create a large-scale and distributed IoT network where different entities can work and collaborate with each other to share and exchange services [Fathy et al., 2018].

2.5 Summary

This chapter analysed current research in service discovery from the perspective of large and dynamic smart city environments. The most related research with regard to services organisation includes approaches based on devices' attributes [Wang et al., 2015, Sivrikaya et al., 2019], and DHT overlays that group services according to their domains and functionalities [Paganelli and Parlanti, 2012]. The most related work regarding to request forwarding is the proposed by Yuan et al. [Yuan et al., 2018], which combines social networks and bio-inspired methods to disseminate requests information. The most related research with regard to service matchmaking includes an interface-based [Chen et al., 2016] and a conversation-based [Urbieta et al., 2017] algorithm. Finally, the most related adaptive approach is the proposed by Wu et al. [Wu et al., 2015], which adapts service organisation according to the number of services in disasters zones.

Figure 2.2 presents to what extent most relevant approaches satisfy a set of criteria defined from the thesis challenges (Section 1.1). The figure shows that approaches that manage services in large smart city environments [Paganelli and Parlanti, 2012, Wang et al., 2015, Sivrikaya et al., 2019] organise these services based on device attributes and do not exploit the context offered by smart cities. They create structures that adapt according to devices' properties but do not respond to cities' evolution, which might affect discovery efficiency because of outdated architectures in dynamic IoT environments. They implement semantic matchmaking to discover composed services in the best case [Sivrikaya et al., 2019]. These semantic processes might present search latency issues because the search spaces created by these device-based structures can still be large. For example, a smart city can still have a large number of services that belongs to a given domain or provide certain functionality. Yuan et al. [Yuan et al., 2018] use social-based and bio-inspired strategies to propagate

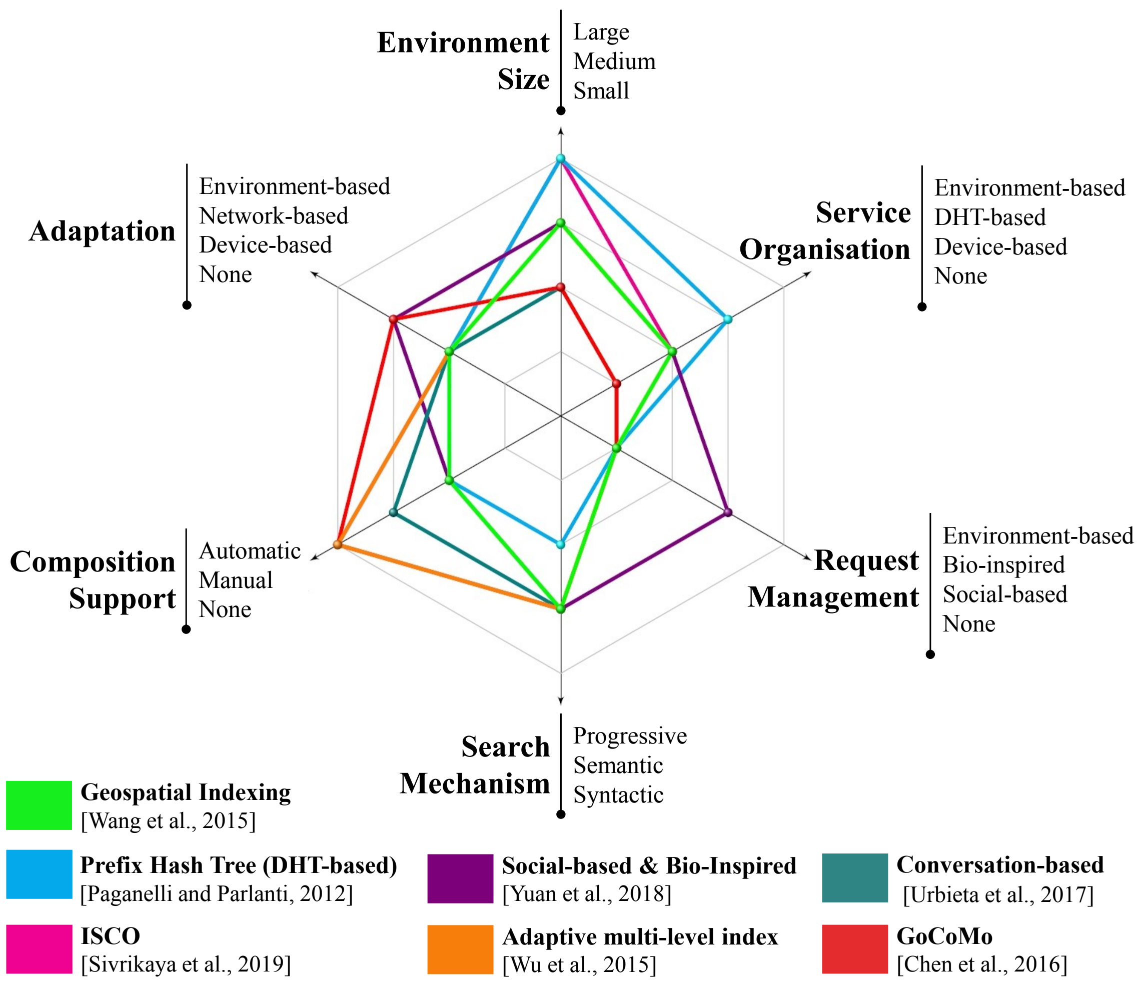


Figure 2.2: State of the art review diagram.

requests in fully distributed networks that form communities. This approach discovers atomic services in environments of medium size and adapts the social structure according to network changes. The inclusion of environment context has the potential to enhance this type of solutions by adding knowledge that enable the management of larger scenarios because of the creation of more relevant search spaces, and more informed decisions with regard to request management.

Solutions that support service composition [Chen et al., 2016, Urbieta et al., 2017] work in small size environments where organisation of services and request management are not key requirements. These approaches respond to changes in services or the network but do not consider changes in the IoT environment. The service search either requires high input from consumers [Urbieta et al., 2017] or expensive planning algorithms [Chen et al., 2016]. Wu et al. [Wu et al., 2015] organise services in an adaptive structure according to services' properties. It works in a centralised infrastructure that does not need to propagate requests but represents a unique point of failure. This approach discovers composed services using semantic methods, which might have latency issues in larger environments because of the number of services.

In summary, open gaps within current service discovery research are:

1. Approaches do not consider contextual information to organise services information or propagate requests. This context has the potential to create more informed service oriented architectures to drive efficient service discovery.

2. The service search either requires high human intervention or might present performance issues with regard to latency and accuracy in large environments. The exploration of progressive search mechanisms can mitigate these limitations.

3. Adaptation mechanisms respond to devices or network changes, or human behaviour which is difficult to capture and manage. They do not consider changes in the IoT environments where they work, causing outdated architectures.

Chapter 3

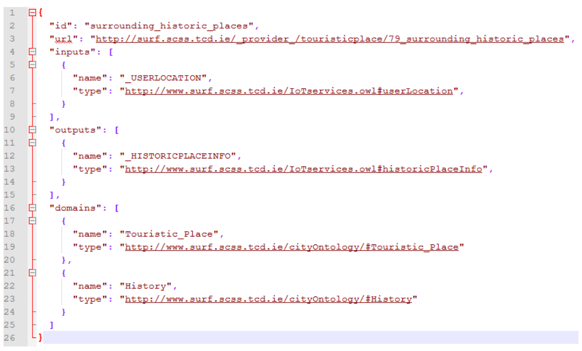
Design

The literature review in Chapter 2 identified a number of limitations in current service discovery for large and dynamic IoT environments (e.g, smart cities). Research gaps with current service discovery approaches are that i) existing research organises services information in structures that do not provide enough information to support efficient discovery in large environments; ii) such structures do not respond to changes in the environment where they work, which negatively impacts discovery performance over time; and iii) current planning methods either need rich input from consumers, which is not feasible in large environments, or work with reduced knowledge which impacts search accuracy and latency.

This chapter introduces *uDiscovery*, a service discovery model for smart cities based on urban context. First, this chapter presents the design objectives of *uDiscovery*, from which a list of requirements is presented. This chapter then introduces the system model and main concepts of *uDiscovery*. It continues with a description of the adaptive discovery and planning models that support *uDiscovery*. Finally, this chapter discusses $uDiscovery's$ contributions.

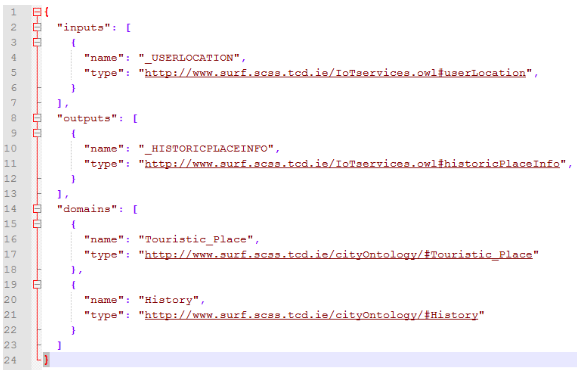
3.1 Design Objectives and Required Features

Chapter 1 introduces the research questions that this thesis addresses. Chapter 2 analyses current service discovery approaches and identifies these limitations. Next design objectives are defined according to these research questions and the state of the art analysis. This thesis aims: to design and build an efficient and adaptive service discovery model for smart cities, and to design and build an efficient search mechanism that respond to complex requests (i.e., service composition support), with minimal input from consumers. This work aims to build a service model that meets these objectives by supporting the following features:



```
{
  "id": "surrounding_historic_places",
  "url": "http://surf.scss.tcd.ie/_provider_/touristicplace/79_surrounding_historic_places",
  "inputs": [
    {
      "name": "_USERLOCATION",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#userLocation",
    }
  ],
  "outputs": [
    {
      "name": "_HISTORICPLACEINFO",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#historicPlaceInfo",
    }
  ],
  "domains": [
    {
      "name": "Touristic_Place",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#Touristic_Place"
    },
    {
      "name": "History",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#History"
    }
  ]
}
```

(a) Service Description



```
{
  "inputs": [
    {
      "name": "_USERLOCATION",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#userLocation",
    }
  ],
  "outputs": [
    {
      "name": "_HISTORICPLACEINFO",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#historicPlaceInfo",
    }
  ],
  "domains": [
    {
      "name": "Touristic_Place",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#Touristic_Place"
    },
    {
      "name": "History",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#History"
    }
  ]
}
```

(b) Request Description

Figure 3.1: *uDiscovery* - Data Formats.

***Feature 1*: Context-based service management**

Smart cities are IoT environments with large number of IoT services (**Challenge 1**). An efficient service discovery model for smart cities must organise services in reduced search spaces and forward requests where they are most likely to be solved, based on cities' information (i.e., urban context) (**RQ.1**).

***Feature 2*: Self-adaptive service management**

Cities are dynamic environments that evolve all the time because of the interactions between different entities (**Challenge 2**). A service discovery model for smart cities must adapt to the organisation of services according to city changes to maintain an efficient discovery over time(**RQ.2**).

***Feature 3*: Composition Support**

Citizens might request for composed services (**Challenge 3**). A service discovery model must support service composition in smart cities by searching for services that constitute such compositions with high accuracy and low latency (**RQ.3**)..

***Feature 4*: Minimal Consumer Input**

High level of consumers' input is not feasible in smart cities because citizens can not know about all available services in the environment (**Challenge 4**). A service discovery model for smart cities must minimise input from consumers and still offer accurate and fast search outputs (**RQ.3**).

3.2 System Model

This work considers cities as entities that offer services to citizens (e.g., transport services). These services are offered in different places and can be supported by software services that deliver useful information (e.g., bus station schedule). Citizens are mobile and can request software services to support their activities. For example, a citizen may want to know which is the best transport mode to use to get home from the city centre. This thesis focuses on the discovery of software services that citizens request and relies on a network of IoT gateways that covers the city. Service providers register their service descriptions in the network of gateways. A service description is defined as $s_{desc} = \langle id, url, I, O, D \rangle$, consisting of a service identifier, url endpoint, inputs, outputs, and domains. For example, Figure 3.1a shows the description of a service that retrieves information about surrounding historical places according to the user's location. This service has an input which corresponds to the user location and is semantically annotated by the concept *userLocation* (i.e., type field). The service output is the information about a historical place which is semantically annotated by the concept $historicPlaceInfo$. This service belongs to two domains named $Touristic_Place$ and $History$, which are also semantically annotated by the concepts in the type fields. All semantic annotations that describe inputs, outputs, and domains are defined in a set of ontologies that the Section 4.1.1 introduces (Chapter 4). Service consumers (e.g., service-based applications, service composition engines, etc.) can send requests that trigger the service discovery process. These requests are defined as $r = \langle I, O, D \rangle$, consisting of request inputs' types, outputs' types and domains. For example, a request for a service that provides information about surrounding historical places according to the user location (Figure 3.1b). This request specifies inputs, outputs, and domains which are semantically annotated as in the service description. These consumers can also send feedback about the discovery process as a boolean mark, which reflects that a discovered service was successful or not for them.

This thesis also considers cities as dynamic environments where different entities (e.g., citizen, city places, city services, city events, etc.) interact. These interactions affect the IT systems that support the city, such as the service discovery model. For example, a flash flood in a city area might cause consumers to ask for services that are not registered in the surrounding gateways. Three types of city events are identified, as follows:

- Unforeseen Events: These are unexpected events that happen in a city (e.g., a natural disaster, or an impromptu protest).

- Scheduled Events: These are events that are known by city authorities (e.g., a concert in a stadium). Authorities know when and where the event happens.
- Periodic Events: These are events that follow a pattern in the city because of the behaviour of its entities (e.g., citizens going to work every day).

3.3 Design Decisions

uDiscovery encompasses a set of decisions that enable the identified features to address the service discovery issues in smart cities with regard to the large number of services, dynamic environments, complex requests, and limited consumers' inputs. This section introduces these decisions from the perspectives of services organisation, requests management, and service planning. Services are organised in relevant search spaces, where requests are forwarded to improve the global discovery efficiency. And, local search efficiency is improved by applying a more informed and progressive planning.

3.3.1 Services Organisation

Previous approaches organise software services by their attributes in structures which might not be efficient in smart cities. They might fail to reduce search spaces and do not provide enough information to drive efficient discovery, as discussed in section 2.1. The next decisions are made to more appropriately organise services in smart cities.

Design Decision 1: City places as urban context

Places are spaces in a city that offer city services from a given domain. For instance, a hospital in a city provides services from the health care domain, or a college provides services from the education domain. Citizens interact in these places and use the offered city services. They also might request software services to get useful information about city services (e.g., a hospital opening hours). *uDiscovery* uses city places information as urban context because this information can be used to infer consumers' needs. For example, a consumer in a hospital is more likely to require services from the health domain. This decision supports Feature 1.

Design Decision 2: Gateways store services information according to surrounding places

uDiscovery organises software services according to city places to put the right service in the right place. IoT gateways are deployed in city places and receive requests from consumers in these places. Gateways store services information according to their surrounding places, as such places influence consumers' requests. For instance, gateways deployed close to a hospital should store services from the health care domain because consumers in the hospital are more likely to require health care related services. This decision supports Feature 1.

Design Decision 3: City events trigger services organisation adaptation

The initial services organisation based on urban context is likely to become outdated because of city events. For example, a flash flood in a city area might cause consumers to ask for services that are not registered in the surrounding gateways. This would affect the discovery performance, and cause smart cities to fail in their attempt to satisfy citizens' needs. *uDiscovery* enables each IoT gateway to recognise city events by measuring the performance of the discovery process over time. If the performance decays, it means that the gateway does not have the required information. *uDiscovery* creates an IoT network that moves services between gateways to respond to city events. This decision supports Feature 2.

3.3.2 Requests Management

Section 2.2 discussed previous approaches to forwarding requests to where they are most likely to be solved. Multicast, flooding, and gossiping-based approaches from WSNs and MANETs do not cover wide geographic areas such as a city, and cause high overhead, network bandwidth and devices battery depletion [Zikria et al., 2018]. Social-based approaches rely on devices proximity and uses interest, social metrics, or flooding-based strategies to disseminate information. These strategies reflect temporal and spatial dimensions from human behaviour, but fail to capture more complex social relationships that are still difficult to identify, manage, and measure. It impacts the dissemination efficiency because data is moved and replicated in environments where content providers and consumers might never have a link between them [Girolami et al., 2015a]. Bio-inspired approaches are a good alternative for information dissemination in large networks. These approaches are based on natural systems that work in complex environments and can propagate information in environments where nodes have a limited knowledge about other participants[Hamidouche et al., 2018].

Design Decision 4: Gateways forward requests according to other gateways surrounding places

Gateways forward consumer's requests according to other gateways surrounding places.

For example, it is more likely that gateways deployed in hospitals have services to solve requests related to the health domain, then requests related to the health domain should be forwarded to these gateway. *uDiscovery* enables each IoT gateway to forward requests to other gateways, based on their surrounding places. For instance, if a gateway receives a request related to the health care domain, and the gateway does not have information to solve such request. The gateway forwards the request to other gateways with a hospital close by. This decision supports Feature 1.

Design Decision 5: Bio-inspired urban based request forwarding

uDiscovery uses urban context to organise and replicate services. It also uses this knowledge to forward requests to gateways where they are most likely to be solved. Each gateway knows about places that surround other gateways in the network, but this knowledge is partial (i.e., a gateway does not know about all gateways in the network). A bio-inspired method supports the optimal propagation of information in environments with partial knowledge. *uDiscovery* uses an *ant colony* forwarding mechanism that sets pheromones information when the network of gateways is configured, and services are organised to define which are the most promising gateways for a given domain. This mechanism updates pheromones information at discovery time depending on the discovery success of a selected gateway. This decision supports Feature 1.

3.3.3 Service Planning

Previous approaches that support service composition either use service conversations or interface-based search to match requests and services. Conversations-based approaches require a high level of input and interface-based approaches have performance issues because of the lack of information and large number of services (Section 2.3). *uDiscovery* is based on the following decisions to address these issues.

Design Decision 6: Goal-driven service planning

Smart city environments expose a large number of services that need to be combined to address complex consumers' requests. These combinations exploit the actual potential of IoT environments by creating on-demand applications from existing services. High input from consumers is not feasible in such large environments because consumers might have a limited knowledge about available IoT services. *uDiscovery* uses a goal-driven approach to support service composition with minimal consumer's input. Consumers express their needs as desired outputs and available inputs which are compared against

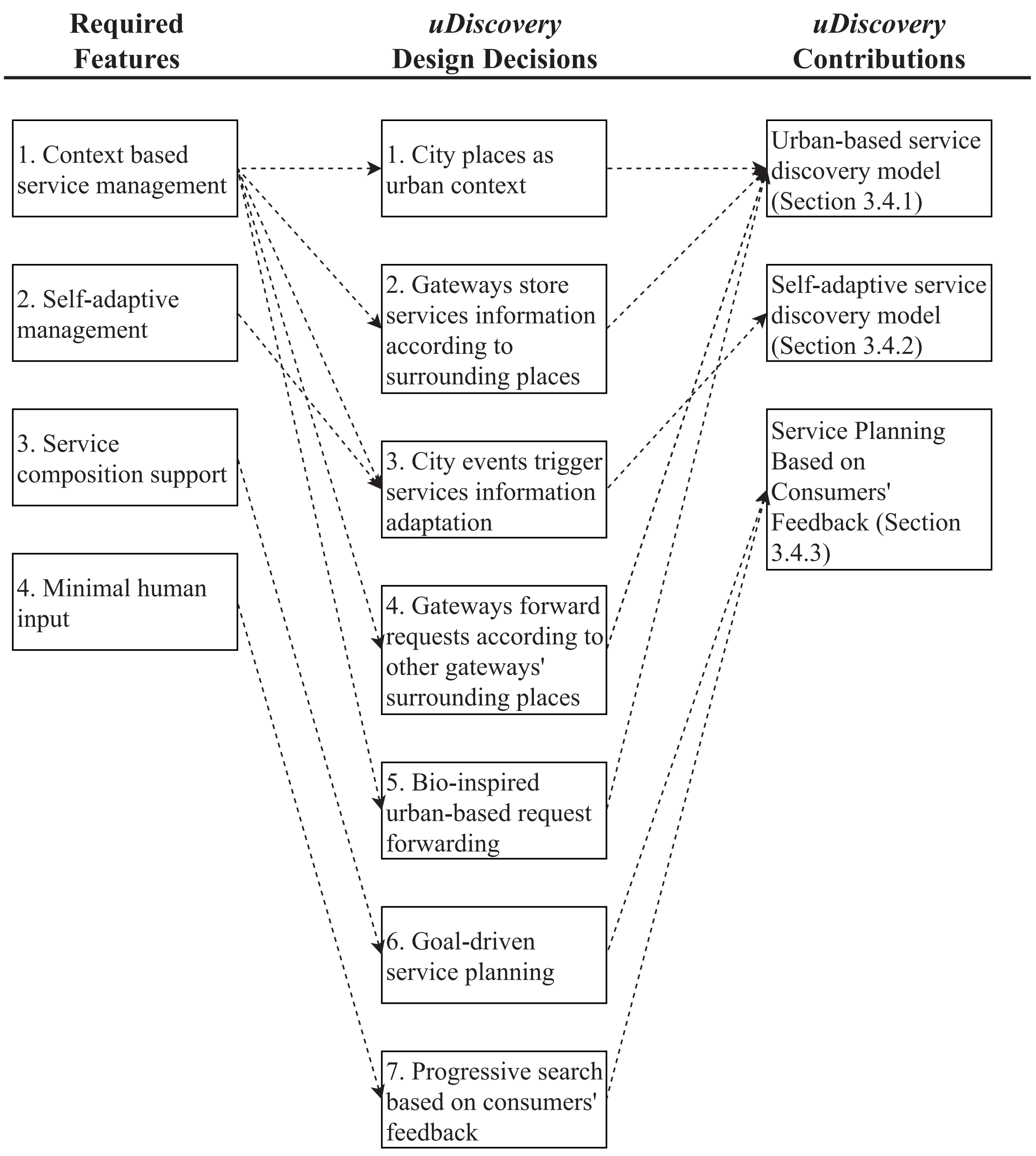

Required Features
uDiscovery Design Decisions
uDiscovery Contributions
1. Context based service management
2. Self-adaptive management
3. Service composition support
4. Minimal human input
1. City places as urban context
2. Gateways store services information according to surrounding places
3. City events trigger services information adaptation
4. Gateways forward requests according to other gateways' surrounding places
5. Bio-inspired urban-based request forwarding
6. Goal-driven service planning
7. Progressive search based on consumers' feedback
Urban-based service discovery model (Section 3.4.1)
Self-adaptive service discovery model (Section 3.4.2)
Service Planning Based on Consumers' Feedback (Section 3.4.3)

Figure 3.2: *uDiscovery* - Design Decisions Map.

the available services to create a service plan that satisfy such needs. This decision supports Feature 3 and Feature 4.

Design Decision 7: Progressive search based on consumers' feedback

Goal-driven planning is an interface-based approach that has accuracy issues because of limited knowledge about consumers' needs (i.e., only inputs and outputs information), and latency problems because it explores all the possible combinations of services that match. *uDiscovery* expands the goal-driven model to use consumers' feedback. Such feedback (i.e., historical data from previous searches) determines if a discovered plan

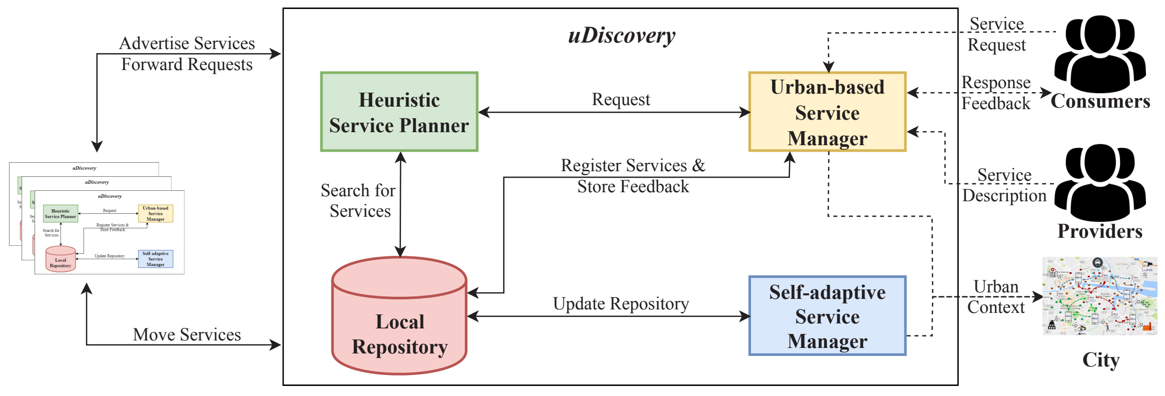


Figure 3.3: *uDiscovery* - High Level Architecture.

was correct or incorrect and is used to improve search accuracy. *uDiscovery* addresses latency issues through two strategies: first, the use of consumers' feedback avoids the exploration of incorrect combinations of services. Second, *uDiscovery* explores the plans according to how well each plan meets the request's requirements. *uDiscovery* explores the most promising plans (i.e., search space reduction), and avoids wasting time on the less promising plans (i.e., progressive search). This decision supports Feature 3.

Figure 3.2 shows how *uDiscovery* design decisions map to the required features. *uDiscovery* makes three contributions based on these decisions. Decision 1, 2, 3, 4, and 5 support an urban-based service discovery model for smart cities that organises services information and solve consumers' requests based on city places. Decision 4 supports a self-adaptive management model that updates the organisation of services according to the city events. Decision 6 and 7 support a service planning model based on consumers' feedback that supports service composition by applying a progressive search. The remainder of this chapter details *uDiscovery* contributions and how they work.

3.4 uDiscovery

uDiscovery is an urban-centric model for service discovery in smart cities (Figure 3.3). *uDiscovery* is deployed on each gateway and is composed of four components. The *urban-based service manager* receives service descriptions from providers and decides to store them in the *local repository* or to advertise them to other gateways based on urban context. This component receives requests and tries to solve them using a *heuristic service planner* that searches for services in the *local repository* based on consumers' feedback. If requests are solved, *uDiscovery* returns the response to consumers and waits for their feedback, which is in turn stored in the *local repository*. Otherwise, *uDiscovery* forwards requests to other

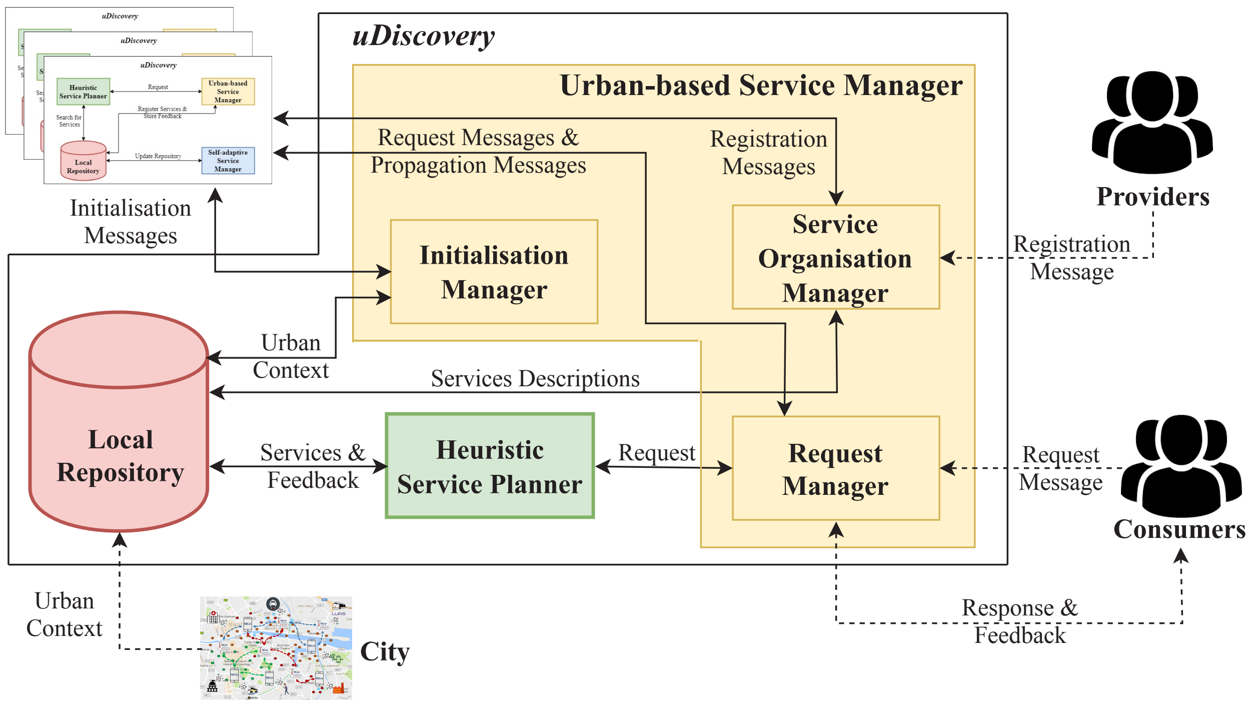


Figure 3.4: *uDiscovery* - Urban-based Service Manager Architecture.

gateways in the network. The *self-adaptive service manager* updates the *local repository* by moving services between gateways according to city events. The following sections explain how these components interact to achieve an adaptive and efficient discovery.

3.4.1 Urban-based Service Discovery Model

uDiscovery organises services using a *urban-based service manager* (Figure 3.4). This component has three sub-components. The initialisation manager sends and receives initialisation messages with gateways information to create links between them based on urban context (Section 3.4.1.2). The service organisation manager receives registration messages from providers and gateways, stores services in the repository and advertises services to relevant gateways to put *the right service in the right place in preparation for discovery* (Section 3.4.1.3). The request manager receives discovery messages from consumers, tries to solve them using the *heuristic planner* (Section 3.4.3), or forwards requests to relevant gateways based on the *ant colony* forwarding mechanism (Section 3.4.1.4).

Parameter	**Description**
x	An integer that defines the distance in metres to recognise surrounding places.
Hops limit	An integer that defines the limit of hops for messages in the network of gateways

Table 3.1: Urban-based Service Discovery Model Parameters.

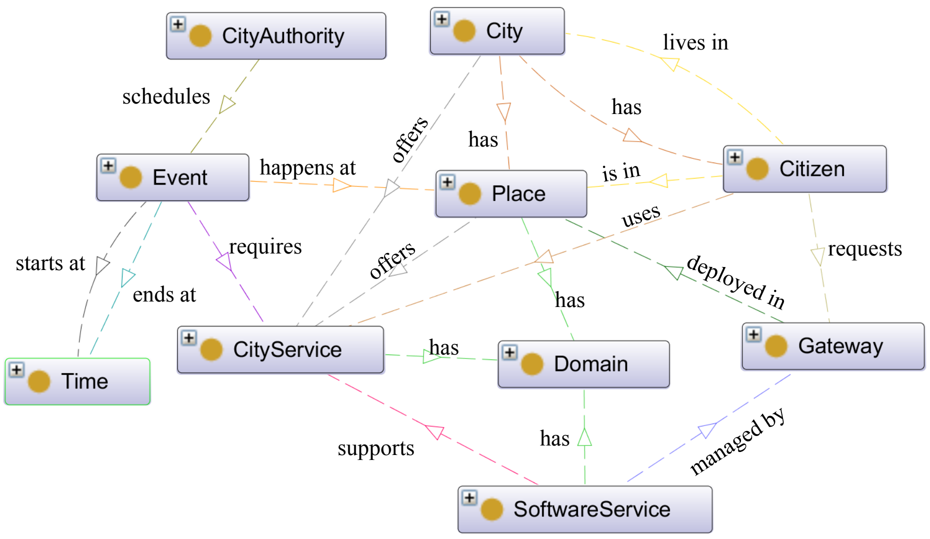


Figure 3.5: *uDiscovery* - Service Discovery Knowledge Model.

Table 3.1 shows the parameters of the urban-based service discovery model. This model uses an integer x as the distance between the gateway and the places which are considered surrounding places (e.g., a place surrounds a gateway if it is in a radius of 100m from the gateway). The hops limit defines the maximum number of hops allowed for each message in the network of gateways to avoid overhead. Chapter 5 experiments with different values for these parameters and reports *uDiscovery's* performance results.

3.4.1.1 Urban Context Management

The *urban-based service manager* handles services information in the network gateways based on the urban context. Each gateway captures the information of its surrounding places and decides how to store, and replicate services information, and forward consumers' requests. *uDiscovery* needs to formalise the urban context in a model that can be usable for each gateway in the network, and represents the different city concepts that interact in the service discovery process. *uDiscovery* creates a knowledge model that defines city concepts and their relations in the service discovery domain (Figure 3.5).

Definition 1. The **knowledge model** is an ontology $O(V, A)$, where V is the vocabulary and A the set of axioms. The vocabulary $V(T, R)$ defines the following concepts:

- **City:** place where the different concepts of *uDiscovery* interact (e.g., Dublin).
- **Citizen:** person who is in a city and can use its services.
- **Place:** places in the city, where citizens perform their daily activities (e.g., bus station, museum, stadium).

- **City Service:** an urban service offered to the citizens by the city or a place (e.g., transport, health service, etc.).
- **Software Service:** piece of software that encapsulates IoT devices capabilities and provides useful information to citizens (e.g., number of available bikes in a station, location of nearby hospitals).
- **Gateway:** digital entity that manages software services.
- **Domain:** different smart city domains such as health, education, mobility.
- **Event:** a thing that happens in the city.
- **City Authority:** different entities or institutions that govern the city.
- **Time:** the temporal dimension of a city.

The vocabulary V also defines a number of relations R between concepts (Table 3.2), and the ontology O defines a set of axioms A to infer city concepts relations (Table 3.3).

3.4.1.2 Initialisation Manager

The *urban based service manager* handles services using the previous knowledge model in a network of IoT gateways. This network needs to be initialised and configured based on the knowledge model to create links between gateways, which drive the exchange of messages in the following stages of service organisation and discovery. The *initialisation manager* configures the network based on the formalised urban context to organise and discover services in a distributed fashion. It initialises the network by exchanging gateways information. This section introduces the main elements of this process, with the algorithms to manage these elements.

Definition 2. A **gateway** is defined as $gw = \langle gw_{id}, gw_{loc}, SP, D, GR, GWS, P, C\rangle$, where gw_{id} is a unique identifier for a gateway, and gw_{loc} is the gateway location. The set SP represents the city places that surround the gateway gw. The set D represents the gateway's domains, which are defined based on the surrounding places SP using the axiom $A3$. The set GR represents the relevance of the gateway for each domain in D. The set GWS represents the list of other gateways in the network that the gateway gw knows. P represents the pheromones values for the links between gw and the gateways in GWS. *uDiscovery* uses these pheromones to find the best route between gateways based on the urban context. Each gateway initialises, increases, or decreases pheromones values according to its interactions with other gateways in the gateways initialisation, service organisation, and service discovery

Table 3.2: Ontology Relations R.

$has\,(City, Citizen)$ $has\,(City, Place)$	The relation between a city and its citizens and places.
$lives\,(Citizen, City)$	The relation between a citizen who lives in a city.
$isIn\,(Citizen, Place)$	The relation between a citizen who is in a place.
$offers\,(City, CityService)$ $offers\,(Place, CityService)$	The relation between a city or a place and the city services that they offer.
$uses\,(Citizen, CityService)$	The relation between a citizen who uses a city service.
$supports\,(SoftwareService, CityService)$	The relation between a software service that supports a city service (e.g., the software service that provides information about nearby hospitals supports the health city service).
$hasDomain\,(Place, Domain)$ $hasDomain\,(CityService, Domain)$ $hasDomain\,(SoftwareService, Domain)$	The relation between places, city services, software services and their domains.
$isDeployed\,(Gateway, Place)$	The relation between a gateway and the place where it is deployed.
$isManagedBy\,(SoftwareService, Gateway)$	The relation between a software service and the gateway that manages it.
$requestsSoftwareService\,(Citizen, Gateway)$	The relation between a citizen who requests a software service to a gateway.
$happensAt\,(Event, Place)$	The relation between an event that happens at a place.
$requires\,(Event, CityService)$	The relation between an event that requires a city service.
$startsAt\,(Event, Time)$	The relation between an event that starts at a given time.
$endsAt\,(Event, Time)$	The relation between an event that ends at a given time.
$schedules\,(CityAuthority, Event)$	The relation between a city authority that schedules a known event.

Table 3.3: Ontology Axioms A.

$A1 : offers\,(x, y) \leftarrow$ $hasDomain(x, z), hasDomain(y, z)$	States that if a *Place* x has the same domains of a *CityService* y, then x $offers$ y.
$A2 : supports\,(x, y) \leftarrow$ $hasDomain(x, z), hasDomain(y, z)$	States that if a *SoftwareService* x has the same domains of a *CityService* y, then x *supports* y.
$A3 : hasDomain\,(x, z) \leftarrow$ $isDeployed(x, y), hasDomain(y, z)$	States that if a *Gateway* x is deployed in the *Place* y and y has the domain z, then x *has* the domain z.

Algorithm 1 Initialisation Manager - Gateway Initialisation.

```
1: function INITIALISATION($x$)                ▷ where $x$ - distance to recognise places
2:     $cityPlaces \leftarrow osmAPI()$
3:     for each $place \in cityPlaces$ do
4:         if $distanceBetween(gw_{loc}, place) <= x$ then
5:             $SP.add(place)$
6:     $cityServices \leftarrow getCityServices(SP)$
7:     $D \leftarrow getDomains(cityServices)$
8:     $GR \leftarrow calculateRelevance(D)$
9:     $sendMessage(GwAdv_{msg}(*, gw), 1)$
```

processes. The set C represents the cost for sending a message from gw to each gateway in the set GWS, and reflects the distance between gateways. P and C are initialised with each gateway and rule the service discovery in next stages. The request manager uses both pheromones and costs values as inputs for the *ant colony* forwarding mechanism to select the gateways where to forward requests.

Definition 3. A **city place** is defined by $cp = \langle cp_{id}, cp_{loc}, CS, D \rangle$, where cp_{id} is the place unique identifier, and cp_{loc} its location. CS represents the city services offered by the place following the axiom $A1$ (See Table 3.3). The set D represents the place's domains.

Definition 4. **Pheromones** (P) represent how relevant is a gateway in the set GWS to solve a request for a given domain from the perspective of gw. Each gateway stores pheromones for each domain as $\tau(d)_{gw_i, gw_j}$ which represents the pheromone for a link between gw_i and gw_j for the domain d.

Definition 5. **Costs** (C) represent how far is each gateway in the set GWS from gw. This **cost** is defined by L_{gw_i, gw_j} as the distance between gw_i and gw_j.

Definition 6. A **gateway advertisement** is defined by $GwAdv_{msg} = \langle id_{rec}, gw, h \rangle$, where id_{rec} is the identifier of the receiver gateway, gw is the information of the sender gateway, and h is the number of hops of the message to avoid network overhead.

uDiscovery uses previous definitions to initialise the network of gateways according to the following steps:

Step 1: Algorithm 1 shows the process that managers perform to initialise the network. Each gateway identifies its surrounding places in a given distance x, using its location and city places data from Open Street Map (OSM)[1] APIs (Line 2 to 4). Then, it defines the city services that surrounding places offer using axiom $A1$, and their domains using axiom

[1]OSM - `https://www.openstreetmap.org/`

$A3$ (Line 5 and 6). Each gateway calculates its relevance for D (See Definition 2.) following equation 3.1 (Line 7).

$$rel(gw, d) = \frac{ncs}{ds} \tag{3.1}$$

where ncs is the number of city services of domain d that are offered by the surrounding places of gw, and ds is the total number of domains of all city services offered by theses places. Finally, each gateway broadcasts its information to other gateways in the network by sending gateway advertisement messages (Line 8).

Step 2: When an *initialisation manager* in gateway gw_i receives a gateway advertisement message from another gateway gw_j (Algorithm 2), the manager stores the data of gw_j in GWS (See Definition 2.) (Lines 5 and 6). It updates the pheromones information P by creating an entry for each domain of gw_j with its respective relevance score (Lines 7 and 8). The manager initialises pheromones values on each gateway based on the urban context represented by the relevance score. These pheromones values enable *uDiscovery* to integrate the urban context of each gateway and the *ant colony* forwarding mechanism that the *request manager* uses in the discovery process. The manager also updates the communication cost C by creating an entry for gw_j with the respective distance (Line 9). Finally, the manager advertises the information about the gateway gw_j to other gateways in GWS, if the message hops is less than the limit of hops. These destinations are selected according to how interesting is gw_j to each gateway in GWS (Lines 10 to 14).

Definition 7. The **interest** of a gateway gw_i in the information of gw_j follows eq 3.2.

$$interest(gw_i, D) = \sum_{x=1}^{n} rel(gw_i, d_x) \tag{3.2}$$

where d_x belongs to D, which is the list of domains of the surrounding places of the gateway gw_j. This equation reflects that a gateway gw_i is interested in gw_j, if gw_i is relevant for one or more domains from D. Destinations are ranked according to this interest, to spread gateways' information to the most similar gateways in the city according to the domains of their surrounding places. Finally, the manager responds to gateway gw_j by sending an initialisation message with the information of gw_i (Line 15).

Each manager maintains the network according to the gateway type (i.e., static or mobile). Static gateways perform a periodic operation to confirm which gateways in the list GWS are

Algorithm 2 Initialisation Manager - Network Initialisation.

```
1: function MESSAGE ARRIVES($msg$)
2:     if $msg\ is\ GwAdv_{msg}$ then
3:         $gw_j \leftarrow msg.gw$
4:         $hops \leftarrow msg.h$
5:         if $gw_j\ not\ in\ GW$ then
6:             $GW.add(gw_j)$
7:             for each $d \in gw_j.D$ do
8:                 $\tau(d)_{gw_i,gw_j} \leftarrow rel(gw_j, d)$
9:             $L_{gw_i,gw_j} \leftarrow distance(gw_i, gw_j)$
10:            $hops \leftarrow hops + 1$
11:            if $hops <= hopsLimit$ then
12:                $destinations \leftarrow getDestinationsByInterest(gw_j.D)$
13:                for each $gw \in destinations$ do
14:                    $sendMessage(GwAdv_{msg}(gw_{id}, gw_j, hops))$
15:            $sendMessage(GwAdv_{msg}(msg.sender, gw_i, hops))$
```

alive. A static gateway gw_i sends a confirmation message to gw_j with a time-out t. If gw_j does not reply within the time-out period, gw_i removes gw_j from GWS. If, after this process, the size of GWS is zero, gw_i repeats the Algorithm 1. For mobile gateways, the maintenance depends on their movement. When a mobile gateway gw_i moves a distance of x meters, it removes gateways in GWS which are out of range by calculating their new distance. If, after this process, the GWS is empty, gw_i repeats the Algorithm 1 with a new list of surrounding places according to its new position. Each manager updates its pheromones' information in a periodic way following the equation 3.3.

$$\tau(d)_{gw_i,gw_j} = (1-\rho)\tau(d)_{gw_i,gw_j} \tag{3.3}$$

where ρ is the pheromones evaporation factor. This evaporation enables the inclusion of gateways that might suddenly appear in the network. $\rho = 0.1$ in this work based on a previous study on the ant colony optimisation (ACO) algorithm for solving the travelling salesman problem [Cheong et al., 2017]. The travelling salesman problem is similar to the discovery problem in this thesis. Each gateway wants to find the shortest path to solve requests through a set of gateways, as the salesman wants to find the shortest path to travel through a set of cities.

3.4.1.3 Service Organisation Manager

The service manager organises services descriptions when it receives registration messages from providers or other gateways in the network. Each gateway stores the service description

Algorithm 3 Service Organisation Manager.

```
function MESSAGE ARRIVES($msg$)
    if $msg\ is\ Reg_{msg}$ then
        $s_{desc} \leftarrow msg.s_{desc}$
        $hops \leftarrow msg.h$
        if $s_{desc}\ not\ in\ localRepository$ then
            $SD \leftarrow s_{desc}.D$
            if $shareDomains(D, SD)$ then
                $localRepository.insert(s_{desc})$
                $sendMessage(RegRes_{msg}(msg.sender, gw_i, s_{desc}))$
        $hops \leftarrow hops + 1$
        if $hops <= hopsLimit$ then
            $destinations \leftarrow getDestinationsByInterest(s_{desc}.D)$
            for each $gw \in destinations$ do
                $sendMessage(Reg_{msg}(gw_{id}, s_{desc}, hops))$
    if $msg\ is\ RegRes_{msg}$ then
        $gw_j \leftarrow msg.sender$
        $s_{desc} \leftarrow msg.s_{desc}$
        for each $d \in s_{desc}.D$ do
            $\tau(d)_{gw_i,gw_j} \leftarrow \tau(d)_{gw_i,gw_j} + \rho$
```

in its *local repository*, if the gateway is relevant following equation 3.1. The manager responds the registration message, and advertises the description to other relevant gateways in the network using the same equation. This equation reflects how relevant is a gateway to store a description according to the service domains and the gateway surrounding places. This process enables *uDiscovery* to organise services in the network of gateways based on urban context. The *manager* uses the following definitions and processes to organise services:

Definition 7. A **registration message** is defined as $Reg_{msg} = \langle id_{rec}, s_{desc}, h\rangle$, where id_{rec} is the identifier of the receiver gateway, s_{desc} is the service description to be registered, and h is the message limit of hops.

Definition 8. Once a gateway gw_i receives a registration message from a provider p, or a service advertisement message from a gateway gw_j, to register a service description s_{desc}, gw_i notifies p or gw_j about the registration success, if gw_i registers s_{desc}. Gateways use this response to reinforce their links to other gateways for a given domain. For example, gw_j increases the pheromones values of gw_i for the domains of s_{desc}, if gw_i registered the s_{desc} that gw_j advertised. This response message message is defined as $RegRes_{msg} = \langle id_{rec}, gw, s_{desc}\rangle$, where id_{rec} is the receiver identifier, gw is the gateway that registered the service, and s_{desc} is the service description that was registered.

Algorithm 3 shows the process that the service managers perform when they receive registration (Lines 2 to 14) and registration response messages (Lines 15 to 19). When a gateway gw_i receives a registration message, the service manager first validates if the received description s_{desc} already exists in its repository. The manager stores s_{desc} and sends a registration response to the sender message, if the service is not in the *local repository* and shares domains with gw_i (Lines 5 to 9). The service manager advertises s_{desc} by sending the registration message to other gateways in the network. Destination gateways are selected according to their potential interest in the service following equation 3.2, where D is the list of domains of the service. These potential interests enable each gateway to advertise services descriptions to relevant gateways based on their surrounding places (i.e., urban context) to put the *right service in the right place*. gw_i advertises services if the number of hops of the registration message is less than the hops limit to avoid network overhead (Lines 10 to 14).

The manager updates the pheromones' values in gw_i when a registration response message arrives. It increases the pheromones value by adding ρ (i.e., 0.1) for each domain in the s_{desc} that was previously registered by gw_j. This increment reinforces the links between gw_i and gw_j for the service domains (Lines 15 to 19). These links are used later by the request manager to forward requests where they are most likely to be discovered.

3.4.1.4 Request Manager

The *request manager* drives a distributed discovery process based on the network of gateways and the services organisation that *uDiscovery* builds. This *manager* aims to improve the discovery efficiency by searching for services in, and forwarding requests to the most relevant search spaces that the service organisation creates. This *manager* is also responsible for sending responses back to consumers and receiving, and propagating their feedback. It propagates responses and feedbacks because a discovery process might involve more than one gateway, when the request is not solved by the gateway that receives the request from the consumer. Each included gateway is added to a list of participants that enable such propagation.

Figure 3.6 shows the different processes that the request manager follows when it receives different messages from consumers or other gateways in the network. The service discovery process starts when the *requestmanager* receives a request from consumers. The *manager* tries to solve the request using the *heuristic service planner*, which searches for services in the local repository based on consumers' feedback (Section 3.4.3). If the *planner* solves the request, the *request manager* sends the response to the consumer. Otherwise, it uses the pheromones and costs information in the *ant colony* mechanism to forward the request to

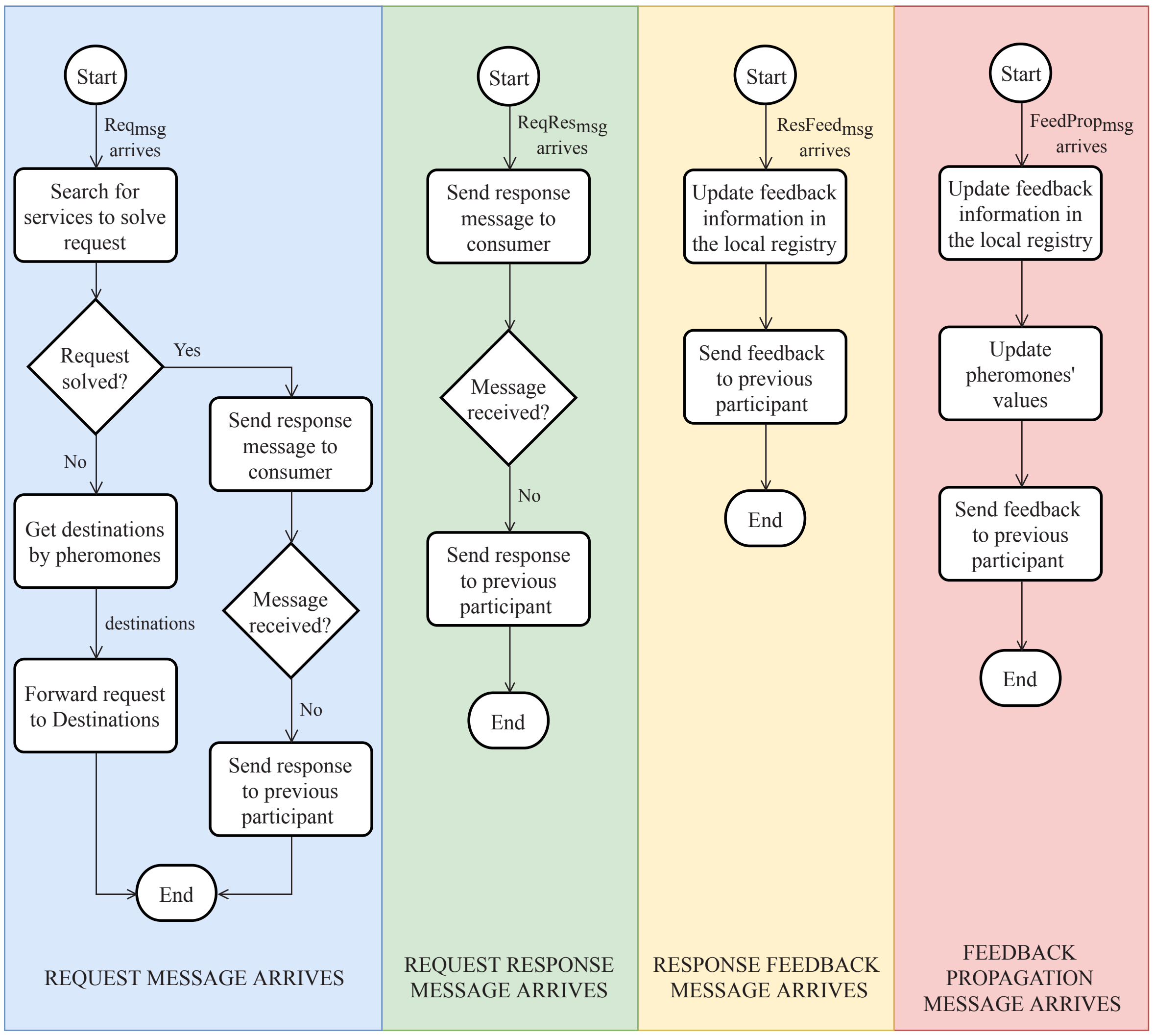

Start
Reqmsg arrives
Search for services to solve request
Request solved?
Yes
No
Send response message to consumer
Get destinations by pheromones
destinations
Message received?
Forward request to Destinations
No
Send response to previous participant
End
REQUEST MESSAGE ARRIVES
Start
ReqResmsg arrives
Send response message to consumer
Message received?
No
Send response to previous participant
End
REQUEST RESPONSE MESSAGE ARRIVES
Start
ResFeedmsg arrives
Update feedback information in the local registry
Send feedback to previous participant
End
RESPONSE FEEDBACK MESSAGE ARRIVES
Start
FeedPropmsg arrives
Update feedback information in the local registry
Update pheromones' values
Send feedback to previous participant
End
FEEDBACK PROPAGATION MESSAGE ARRIVES

Figure 3.6: *uDiscovery* - Request Manager Processes.

the most promising set of gateways according to its partial knowledge of the network, which is built when the network is configured and the services organised. If the consumer does not receive the response, the *manager* sends the response message to the previous participant in the distributed discovery process. When a gateway receives a response message, the *manager* tries to send the response to the consumer. If the consumer does not receive the response, it sends the response to the previous participant. If a consumer gets a response, it might send its feedback about the retrieved services. The *manager* stores the feedback in the local registry using the *heuristic planner* (Section 3.4.3) and propagates this feedback to previous participants. When a gateway receives feedback information from other gateway, it updates the feedback information in the local registry, and the pheromones values to reinforce gateways links according to the success or failure of the discovery process. The *request manager* uses the following definitions and algorithms to realise previous processes.

Definition 9. A **request message** is defined as $Req_{msg} = \langle id_{rec}, c, r, PGWS, PSOL, h \rangle$, where id_{rec} is the identifier of the receiver gateway, r is the service request, and c the request consumer. The set $PGWS$ represents the previous gateways that the request has visited, the set $PSOL$ represents the partial solutions that previous gateways have discovered, and h is the hops limit for the message, to avoid network overhead.

Definition 10. A **request response message** is used to send the response to consumers and is defined as $ReqRes_{msg} = \langle id_{rec}, c, r, SOL, GWS \rangle$, where id_{rec} is the receiver identifier, c the consumer, r consumer' request (Section 3.2), SOL is the list of plans that solve the request r, and GWS are the gateways that participate in the discovery process.

Definition 11. A **response feedback message** is used by consumers to send feedback about responses. This message is defined as $ResFeed_{msg} = \langle id_{rec}, r, FSOL, GWS \rangle$, where id_{rec} is the receiver identifier, r is the consumer request (Section 3.2), $FSOL$ are the solutions with their respective feedback, and GWS are the gateways that participated to solve r. Each plan feedback is a Boolean mark that defines if the plan was correct or incorrect from the consumer's perspective. For example, a consumer might mark a plan with 1 if its execution was successful. The *heuristic service planner* uses this mark to perform a search that avoids the exploration of incorrect plans (Section 3.4.3).

Definition 12. Different gateways can participate in the discovery process. But only the one that sent the response to the consumer receives its feedback. This information is propagated to all participants. Gateways use a **feedback propagation message** to send consumers' feedback to other participants. This message is defined by $FeedProp_{msg} = \langle id_{rec}, r, FSOL, PGWS \rangle$, where id_{rec} is the receiver identifier, r is the solved request, $FSOL$ are the solutions for r with their feedback, and $PGWS$ are the participant gateways.

Algorithm 4 shows the detailed processes that the *request manager* performs when it receives request messages (Lines 7 to 22), request response messages (Lines 23 to 28), request feedback messages (Lines 29 to 35), and feedback propagation messages (Lines 36 to 44). The *request manager* in a gateway gw_i searches for services in its repository to solve a request by using the *planner* (Section 3.4.3) (Line 8). If the request is solved, the manager sends the response to the consumer. If the consumer does not receive the response, the manager sends the response to other participant in the discovery process (Lines 10 to 15). If the request is not solved, the manager forwards the request and the partial solutions to other gateways in the network. It selects the most promising destinations using the pheromones and cost information as inputs for the *ant colony* forwarding mechanism. Requests are forwarded if the message hops is less than the hops limit to avoid overhead (Lines 16 to 22).

Algorithm 4 Request Manager.

```
1: function MESSAGE ARRIVES($msg$)
2:     $r \leftarrow msg.r$
3:     $c \leftarrow msg.c$
4:     $PGWS \leftarrow msg.PGWS$
5:     $PSOL \leftarrow msg.PSOL$
6:     $hops \leftarrow msg.h$
7:     if $msg$ is $Req_{msg}$ then
8:         $SOL \leftarrow heuristicPlanning(r, PSOL)$
9:         $PGWS.add(gw_i)$
10:        if $solved$ then
11:            $sendMessage(DiscRes_{msg}(c_{id}, c, r, SOL, PGWS))$
12:            if $\neg received$ then
13:                $PGWS.remove(gw_i)$
14:                $gw \leftarrow getPreviousGateway(PGWS)$
15:                $sendMessage(DiscRes_{msg}(gw_{id}, c, r, SOL, PGWS))$
16:        else
17:            $PSOL.add(SOL)$
18:            $hops \leftarrow hops + 1$
19:            if $hops <= hopsLimit$ then
20:                $destinations \leftarrow getDestinationsByPheromones(r.D)$
21:                for each $gw \in destinations$ do
22:                    $sendMessage(Disc_{msg}(gw_{id}, c, r, PGWS, PSOL, hops))$
23:    if $msg$ is $ReqRes_{msg}$ then
24:        $sendMessage(DiscRes_{msg}(c_{id}, c, r, SOL, PGWS))$
25:        if $\neg received$ then
26:            $PGWS.remove(gw_i)$
27:            $gw \leftarrow getPreviousGateway(PGWS)$
28:            $sendMessage(DiscRes_{msg}(gw_{id}, c, r, SOL, PGWS))$
29:    if $msg$ is $ResFeed_{msg}$ then
30:        $r \leftarrow msg.r$
31:        $FSOL \leftarrow msg.SOL$
32:        $localRepository.updateFeedback(r, FSOL)$
33:        $PGWS.remove(gw_i)$
34:        $gw \leftarrow getPreviousGateway(PGWS)$
35:        $sendMessage(DiscProp_{msg}(gw_{id}, r, FSOL, PGWS))$
36:    if $msg$ is $FeedProp_{msg}$ then
37:        $gw_j \leftarrow msg.sender$
38:        $FSOL \leftarrow msg.SOL$
39:        $localRepository.updateFeedback(r, FSOL)$
40:        for each $d \in r.D$ do
41:            $\tau(d)_{gw_i,gw_j} \leftarrow \tau(d)_{gw_i,gw_j} + f$
42:        $PGWS.remove(gw_i)$
43:        $gw \leftarrow getPreviousGateway(PGWS)$
44:        $sendMessage(DiscProp_{msg}(gw_{id}, r, FSOL, PGWS))$
```

Definition 13. The **potential** of a gateway gw_j to solve a request r forwarded from a gateway gw_i is defined by equation 3.4.

$$potential(gw_j, D) = \sum_{x=1}^{n} \frac{\tau(d_x)_{gw_i,gw_j}{}^{\alpha} \eta_{gw_i,gw_j}{}^{\beta}}{\sum_{y=1}^{m} \tau(d_x)_{gw_i,gw_y}{}^{\alpha} \eta_{gw_i,gw_y}{}^{\beta}} \tag{3.4}$$

where $\tau(d_x)_{gw_i,gw_j}$ is the pheromone value for the link from gw_i to gw_j for the domain d_x that belongs to the list of domains of the request D with size n. α controls the influence of $\tau(d_x)_{gw_i,gw_j}$ and is greater or equal to 0. η_{gw_i,gw_j} is the cost of sending a message from gw_i to gw_j (i.e., the distance) and is defined as $\eta_{gw_i,gw_j} = \frac{1}{L_{gw_i,gw_j}}$. β controls the influence of η_{gw_i,gw_j} and is greater or equal to 1. $\alpha = 1$ and $\beta = 2$ in this work based on the previous study on the ACO algorithm for solving the travelling salesman problem [Cheong et al., 2017], and to prioritise closer gateways which might imply less hops and latency in the discovery process. m is the number of gateways that are relevant for the d_x according to the pheromones values in gw_i. The *ant colony* forwarding mechanism uses the potential of each gateway to rank destinations and selects the most promising ones based on the urban context, which is reflected by the pheromones values. Gateways forward requests where they are most likely to be solved using this mechanism in a distributed environment where each node has a limited knowledge about the rest of the network.

If the manager receives a response message, it sends the response to the consumer (Line 24). If the consumer does not receive the response because there is no link between gw_i and c, the manager sends the response to the previous gateway that participate in the discovery process (Lines 25 to 28). If gw_i receives a feedback message from a consumer, it updates the feedback for the discovered plans in the repository (Lines 29 to 32). The *request manager* propagates this feedback to the previous participant (Lines 33 to 35). Once gw_i receives a message $DiscProp_{msg}$ from another gateway gw_j, the manager updates the feedback information for the discovered plans in the *local repository* (Lines 36 to 39). The manager also updates the pheromone information for the link between gw_i and gw_j for each request domains by adding the feedback f (Line 41). Finally, the manager propagates the feedback information to other participants of the discovery process (Lines 42 to 44).

3.4.2 Self-adaptive Service Discovery Model

uDiscovery adapts the organisation of services according to the city events to maintain an efficient service discovery over time. *uDiscovery* creates an IoT network of gateways, which exchange services information based on the city changes to put *the right service in the right*

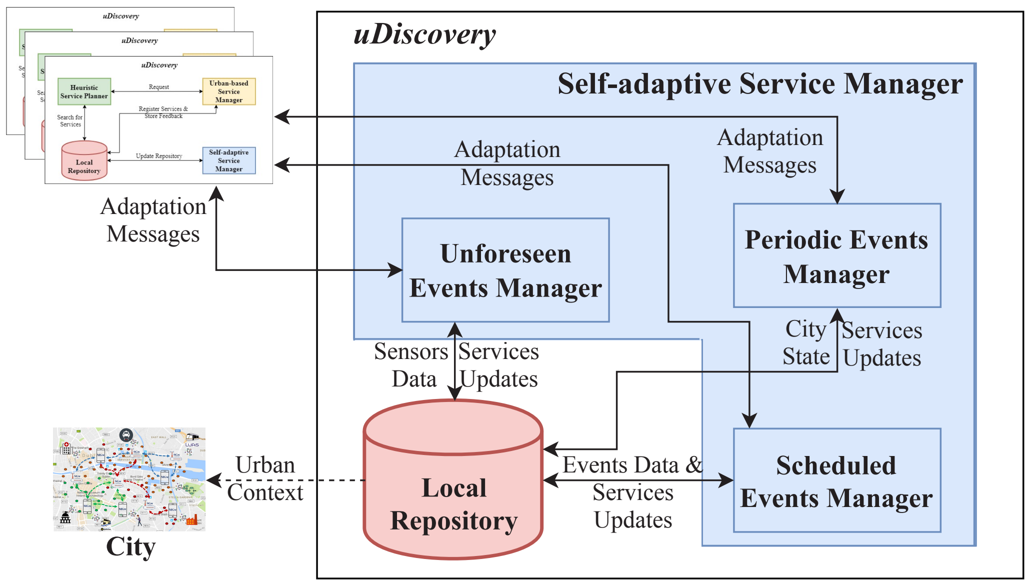


Figure 3.7: *uDiscovery* - Self-adaptive Service Manager Architecture.

place, at the right time. *uDiscovery* deploys a *self-adaptive service manager* on each IoT gateway, which is detailed in the Figure 3.7. There are three sub-components that adapt the service information in the *local repository* according to the city events. These components identify city events and move services information in the network by sending adaptation messages to other gateways. The *unforeseen events manager* identifies unforeseen events and moves services accordingly. The *scheduled events manager* queries pre-defined events in the repository and updates the services. Finally, the *periodic events manager* updates the services information based on city patterns.

Table 3.4 shows the parameters of the self-adaptive model. This model uses a threshold T to validate if the utility of the discovery process is acceptable, or adaptive actions are needed to improve its value. Hidden nodes defines the number of nodes in the neural network that *uDiscovery* uses to manage periodic events. The learning rate is used by the periodic events manager to learn from city behaviour. Chapter 5 experiments with different values for these parameters and reports $uDiscovery's$ performance results.

The *self-adaptive service manager* follows the principles defined by Lalanda et. al for autonomic systems [Lalanda et al., 2013].

Definition 14. The self-adaptive manager defines as a **policy** the improvement of the service discovery efficiency in smart cities over time. The discovery efficiency is defined as a set of **goals** that *uDiscovery* pursues. *uDiscovery* aims to increase the number of solved requests, provide the right results (i.e., search precision), reduce discovery latency, reduce network

Parameter	Description
T	A double that defines the threshold for the utility value in the adaptive process for unforeseen events
Hidden Nodes	An integer that defines the number of hidden nodes in the neural network used to manage periodic events
Learning Rate	A doulble that defines the learning rate of the algorithm that is used to manage periodic events

Table 3.4: Self-adaptive Service Discovery Model Parameters.

overhead, and reduce resource usage.

The manager has sensors that monitor how *uDiscovery* meets these goals. These sensors collect data about the number of requests that the system receives rr_t, and the number of requests that the system solves sr_t. They measure the search precision sp_{sr_i}, response time rt_{sr_i}, and numbers of hops nh_{sr_i} of each solved request, and the percentage of used storage in the gateway pus_t. as follows:

Definition 14. The manager defines a set of metrics to aggregate the sensed data. The rate of solved requests rsr_t is the relation between solved requests and received requests at a given time t (*eq* 3.5).

$$rsr_t = \frac{sr_t}{rr_t} \tag{3.5}$$

The average precision asp_t aggregates the precision of the solved requests up to a given time t. It follows equation 3.6.

$$asp_t = \frac{\sum_{i=1}^{sr_t} sp_{sr_i}}{sr_t} \tag{3.6}$$

The average response time art_t aggregates the latency of the solved requests until a given time t. It follows equation 3.7.

$$art_t = \frac{\sum_{i=1}^{sr_t} rt_{sr_i}}{sr_t} \tag{3.7}$$

The average number of hops anh_t aggregates the number of hops of the solved requests. It follows equation 3.8.

$$anh_t = \frac{\sum_{i=1}^{sr_t} nh_{sr_i}}{sr_t} \tag{3.8}$$

The system needs to evaluate changes to latency over time. Equation 3.9 formalises the variation to latency from a time $t-1$ to a time t.

$$v(art_t) = \frac{art_t - art_{t-1}}{art_{t_n}} \tag{3.9}$$

Definition 15. The manager combines these metric to define a **system utility function** (*eq.* 3.10). Coefficients show the priority of each goal in this thesis, and how they affect overall performance of *uDiscovery*. The rate of solved requests is most important as it reflects how many requests a gateway solves from the total of received requests. The rest of metrics can be applied, if and only if a request was solved before. The larger the rate of solved requests, the better. The average precision is next, as the consumer expects correct services in response to a request. *uDiscovery* assumes that accurate responses are more desirable, because incorrect responses cannot be used by consumers even if they take less time or network usage to be discovered. The larger the precision, the better. The average response time is third as it is desirable to retrieve services with low latency, but *uDiscovery* assumes that consumers are more interested in getting a correct answer even if it takes longer time. The lower the response time, the better. The average number of hops is fourth following the same assumption. It is important to minimise the number of visited gateways to solve a request from the network perspective (i.e., minimise network overhead). However, *uDiscovery* assumes that consumers are more concerned about getting a correct response in a short time rather than the number of visited gateways. Nonetheless, the lower the number of hops, the better as it means less network overhead. The percentage of used storage is less important than the other variables as *uDiscovery* assumes that it should be invisible to consumers and the storage capacity can be easily improved in an IoT gateway (e.g., upgrading the gateway's SD card).

$$Maximise : u(sd)_t = 5rsr_t + 4asp_t - 3v(art_t) - 2anh_t - pus_t \tag{3.10}$$

Figure 3.8 illustrates the processes that the *adaptive manager* follows to respond to unforeseen, scheduled, and periodic city events. The *manager* identifies unforeseen events by evaluating the utility value of the discovery process. The *adaptive manager* asks for services to other gateways based on the pheromones' information, if the utility value is less than

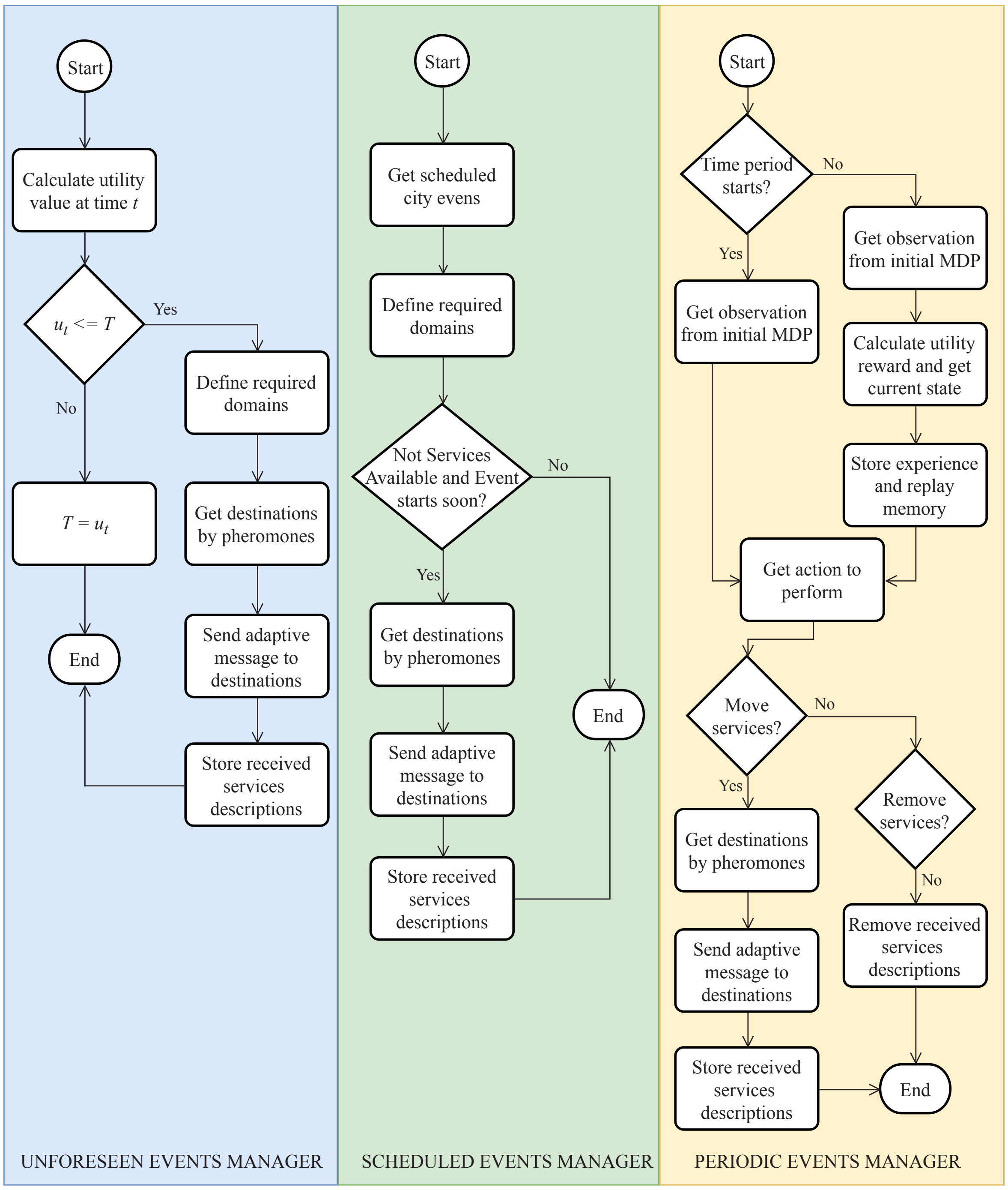

Start
Calculate utility value at time t
$u_t <= T$
Yes
No
Define required domains
$T = u_t$
Get destinations by pheromones
End
Send adaptive message to destinations
Store received services descriptions
UNFORESEEN EVENTS MANAGER
Start
Get scheduled city evens
Define required domains
Not Services Available and Event starts soon?
No
Yes
Get destinations by pheromones
End
Send adaptive message to destinations
Store received services descriptions
SCHEDULED EVENTS MANAGER
Start
Time period starts?
No
Yes
Get observation from initial MDP
Get observation from initial MDP
Calculate utility reward and get current state
Store experience and replay memory
Get action to perform
Move services?
No
Yes
Remove services?
No
Get destinations by pheromones
Remove received services descriptions
Send adaptive message to destinations
Store received services descriptions
End
PERIODIC EVENTS MANAGER

Figure 3.8: *uDiscovery* - Self-Adaptive Manager Processes.

the threshold T. Otherwise, the *manager* updates the threshold with the utility value to improve the discovery performance in a continuous fashion. The *manaager* checks scheduled events in the *local repository* and asks for services when the event is about to start, using the pheromones' information to determine which gateways are most likely to have services information related to the events domains.

The *adaptive manager* models the management of periodic events as a reinforcement learn-

Algorithm 5 Unforeseen Events Manager.

```
1: function UTILITYMONITORING($t$)                ▷ where $t$ is the current time
2:     $u_t \leftarrow calculateUtility(t)$
3:     if $u_t <= T$ then
4:         $RD \leftarrow defineRequiredDomains()$
5:         $destinations \leftarrow getDestinationsByPheromones(RD)$
6:         for each $gw \in destinations$ do
7:             $sendMessage(Adp_{msg}(gw_{id}, RD, 1))$
8:         $localRepository.insert(receivedServices)$
9:     else
10:        $T \leftarrow u_t$
```

ing problem, where the manager knows the environment states and possible actions. But it does not know the model of the environment (i.e., transitions between states, and rewards for each transition). Then, the manager must learn this model from its interaction with the environment. Model-free reinforcement learning algorithms are designed to handle such kind of complex environments. The *adaptive manager* uses a Deep Q-Network (DQN) algorithm [Mnih et al., 2015], which is a model-free reinforcement learning model that combines Deep Neural Networks and Q-Learning to learn about city's periodic events. The *manager* gets the action to perform under given circumstances from the DQN-algorithm, which determines if services must be moved between gateways or removed from the local repository. The *manager* provides the environment information to the DQN-based algorithm based on the utility function defined before (Equation 3.10). The DQN algorithm uses Deep Neural Networks to estimate the Q-values that the Q-Learning algorithm uses based on the environment information. The number of hidden nodes, and the learning rate are defined as parameters for this model in the Table 3.4 because they influence the performance of the neural networks to extract and learn features from the environment. Chapter 5 presents the effect of these parameters in the management of periodic events.

The rest of this section explains in details how the *manager* adapts the service organisation to respond to each type of city event.

3.4.2.1 Unforeseen Events Manager

The *unforeseen events manager* handles unforeseen events using the utility function (*eq.* 3.10). Algorithm 5 shows the reactive process that gateways perform in detail. Each gateway calculates the utility function at a given time t (Line 2). If the utility is lower than, or equal to, a threshold T, the gateway assumes that an event is happening. The gateway deduces which service domains are required by the city event based on unsolved requests that

Algorithm 6 Scheduled Events Manager.

```
function scheduledEventsMonitoring($t$)                ▷ where $t$ is the current time
    $E \leftarrow getScheduledEvent()$
    $RD \leftarrow requiredDomains(E)$
    if $\neg gatewayHasServices(domains)$ then
        if $event.startsSoon()$ then
            $destinations \leftarrow getDestinationsByPheromones(RD)$
            for each $gw \in destinations$ do
                $sendMessage(Adp_{msg}(gw_{id}, RD, 1))$
            $localRepository.insert(receivedServices)$
            while $eventIsHappening()$ do
                keep services
            $removeServices()$
        else
            do nothing
    else
        do nothing
```

negatively impacted the utility function (Line 4). The gateway sends an adaptive message asking for services from these domains to other gateways in the network, and stores responses. Gateways are selected according to the pheromones' information in the gw_i for the required domains following the equation 3.4, where D is the set of required domains (Lines 5 to 7). It guarantees that gw_i asks for services from the most promising known gateways according to the urban context represented in the pheromones' values. gw_i stores, in the *local repository*, the services that destinations send as a response to the services' request (Line 8). If the utility is greater than the threshold T, the gateway updates the threshold with the current utility value. This guarantees a continuous improvement of the system (Line 10).

Definition 16. An **adaptive message** is defined as $Adp_{msg} = \langle id_{rec}, RD, h \rangle$, where id_{rec} is the identifier of the receiver gateway, RD is the set of domains of the required services, and h is the message hops.

When a gateway gw_j receives an adaptive message, gw_j searches services that belong to the required domains in its *local repository* and sends them as response. If the gateway does not find services and the message hops is less than the hops limit, it forwards the adaptive message to relevant gateways in the network, which are selected according the pheromones' information in gw_j.

3.4.2.2 Scheduled Events Manager

The *scheduled events manager* handles foreseen events based on the knowledge model (Figure 3.5). This model defines the concepts city event, time (i.e., temporal dimension), and city authority. Events happen at a city place and require city services. They have start and end times and can be scheduled by authorities. Times are represented in days, hours, minutes and seconds. A city authority schedules city events in the system by specifying the place where it happens and the start and end times. Algorithm 6 presents how scheduled events are managed in detail. A gateway gw_i retrieves information about scheduled events in its surroundings, and determines which service domains are required by these events (Lines 2 and 3). If gw_i does not manage services for any of the required domains, it evaluates whether the start time of the event is soon (Lines 4 and 5). gw_i sends requests for services from the required domains to other gateways before the event starts. Gateways are selected according to the pheromones' information in gw_i for the required domains following the equation 3.4, where D is the set of required domains. gw_i stores the received services and keeps them until the event ends (Lines 6 and 12).

3.4.2.3 Periodic Events Manager

Periodic events follow a pattern in the city because of the interaction of its entities. Such interactions are repetitive as entities have a similar behaviour at a similar time (e.g., traffic congestion at peak hours). The *periodic events manager* should move services according to these events because they might have an impact in the consumers' services requests. For example, citizens might ask for transportation related services at peak hours. The *periodic events manager* tries to learn these patterns from the citizens' requests to proactively reorganise services. The *periodic manager* defines a Markov Decision Process to learn from citizens' requests. The *manager* knows knows the environment states S and possible actions A to perform, but it does not know the model of the environment. Such model is composed by the conditional distribution $P(s'|s,a)$ of next states given a current state and an action, and the reward function of transitioning from an state s to s', $P(s'|s,s')$. The *manager* must learn the environment model from experience (i.e., moving services, observing utility value, and generalising experiences). Current research in reinforcement learning has proposed model-free approaches to learn from complex environments when models are unknown. *uDiscovery* uses the DQN algorithm [Mnih et al., 2015], which combines Deep Neural Networks and Q-Learning to learn about city's complex behaviour. The *periodic manager* aims to find successful adaptive policies that proactively move services and improve the discovery efficiency in smart city environments. To this end, the *periodic manager* starts by defining the city states, which are composed of a time and the service organisation status, and the possible actions, which enable the exchange of services between gateways. The manager

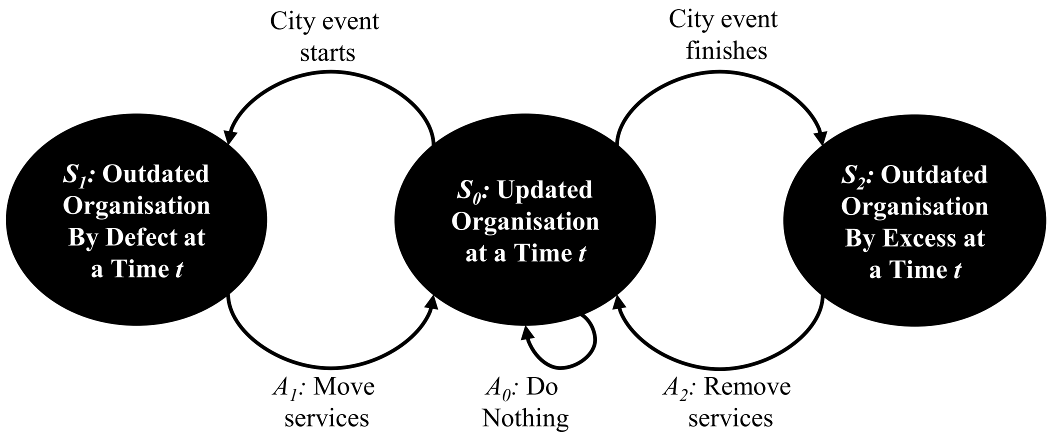

City event starts
City event finishes
S1: Outdated Organisation By Defect at a Time t
S0: Updated Organisation at a Time t
S2: Outdated Organisation By Excess at a Time t
A1: Move services
A0: Do Nothing
A2: Remove services

Figure 3.9: *uDiscovery* - City States Model.

knows the time according to the knowledge model (Figure 3.5) and defines time periods, city states, and possible actions as follows:

Definition 17. A **time period** starts at hour 0 and ends at hour n, after which another period starts. For instance, a day period starts at hour 0 and ends at hour 24.

Definition 18. uDiscovery defines three **city states** and three **actions** to capture a service organisation status 3.9. S_0 represents the state when the service organisation is up to date at time t. S_1 represents an outdated service organisation by defect (i.e., missing services) at a given time t after an event starts. S_3 represents an outdated service organisation by excess (i.e., there are too many services) at a time t after an event ends. The manager performs three actions, depending on the city state. The manager does nothing if the system is in state S_0 (i.e., A_0); the manager sends adaptive messages to other gateways if the system is in state S_1 (i.e., A_1); and the manager removes services if the system is in state S_2 (i.e., A_2).

The *periodic events manager* links the time periods to the states to create its view of the city. For example, the manager can determine that the service organisation is updated at a given hour in a time period. The *manager* does not know the reward values on transitioning between states. But, it knows the impact of performing an action in the utility function. Equation 3.11 formalises this impact and enables the *manager* to observe and store the impact of an action in the service organisation at a given state. This equation defines that if the utility function increases from one time t_i to another time t_j, the reward will be positive. Otherwise, the reward will be negative.

$$r_t = u(sd)_t - u(sd)_{t-1} \tag{3.11}$$

Algorithm 7 Periodic Events Manager.

```
Require:
    $DQN\ dqn$
    $Action\ action$
    $Observation\ obs$
function PERIODICEVENTSMONITORING($t$)        ▷ where $t$ is the current city time
    if $t.period.hour = 0$ then
        $obs \leftarrow dqn.initMDP().lastObs()$
    else
        $u_t \leftarrow calculateUtility(t)$
        $u_{t-1} \leftarrow calculateUtility(t-1)$
        $r_t \leftarrow u_t - u_{t-1}$
        $s_t \leftarrow getCityState(r_t, t)$
        $obs \leftarrow dqn.storeAndReplay(obs, action, r_t, s_t)$
    $action \leftarrow dqn.getActionToPerform(obs)$
    if $action = A_0$ then
        $do\ nothing$
    if $action = A_1$ then
        $RD \leftarrow defineRequiredDomains()$
        $destinations \leftarrow getDestinationsByPheromones(RD)$
        for each $gw \in destinations$ do
            $sendMessage(Adp_{msg}(gw_{id}, RD, 1))$
        $localRepository.insert(receivedServices)$
    if $action = A_2$ then
        $localRepository.remove(receivedServices)$
```

Algorithm 7 shows how the *periodic manager* tries to learn from periodic events based on the DQN algorithm. This algorithm requires a *DQN* object that implements the algorithm defined by Mnih et al. [Mnih et al., 2015], a variable *action* that stores the last action of the manager and a variable to store the last *observation* of the system about the city. The *DQN* object uses a Markov Decision Process (MDP) model, which includes the city states space, to define the status of the service organisation at a given hour in a time period, and the set of possible actions to make in a given state. The size of the states space is equal to the number of hours of the time period (e.g., 24) multiplied by the number of possible actions (i.e., 3). Initial states represent the status of the service organisation when a time period starts (e.g., updated organisation at hour 0). Final states represent the status of the service organisation when a time period ends (e.g. outdated organisation by excess at hour 24).

The *periodic manager* executes Algorithm 7 and starts by checking the time. If the time period is about to start (i.e., at hour 0), the manager gets a city observation from the city *MDP*, which represents the initial city state (i.e., the service organisation status at hour 0) (Lines 2 and 3). Otherwise, the manager calculates the system reward at the current hour and determines the new city state (Lines 5 to 8). If the reward is positive, the organisation

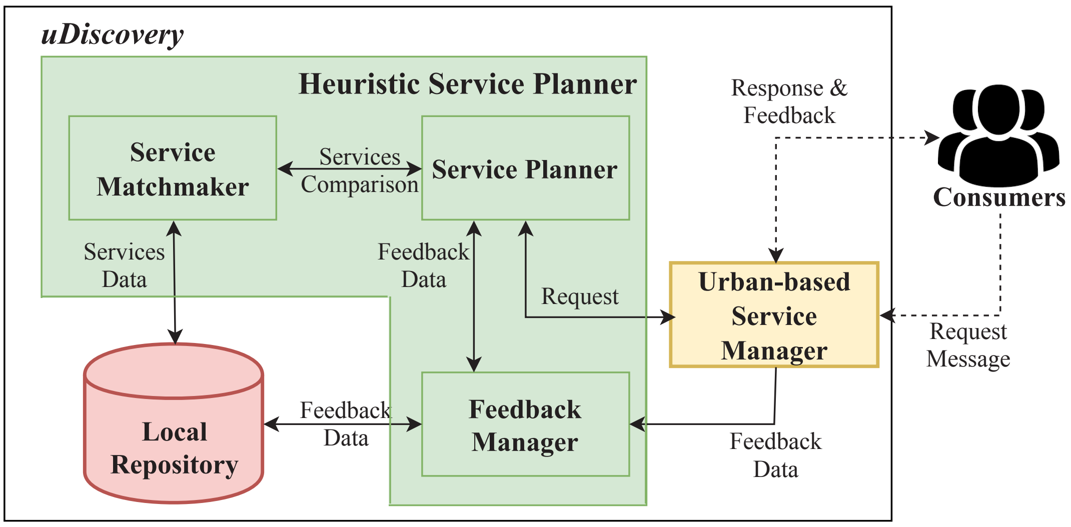


Figure 3.10: *uDiscovery* - Service Planner Architecture.

of services is updated. Otherwise, the organisation of services is ”outdated by defect” (i.e., there are services missing) when the rate of solved requests, search precision, response time or number of hops metrics are negatively affected. Or, it is ”outdated by excess” (i.e., when there are too many services) when the percentage of used storage has increased. The *manager* gets a new observation from the DQN algorithm according to the last observation, the last action, the current reward and the current state (Lines 8 and 9). Finally, the manager retrieves the action to perform and acts according to the new observation (Lines 10 to 17). *uDiscovery* splits the DQN algorithm into two functions to integrate the city's information (i.e., city state, and service discovery reward). The first function (line 10) defines the action to perform given a system observation by following the $\epsilon - greedy$ policy that DQN implements. The second function is invoked (line 9) after at least one action has been performed by the manager. This function stores a transition in the manager's replay memory (i.e., DQN tuple that stores systems transitions), performs experience replay to learn from previous transitions (i.e., DQN's experience learning model), and returns the next observation.

3.4.3 Service Planning based on Consumers' Feedback

uDiscovery searches for services in gateways local repositories using a *heuristic service planner* based on consumers' feedback. Local service search can also impact discovery efficiency because of the exploration of all services combinations, and the inclusion of incorrect services as discussed in Chapter 2. The *heuristic service planner* aims to improve local search efficiency by implementing a progressive search with minimal consumer's input. Such search reduces search spaces in sequential steps, uses consumers' feedback, to avoid the exploration

Parameter	**Description**
Feedback Threshold	A double between 0 and 1 that defines the threshold to validate a discovered edge in a service plan
Functional Threshold	A double between 0 and 1 that defines the threshold to explore a service plan according to how well it addresses functional requirements
K-value	An integer that defines the top-K candidates to rank and explore

Table 3.5: Heuristic Service Planner Parameters.

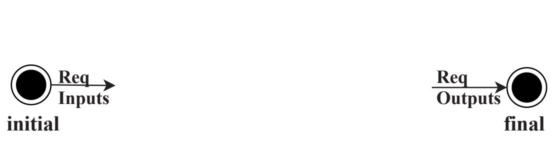
of incorrect services, and ranks plans according to their functional properties to explore the most promising ones. *uDiscovery* defines consumer's feedback as a Boolean mark that indicates the success (i.e., the mark is 1) or failure (i.e., the mark is 0) of a discovered plan at execution time. *uDiscovery* uses a Boolean value because it is the minimal input that can be expected from consumers. This mark can be provided by service-oriented components or applications that execute the discovered plan and can determine if the execution was successful or fails [Palade et al., 2018]. Figure 3.10 shows the detailed architecture of the planner. It is composed by three sub-components. The *service planner* receives requests from the *urban-based service manager* and creates service plans by using the *service matchmaker*. This matchmaker matches requests against services in the local registry based on semantic and syntactic methods. The planner validates these comparisons using the *feedback manager* that manages feedback information. It stores feedback data that the *urban-based service manager* receives from consumers in the *local repository*. The *heuristic planner* creates service plans based on consumers' feedback to satisfy consumer's needs.

Table 3.5 shows the parameters of the *planner*. This model uses a $feedback\ threshold$ to validate discovered service relations against consumers' feedback, and a $functional\ threshold$ to determine if a plan meets functional requirements The *planner* also uses a K-value to limit the number of explored plan candidates in the planning process. Chapter 5 experiments with different values for these parameters and reports $uDiscovery's$ performance results.

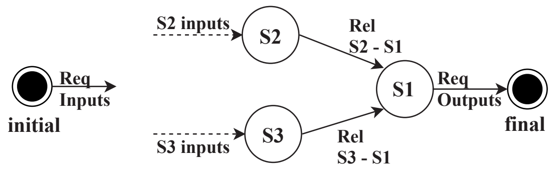
The *heuristic service planner* follows next definitions and processes:

Definition 19. A **service plan** is a directed graph $G(v, e)$, where vertices v are services and edges e are the input/output relations between them (e.g., the output of the service $S2$ in the 3.11c is an input of the service $S1$).

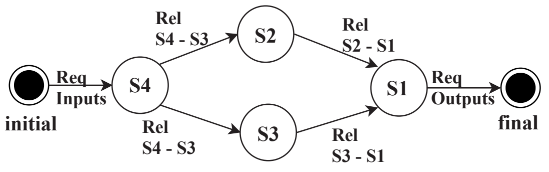
Figure 3.11 shows services plans at different stages in the planning process. Figure 3.11a



(a) Initial Plan.



(b) Partial Plan.



(c) Finished Plan.

Figure 3.11: *uDiscovery* - Services Plans.

presents the initial stage where the plan has two vertices (i.e., initial and final) that represent the request inputs and outputs. Figure 3.11b shows an intermediate state where the plan is partial because not all the required I/O are satisfied. Figure 3.11c presents the final plan where the graph is completed and all I/O are satisfied.

The *feedback manager* receives feedback from consumers, and updates the *local repository* where the data of discovered plans is stored. There are two data structures that store this historical information:

- Plans: This stores past discovered plans as graphs that include vertices (i.e., services in a plan), and edges (i.e., relations between two services in a plan).
- Relationships: This stores each relation between two services (i.e., an edge between two vertices) discovered by the matchmaker in past plans. Each relation includes the source parameter type, the source service domain, the target parameter type, the target service domain, the number of past plans that have included this relation, and the number of successful plans that have included this relation. This counter of successful plans is updated when a consumer sends a feedback of 1 (i.e., the discovered plan was successful) related to a plan that includes the relation. If the plan that includes the relation failed (i.e., the feedback value is 0), the feedback manager only updates the counter of plans that have included the relation.

The services and requests parameters are semantically annotated as shown in the Figure 3.1 in the Section 3.2. The *service matchmaker* uses these annotations to reduce the search space and identify relations between services to build service plans. Each identified relation is validated against the consumer feedback in the *local repository* where the *feedback manager* stores the knowledge about past discovered plans. This knowledge represents relationships between domains that are extracted from consumers' feedback thanks to the inclusion of source and target domains in each stored relationship. This avoids the inclusion of services that have input/output relationships but belong to non-related domains. For example, service s_1 has body temperature as output and belongs to the health domain, service s_2 has room temperature as input and belongs to the buildings domain. They have an input/output relationship as s_1 produces, and s_2 consumes temperature measurements, but it is incorrect and will receive a negative feedback when it appears in a plan. It is important to note that *uDiscovery* builds knowledge about relationships between domains automatically from historical searches, it does not require previous definitions from human experts.

Figure 3.12 shows the discovery process where the *service planner* searches for plans to meet consumers' request. This process starts when a gateway gw_i receives a discovery message from consumers or other gateways (Algorithm 4), and the *request manager* calls the *heuristic planner* and passes the following input parameters:

- Request r: This is the requirement defined by the consumer and includes the inputs and outputs of the service that the consumer needs.
- Previous Solutions $PSOL$: This is a set of plans that partially solve the request and was discovered previously. It is a list of partial graphs 3.11b, if the message comes from other gateway, or an empty list if the message comes from the consumer.

The *heuristic service planner* performs a planning process based on these inputs as follows:

Step 1: Initial Plans and Search Space Definition

The *planner* receives the request and previous plans and initialises the plans that are going to be manipulated in next steps. If the list of previous plans is empty (i.e., the message comes from the consumer), the *planner* creates a new graph with two vertices: initial and final. The outputs of the initial vertex are the inputs in the request and the inputs of the final vertex are its outputs (Figure 3.11a). The final vertex is an unsolved vertex as there are no vertices in the graph that provide their inputs. If the list of previous plans is not empty, these partial plans are used as initial graphs (Figure 3.11b). The goal is to build a complete graph where the initial and final vertices are linked through a chain of solved vertices meeting consumer requirements(Figure 3.11c).

The *planner* defines the search space for each initial plan. This search space is a set of

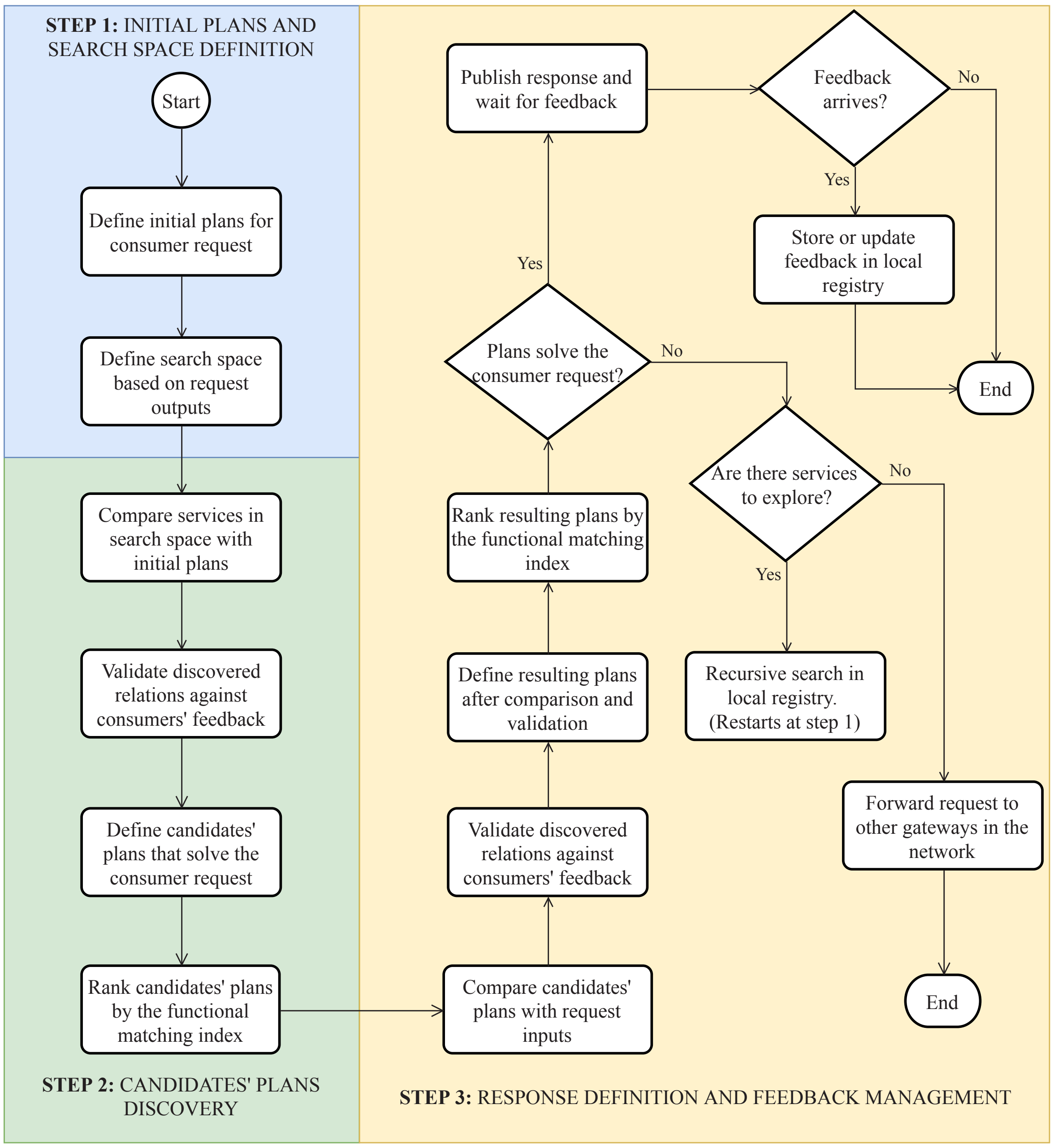

STEP 1: INITIAL PLANS AND SEARCH SPACE DEFINITION
Start
Define initial plans for consumer request
Define search space based on request outputs
Compare services in search space with initial plans
Validate discovered relations against consumers' feedback
Define candidates' plans that solve the consumer request
Rank candidates' plans by the functional matching index
STEP 2: CANDIDATES' PLANS DISCOVERY
Compare candidates' plans with request inputs
Validate discovered relations against consumers' feedback
Define resulting plans after comparison and validation
Rank resulting plans by the functional matching index
Plans solve the consumer request?
Yes
No
Publish response and wait for feedback
Feedback arrives?
Yes
No
Store or update feedback in local registry
End
Are there services to explore?
Yes
No
Recursive search in local registry. (Restarts at step 1)
Forward request to other gateways in the network
End
STEP 3: RESPONSE DEFINITION AND FEEDBACK MANAGEMENT

Figure 3.12: *uDiscovery* - Service Planner Processes.

services where their outputs match unsolved vertices in the plan. The *planner* queries the repository to get these services according to their I/O. This query reduces the search space and improves the search efficiency as the *planner* does less iterations in next steps (i.e., coarse-to-fine progressive search) [Pattar et al., 2018]..

Step 2: Candidates' Plans Discovery

The *planner* tries to complete the initial plans using the services in the search space. It uses the *service matchmaker* to discover I/O relations (i.e., edges in the graph) between the outputs of each service and the inputs of each unsolved vertex in each initial plan. The *matchmaker* includes semantic and syntactic methods to compare two service parameters, based on previous works [Urbieta et al., 2015]. The *matchmaker* gives a score to the discovered edges that is used to rank and select the candidate plans (i.e., plans that can solve the request) in the next steps. The matching methods compare two service parameters using the semantic annotations. Each parameter P has a type that corresponds to a concept in an ontology O and each method checks the relation between them according to the ontology.

Definition 20. uDiscovery uses the matching methods already defined in the literature to discover relations between services inputs and outputs [Klusch et al., 2016]. *uDiscovery* extends these methods by giving a score to each identified service relation according to the strength of the matching method, as follows:

- $equivalent(P_1, P_2)$: The type of P_1 is conceptually equivalent to the type of P_2 in O. This method gives a score of 4 to the relation between P_1 and P_2, as this semantic method is the strongest because it only match parameters that are semantically equal (i.e., it has less chance to introduce false positives).

- $plugin(P_1, P_2)$: The type of P_1 is a sub-concept of the type of P_2 in O. This method gives a score of 3 to the relation between P_1 and P_2, as this semantic method is stronger than *subsume* and the syntactic similarity method, but weaker than *equivalent*.

- $subsume(P_1, P_2)$: The type of P_1 is a super concept of the type P_2 in O. This method gives a score of 2 to the relation between P_1 and P_2, as this semantic method is stronger than the syntactic similarity method.

- $similarity(P_1, P_2)$: This method calculates the syntactic similarity between P_1 and P_2 using cosine similarity (Eq 3.12) as this produces most accurate results according to previous work [Urbieta et al., 2017]. This method gives a score of 1 to the relation between P_1 and P_2, as this is the weakest method because is more likely to introduce false positives.

$$cos(P_1, P_2) = \frac{Pv_1 \cdot Pv_2}{||Pv_1|| \cdot ||Pv_2||} \tag{3.12}$$

where Pv_1 and Pv_2 are vectors with the words that describe P_1 and P_2.

The *matchmaker* discovers relations between services in the search space and the unsolved vertices in each plan. The *planner* validates each discovered relation against the consumers' feedback in the *local repository* using the *feedback manager*. This validation avoids the inclusion of unexpected services in the plans. It means that a discovered relation is ignored if the plans that included it before were incorrect. The *planner* uses the historical data from previous searches to compute an historic success index (HSI) for each discovered relation (i.e. an edge in a plan). The *planner* compares this index against a feedback threshold to determine if the discovered relation is valid. The historic success index is calculated using equation 3.13.

$$HSI(e) = \frac{successfulPlans}{totalPlans} \tag{3.13}$$

where $successfulPlans$ is the number of correct plans that have included the edge e and $totalPlans$ is the number of plans that have included the edge e. The *planner* uses this index to determine whether a discovered relation is valid (i.e., the edge should be included or not in the plan). If $HSI >= feedbackThreshold$, the discovered relation is valid, otherwise it is ignored. The *planner* creates a candidate plan (i.e., a plan that potentially solves the request) for each combination of valid discovered relations for an initial plan. Previous planning approaches create and explore candidates for all combinations, which increases their latency. The number of candidates can be large even with this constraint. The *service planner* selects the top k candidates based on the scores defined in the matching process. Each candidate has a functional matching index (FMI) that defines how well the candidate meets the request's functional requirements. This index is computed according to eq. 3.14.

$$FMI(G(v,e)) = \frac{\sum_{i=1}^{n} x_i}{4n} \tag{3.14}$$

where n is the number of edges in graph G (i.e., the graph that represents the candidate plan), and x_i is the score for a particular edge according to the matching method that discovers it. $4n$ is the maximum value that the plan can receive, which means that all edges in the graph were discovered by the strongest matching method (i.e., $equivalent(P_1, P_2)$). The candidate is added to the list of candidates if the length of the list is less than or equal to K and $FMI >= functionalThreshold$.

Step 3: Response Definition and Feedback Management

The *planner* has a list of the top candidate plans that can solve the request at this point. It decides which of these plans effectively solve the request. The *matchmaker* compares the outputs of the initial vertex (i.e., inputs of the request) with the inputs of the non-solved

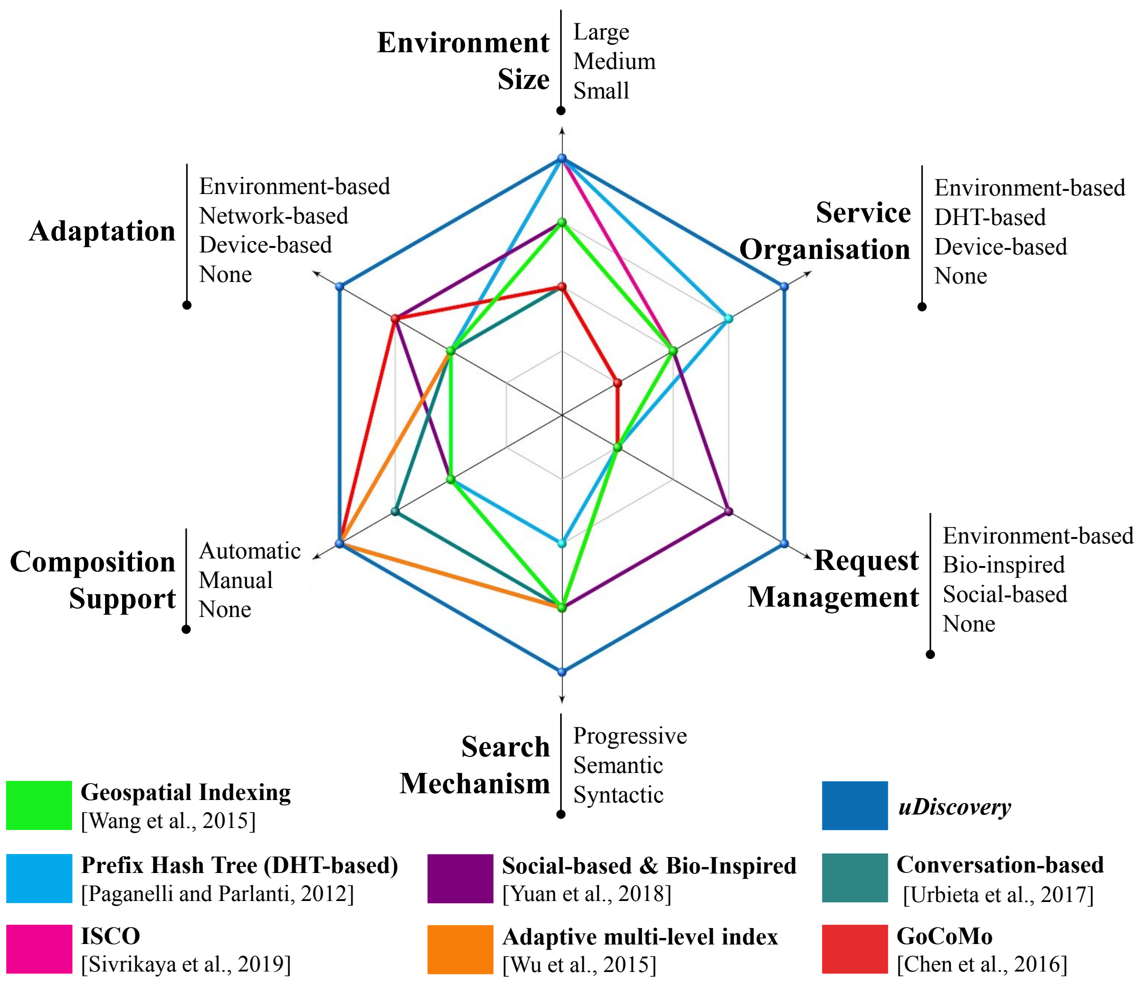


Figure 3.13: *uDiscovery's* features compared to closest approaches.

vertices for each candidate using the previous matching methods (i.e., *equivalent*, *plugin*, *subsume*, and *similarity*). The *serviceplanner* validates the discovered relations calculating the *HSI* and comparing it with the *feedbackThreshold* as defined in the previous step. If the request is not solved after this comparison and there are services in the repository to explore, it starts the process again in a recursive fashion. The *matchmaker* returns the list of plans to the gateway gw_i which continues the discovery process according to Algorithm 4. The *feedback manager* stores the discovered plans in the *local repository.* Then, it subscribes to a feedback message from the consumer, which can send its feedback as a boolean mark according to the success or failure of the discovered service at execution time. The consumer publishes feedback sending the plan identifier and the Boolean mark. The *feedback manager* stores this feedback in the structures defined before.

3.5 Design Summary

This chapter introduces *uDiscovery* and the design of its contributions to address open issues in the discovery of services in smart cities. *uDiscovery* is an urban-based model that improves discovery efficiency in large scale and dynamic city environments.

uDiscovery formalises this urban context in a knowledge model that includes the city concepts that interact in the service discovery domain. *uDiscovery* configures a network of IoT gateways that manage services in the city based on this knowledge model. It annotates gateways with their surrounding city places information. Each gateway decides how to organise services in the network according to these annotations to put the services information in the *right place*. *uDiscovery* solves consumers' requests in the network of gateways by forwarding them where they are most likely to be discovered. It uses a bio-inspired method on top of the urban based service organisation to propagate the information in an environment where each gateway has a partial knowledge about other gateways in the network. *uDiscovery* adapts the service organisation according to three types of events (i.e., unforeseen events, scheduled events, and periodic events). *uDiscovery* senses the impact of these events in the discovery metrics, and the city state to move services between gateways and keep a good discovery efficiency over time.

uDiscovery searches for services in local repositories using a heuristic planner based on consumers' feedback. *uDiscovery* improves search efficiency and minimises human input. Consumer feedback (i.e., historical data from previous searches) is used to improve search accuracy. *uDiscovery* improves search latency by avoiding the exploration of incorrect combinations of services. Second, the model explores the plans according to how well each plan meets the request's requirements.

uDiscovery enables a self-adaptive service discovery model based on the environment context. *uDiscovery* puts the right service in the right place at the right time in preparation for discovery, and also forwards requests where they are most likely to be solved. *uDiscovery* offers an automatic planning that supports service composition and improves search efficiency minimising consumers' input. Figure 3.13 compares *uDiscovery* features with the closest approaches from the literature. *uDiscovery* contributions close the gap of current approaches with regard to an adaptive service discovery based on the environment context in large environments. The remaining of this thesis describes the implementation, evaluation and limitation of *uDiscovery*.

Chapter 4

Implementation

Chapter 3 describes the design of *uDiscovery* and the decisions to address the service discovery challenges in smart cities. This chapter details the implementation of *uDiscovery*. Section 4.1 introduces the architecture, data models and structural models that support *uDiscovery*. Section 4.2 details initialisation, registration, and discovery processes. Section 4.3 details the adaptive processes. Finally, Section 4.4 summarises this chapter.

4.1 uDiscovery Architecture

Figure 4.1 illustrates the architecture of *uDiscovery*, which is composed by four main subcomponents as follows:

Urban-based Service Manager

This component manages city, gateways, services, and requests information to initialise and maintain the network of IoT gateways that discovers services based on urban-context according to Section 3.4.1. This manager is composed of three components as follows:

- **Initialisation Manager:** This component uses urban context from the local repository to initialise and maintain the network of gateways according to Section 3.4.1.2. It recognises surrounding places and defines the gateway relevance. It exchanges information with other gateways in the network through gateway advertisement messages to create links that support the discovery of services based on urban context.

- **Service Organisation Manager:** This component receives service registration messages from providers or other gateways in the network. It stores and advertises the services information to other gateways based on urban context based on Section 3.4.1.3.

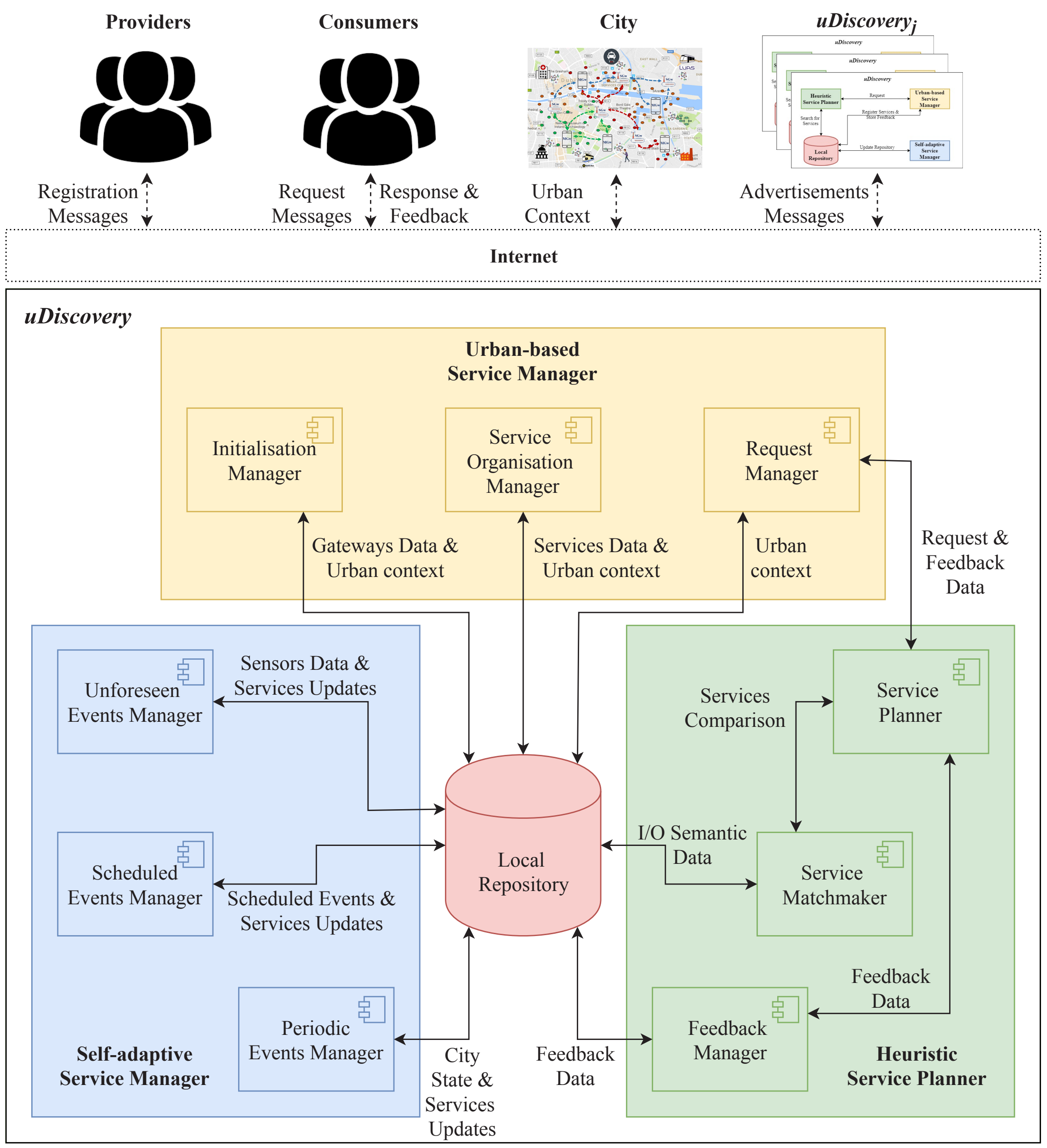


Figure 4.1: *uDiscovery* - Detailed Architecture.

- **Request Manager:** This component receives requests from consumers or other gateways. It tries to solve these requests by using the heuristic service planner component (Section 3.4.1.4). This component either sends a response or advertise the request to other gateways, once a request is processed by the planner and an output is generated (i.e., requests is solved, partially solved, or not solved). It also receives and propagates feedback information to other gateways in the network.

Self-adaptive Service Manager

This component is responsible for the adaptive processes (Section 3.4.2). It exchanges messages with other gateways in the network to move services information. This component recognises and responds to city events based on next components:

- **Unforeseen Events Manager:** This component responds to unforeseen events in the city (Section 3.4.2.1). It gets information from sensors and acts accordingly to update the local repository and improve the system performance.
- **Scheduled Events Manager:** This component recognises scheduled events and acts accordingly to move services where they are likely to be needed according to Section 3.4.2.2.
- **Periodic Events Manager:** This component identifies city states and moves services between gateways based on historical learning from city behaviour according to Section 3.4.2.3.

Heuristic Service Planner

This component searches for services in the local repository. It implements a heuristic goal-driven planning process (Section 3.4.3) on top of three components as follows:

- **Service Planner:** This component implements the progressive service planning based on consumers' feedback. It builds graphs of services by using the service matchmaker and the feedback manager to compare service and requests parameters.
- **Service Matchmaker:** This component implements semantic and syntactic methods to compare services and requests parameters. It uses the semantic annotations from the local repository to match this information.
- **Feedback Manager:** This component retrieves and updates consumers' feedback in the local repository. It receives requests from the service planner and feedback information from the request manager.

Local Repository

The local repository is composed of a non-relational database and a set of ontologies, which are detailed in the next section. Each gateway has a local repository where data is queried, stored, updated, and removed to support different *uDiscovery* processes

4.1.1 uDiscovery Data Model

uDiscovery works with of a data model formulated as JSON documents. The local repository database stores documents about services, urban context, system metrics, city events, dis-

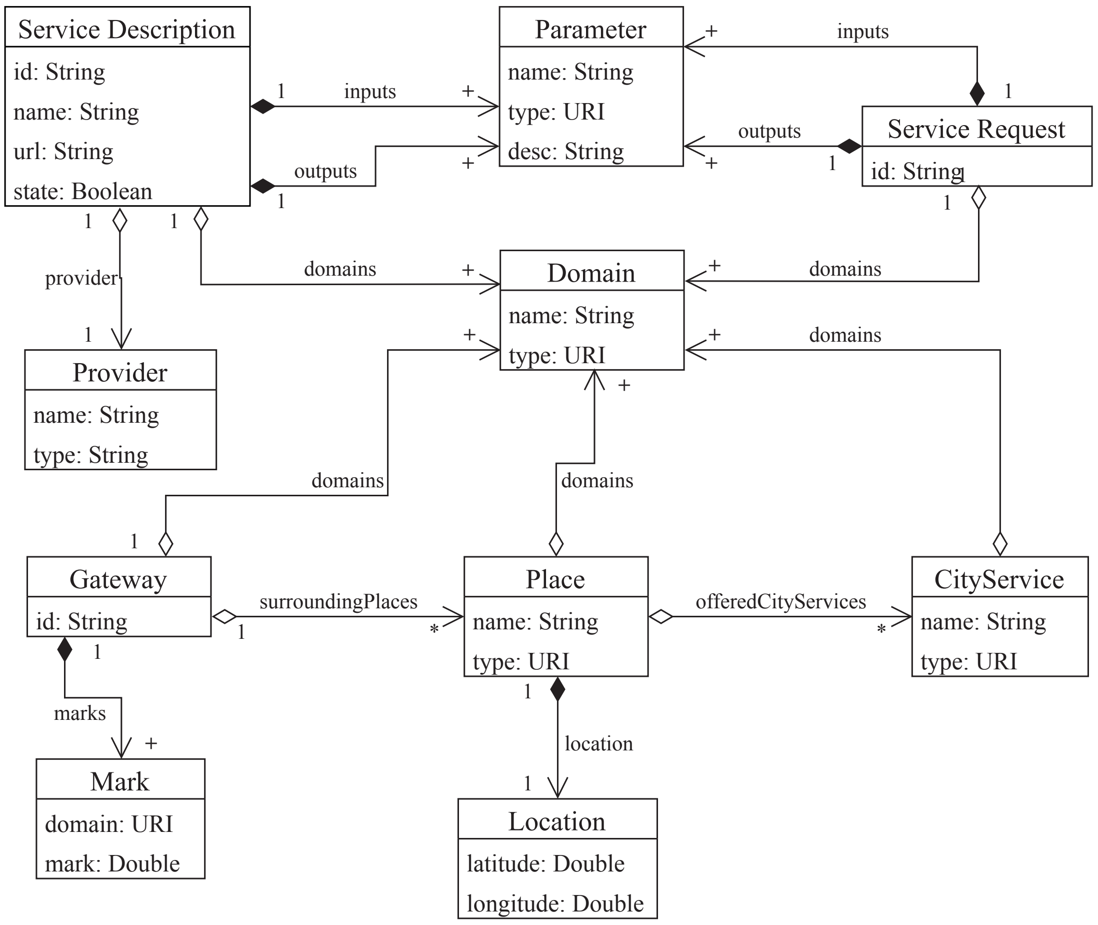


Figure 4.2: *uDiscovery* - Urban based Service Manager Data Model.

covered plans and consumers' feedback. Ontologies in OWL format store the urban context (Section 3.2), describe services and requests, and support semantic matching.

Figure 4.2 shows the data entities that interact in the *uDiscovery's* service organisation and discovery processes (Sections 3.4.1). They are described as follows:

- **Service Description:** This entity represents descriptions that *uDiscovery* registers, organises, and discovers. Each description has a service id, name, URL (i.e., service endpoint), and state (i.e., active or inactive). This entity includes the provider information, inputs and outputs parameters, and a list of service domains.

- **Provider:** This entity represents providers that register services in *uDiscovery*. Each provider includes a service name and type (e.g., WSNs, Web Server).

- **Parameter:** This entity represents service or request parameters. A parameter has a name, a type which corresponds to a concept in the knowledge model (i.e., URI), and a description field that gives extra information about the parameter.

- **Domain:** This entity represents city domains (e.g., educational domain). Each domain has a name and a type which corresponds to a concept in the knowledge model (i.e., URI).
- **Service Request:** This entity represents requests from consumers that *uDiscovery* solves. Each request has an identifier, one or more input parameters, one or more output parameters, and a set of domains.
- **Gateway:** This entity represents gateways information which *uDiscovery* exchanges to initialise and maintain the network of gateways. A gateway has an identifier, a list of surrounding places, a list of domains, and marks that define the relevance of the gateway for each domain.
- **Place:** This entity represents a city place which is the main context attribute that *uDiscovery* uses. A place has a name, an URI that corresponds to a concept in the city knowledge model, a list of offered city services, and a location.
- **Mark:** This entity represents the relevance of a gateway for a given domain. Each mark has a domain defined by an URI, and a numeric mark.
- **City Service:** This entity represents a service which cities or places offer to their citizens (e.g., health care service). Each city service has a name, an URI that corresponds to a concept in the knowledge model and a list of domains.
- **Location:** This entity represents a location in the city (i.e., coordinates).

Figure 4.3 illustrates the structures and entities that support the adaptation processes (Section, 3.4.2). They are described as follows:

- **Sensors:** This structure stores the service discovery metrics that the adaptive manager uses to evaluate *uDiscovery* performance. It stores the number of solved requests, the accumulated search precision, the accumulated response time, the accumulated number of hops, and a measure of the storage used during the discovery processes.
- **Aggregators:** This entity represents the aggregated data of the discovery performance (i.e., sensors measurements). An aggregator has the rate of solved requests, average search precision, average response time, average number of hops, and percentage of used storage at a given time.
- **Utilities:** This structure stores the history of *uDiscovery* utilities. A utility has a value at a given time.

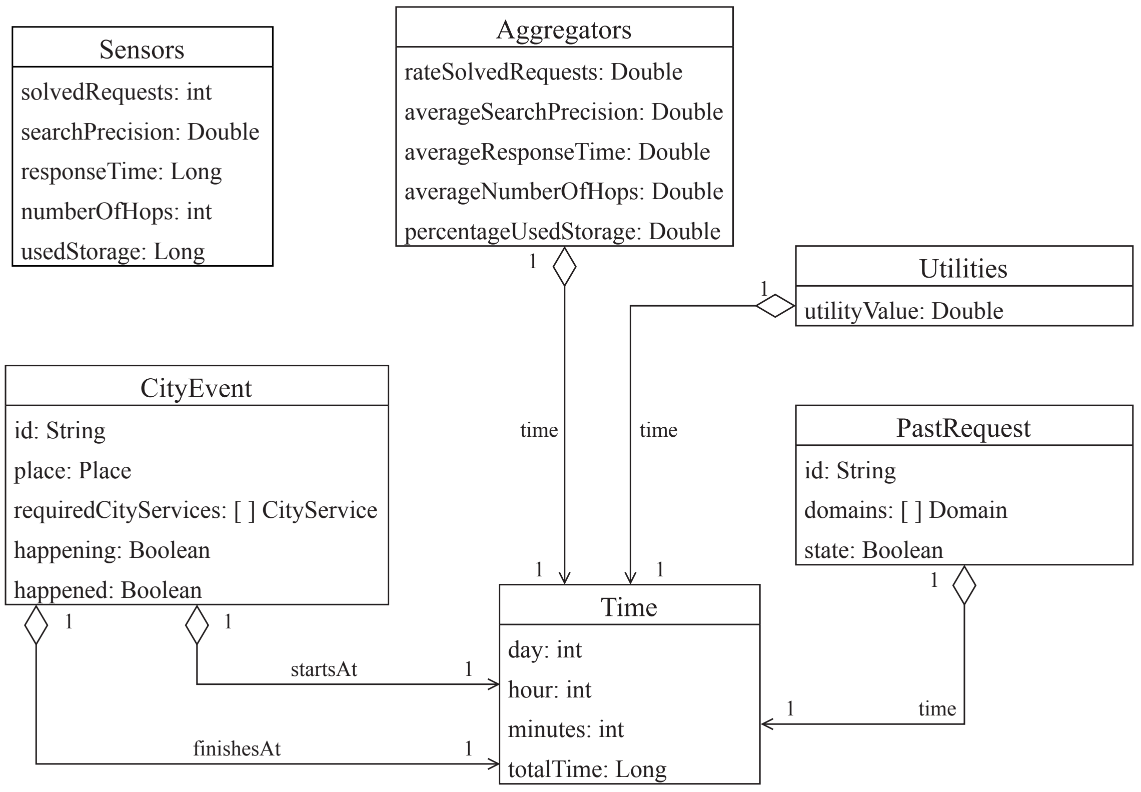


Figure 4.3: *uDiscovery* - Self-Adaptive Service Manager Data Model.

- **City Event:** This entity represents an event in the city. It has an id, a place where it occurs, a list of required city services, two attributes that indicate whether the event is happening or has happened, and a start and finish times.

- **Past Requests:** This entity represents a past request and is used by the adaptive manager to define the required domains (i.e., the domains of the unsolved requests). Each request has an identifier, a list of domains, and a state that indicates whether the requests was solved.

- **Time:** This entity represents the time in the city. Each time entity has day, hour, minutes and a total time which is the system time in milliseconds.

Figure 4.4 illustrates the entities that the heuristic planner uses (Section 3.4.3). They are described as follows:

- **Graph:** This entity represents a service plan discovered by the planner. Each graph (Figure 3.11) has a set of input and output parameters, a set of domains and a mark that reflects its functional matching index. A graph is composed of two or more vertices and zero or more edges.

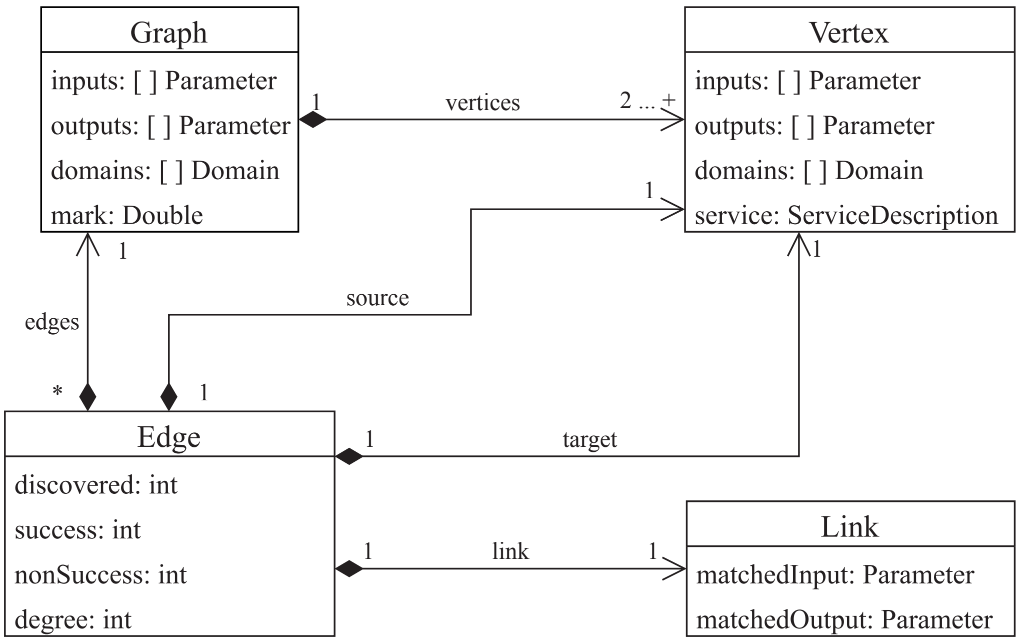


Figure 4.4: *uDiscovery* - Heuristic Service Planner Data Model.

- **Vertex:** This entity represents a vertex in a graph. Each vertex has a set of inputs and outputs, a set of domains and a service description.
- **Edge:** This entity represents an edge between two vertices in the graph. Each edge has a source and target vertex, a link, and stores feedback information as the number of times that the edge has been discovered before, the number of times that the edge has been part of a successful discovery process, the number of times that the edge has been part of an unsuccessful discovery process, and the degree of the matching defined by the matchmaker.
- **Link:** This entity represents the parameters that link two vertices in an edge. Each link has an input and output parameters that matched.

uDiscovery's knowledge model (Figure 3.5) is populated using information from previous works as follows: subclasses for the concept *Place* were extracted from OpenStreetMap (OSM)[1] that define physical features of a city (e.g., College, Road, Dentist). City services are defined based on the ontology KM4City[2] [Nesi et al., 2016] as subclasses of the concept *CityService* in the *uDiscovery's* knowledge model (e.g., accommodation service, cultural service). The classes and individuals for the concept *Domain* were defined according to literature, which defines IoT and smart city domains [Al-Fuqaha et al., 2015]. The knowledge

[1]OSM Features - `http://wiki.openstreetmap.org/wiki/Map_Features`
[2]KM4City Ontology - `http://www.km4city.org/`

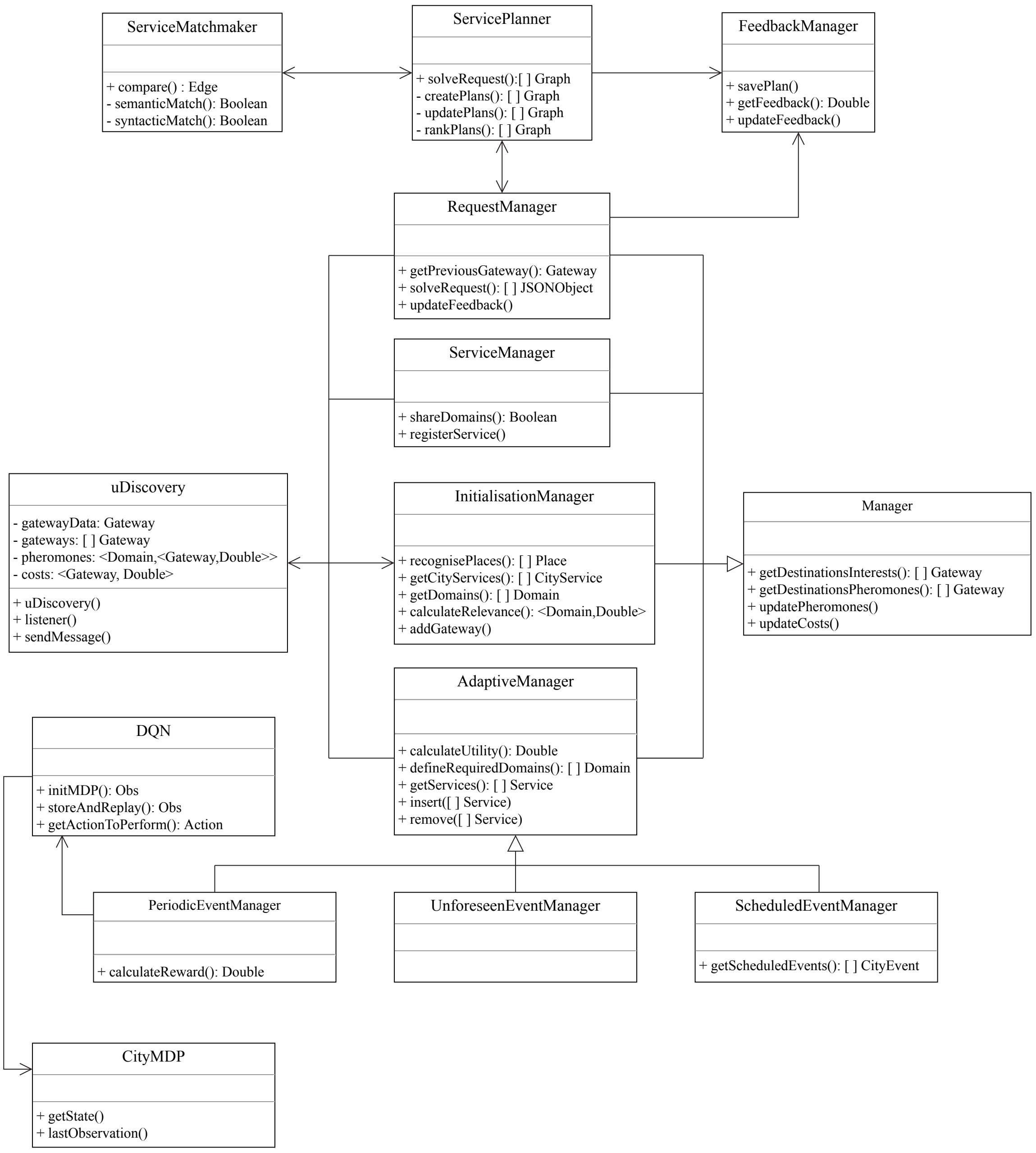

ServiceMatchmaker
+ compare() : Edge
- semanticMatch(): Boolean
- syntacticMatch(): Boolean
ServicePlanner
+ solveRequest():[ ] Graph
- createPlans(): [ ] Graph
- updatePlans(): [ ] Graph
- rankPlans(): [ ] Graph
FeedbackManager
+ savePlan()
+ getFeedback(): Double
+ updateFeedback()
RequestManager
+ getPreviousGateway(): Gateway
+ solveRequest(): [ ] JSONObject
+ updateFeedback()
ServiceManager
+ shareDomains(): Boolean
+ registerService()
uDiscovery
- gatewayData: Gateway
- gateways: [ ] Gateway
- pheromones: <Domain,<Gateway,Double>>
- costs: <Gateway, Double>
+ uDiscovery()
+ listener()
+ sendMessage()
InitialisationManager
+ recognisePlaces(): [ ] Place
+ getCityServices(): [ ] CityService
+ getDomains(): [ ] Domain
+ calculateRelevance(): <Domain,Double>
+ addGateway()
Manager
+ getDestinationsInterests(): [ ] Gateway
+ getDestinationsPheromones(): [ ] Gateway
+ updatePheromones()
+ updateCosts()
AdaptiveManager
+ calculateUtility(): Double
+ defineRequiredDomains(): [ ] Domain
+ getServices(): [ ] Service
+ insert([ ] Service)
+ remove([ ] Service)
DQN
+ initMDP(): Obs
+ storeAndReplay(): Obs
+ getActionToPerform(): Action
PeriodicEventManager
+ calculateReward(): Double
UnforeseenEventManager
ScheduledEventManager
+ getScheduledEvents(): [ ] CityEvent
CityMDP
+ getState()
+ lastObservation()

Figure 4.5: *uDiscovery* - Classes Diagram.

model has 9 super classes which group 40 individual to describe different domains in a city. Services and requests parameters are described using the ontologies defined by the OWLS-TC V4 dataset[3]

4.1.2 uDiscovery Class Diagram

Figure 4.5 illustrates the class diagram of the implementation. It is composed of the following classes:

[3]OWLS-TC V4 Dataset - `http://projects.semwebcentral.org/projects/owls-tc/`

- **uDiscovery:** This class is the communication interface of each gateway in the network. It stores data about the local gateway, other gateways in the network, and the links with these gateways (i.e., pheromones and costs). This class listens for messages from providers, consumers and other gateways, and sends messages to other entities in the network according to *uDiscovery* managers. It redirects gateways advertisements to the initialisation manager, registration messages to the service manager, discovery messages, to the request manager, and adaptation messages to the adaptive manager.

- **Manager:** This is an abstract class that groups *uDiscovery* managers. Each manager can get gateways destinations by interest or pheromones, and update pheromones and costs.

- **Initialisation Manager:** This class is responsible for the network initialisation and maintenance processes (Section 3.4.1.2). It has methods to recognise surrounding places, define city services and domains, calculate the gateway relevance for a domain, and add gateways to the list of known gateways. This manager sends advertisements messages with gateways information through the *uDiscovery* class.

- **Service Manager:** This class is responsible for the organisation of services (Section 3.4.1.3). It has methods to define shared domains between two lists of domains, and register services in the local repository. This manager sends advertisements messages with services information through the *uDiscovery* class.

- **Request Manager:** This class is responsible for the resolution of requests (Section 3.4.1.4). It uses the service planner to solve requests and has methods to get a gateway that has participated in the resolution of a request, to determine if a request was solved, and to update the feedback. This request manager sends responses to consumers, or requests to other gateways through the *uDiscovery* class.

- **Adaptive Manager:** This is an abstract class that groups the managers responsible for adaptation processes (Section 3.4.2). Each adaptive manager has methods to calculate the utility value, define required domains, get services from local repository, register services in the repository and remove services. Adaptive managers receive and send adaptation messages through the *uDiscovery* class.

- **Unforeseen Event Manager:** This class responds to unforeseen events (Section 3.4.2.1). It calculates the utility, defines required domains, sends and receives messages, and updates the local repository accordingly.

- **Scheduled Event Manager:** This class responds to scheduled events (Section 3.4.2.2). It has a method to periodically pull scheduled events and act accordingly.
- **Periodic Event Manager:** This class responds to periodic events (Section 3.4.2.3). It has a method to calculate system rewards. It uses the DQN class to get city states information and actions to perform according to historical learning from city behaviour.
- **DQN:** This class implements the DQN algorithm, which is used from the Deep Learning from Java (DL4J)[4] library. It has a method to initialise the city MDP when a city period is starting.The DQN algorithm is splited in two methods to integrate discovery rewards and city status. One method stores experiences and performs replay to learn from them, and the another one returns the action to be performed according to a city observation.
- **City MDP:** This class represents the city Markov decision process. It has methods to get current city state, and last city observation.
- **Service Planner:** This class implements the heuristic planning algorithm (Section 3.4.3). It has methods to solve requests and create, update and rank service plans. It uses the service matchmaker to compare services and requests parameters, and the feedback manager to validate these comparisons' outputs.
- **Service Matchmaker:** This class implements the methods to compare parameters. It has methods to perform semantic and syntactic comparisons.
- **Feedback Manager:** This class manages consumers' feedback. It implements methods to save discovered plans, retrieve feedback and update feedback in the defined data model.

4.2 Urban-based Service Management

This section presents the interaction between *uDiscovery* components, data entities, and classes to initialise the network of gateways, organise services and solve requests in the *uDiscovery* implementation

4.2.1 Initialisation Management

Figure 4.6 illustrates the behaviour that *uDiscovery* follows to initialise the network of gateways (Section 3.4.1.2). The initialisation manager first retrieves urban context from the

[4]DL4J - `https://deeplearning4j.org/`

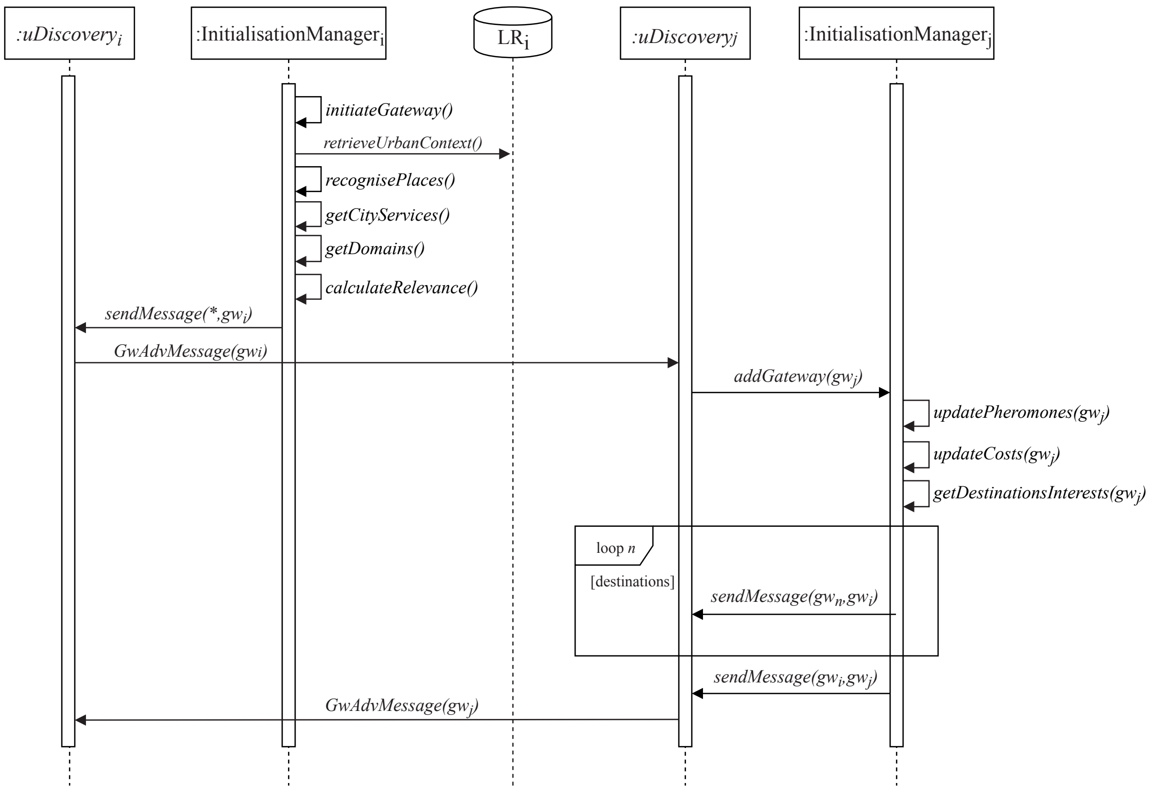


Figure 4.6: *uDiscovery* - Initialisation Sequence Diagram.

repository to define the surrounding places, the offered city services, and the gateway domains. Then, it calculates the relevance of the gateway for each domain, and advertises the gateway information to other gateways in the network through the *uDiscovery* interface.

The *uDiscovery* interface asks the initialisation manager to add a gateway, once it receives a gateway advertisement message. The initialisation manager stores the gateway information and update the pheromones and cost values. Then, it defines relevant destinations (i.e., other gateways in the network) for the gateway information. It sends a gateway advertisement message to each candidate through the *uDiscovery* interface.

4.2.2 Service Organisation

Figure 4.7 illustrates the behaviour when a registration message arrives to *uDiscovery* (Section 3.4.1.3). The interface asks the service manger to register the service description. The service manager determines whether the service should be stored in the local repository by checking if the service and the gateway share domains. If they do, the service manager inserts the service in the local repository, and replies with a success message to the registration message sender through the *uDiscovery* interface.

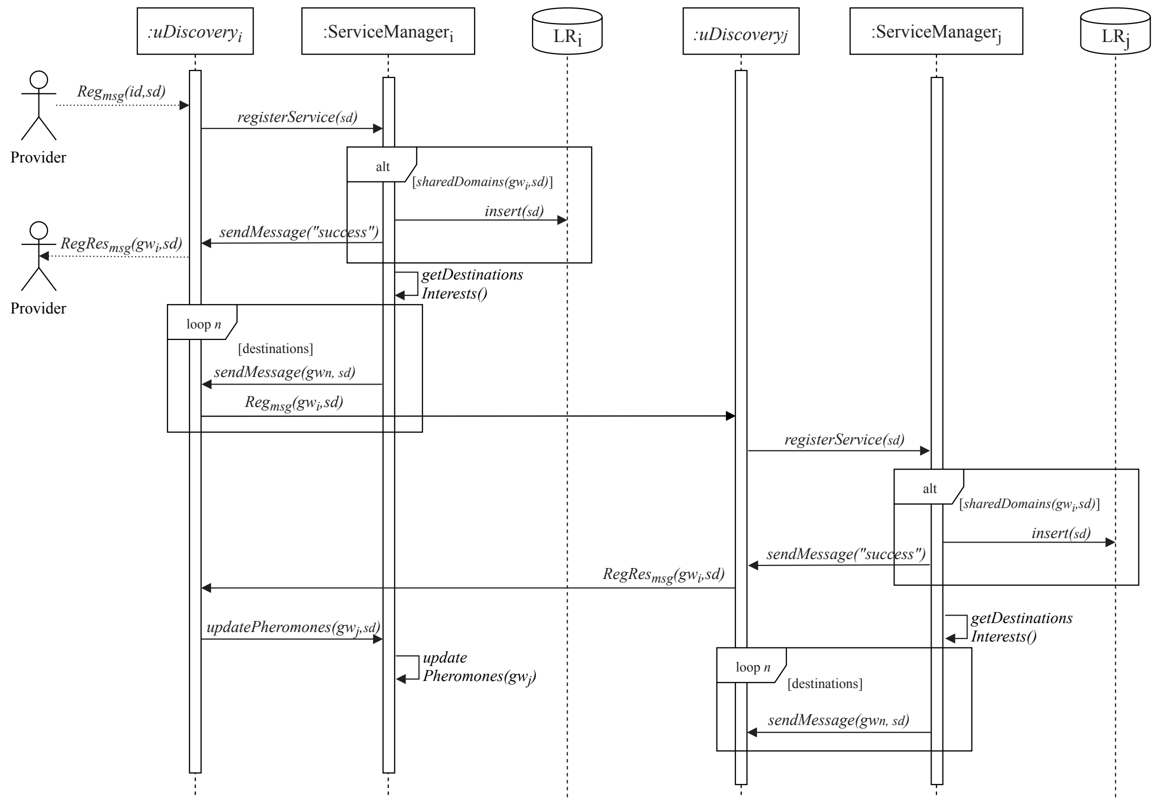


Figure 4.7: *uDiscovery* - Registration Sequence Diagram.

The interface asks the service manager to update the pheromones values when it receives a registration response message.

4.2.3 Requests Resolution

Figure 4.8 illustrates the behaviour when a discovery message arrives (Section 3.4.1.4). The interface asks the request manager to solve the received request. The request manager calls the service planner, which creates the initial plans for the request and retrieves the relevant search spaces from the local repository. The planner iterates over this search space and uses the matchmaker to compare available services with the current plans. The matchmaker applies the semantic and syntactic methods to find edges and vertices for the plan. The planner validates the comparisons' outputs with the feedback information by calling the feedback manager. Then, the planner updates the list of plans according to this validation, and compares the updated plans with the inputs to define if the request is solved. This comparison also uses the syntactic and semantic methods from the matchmaker. The planner ranks the plans according to the functional matching index and returns the list to the request manager, once the search finishes. The request manager receives the plans and checks if the request was solved. If it was, the request manager sends the response through the *uDiscovery*

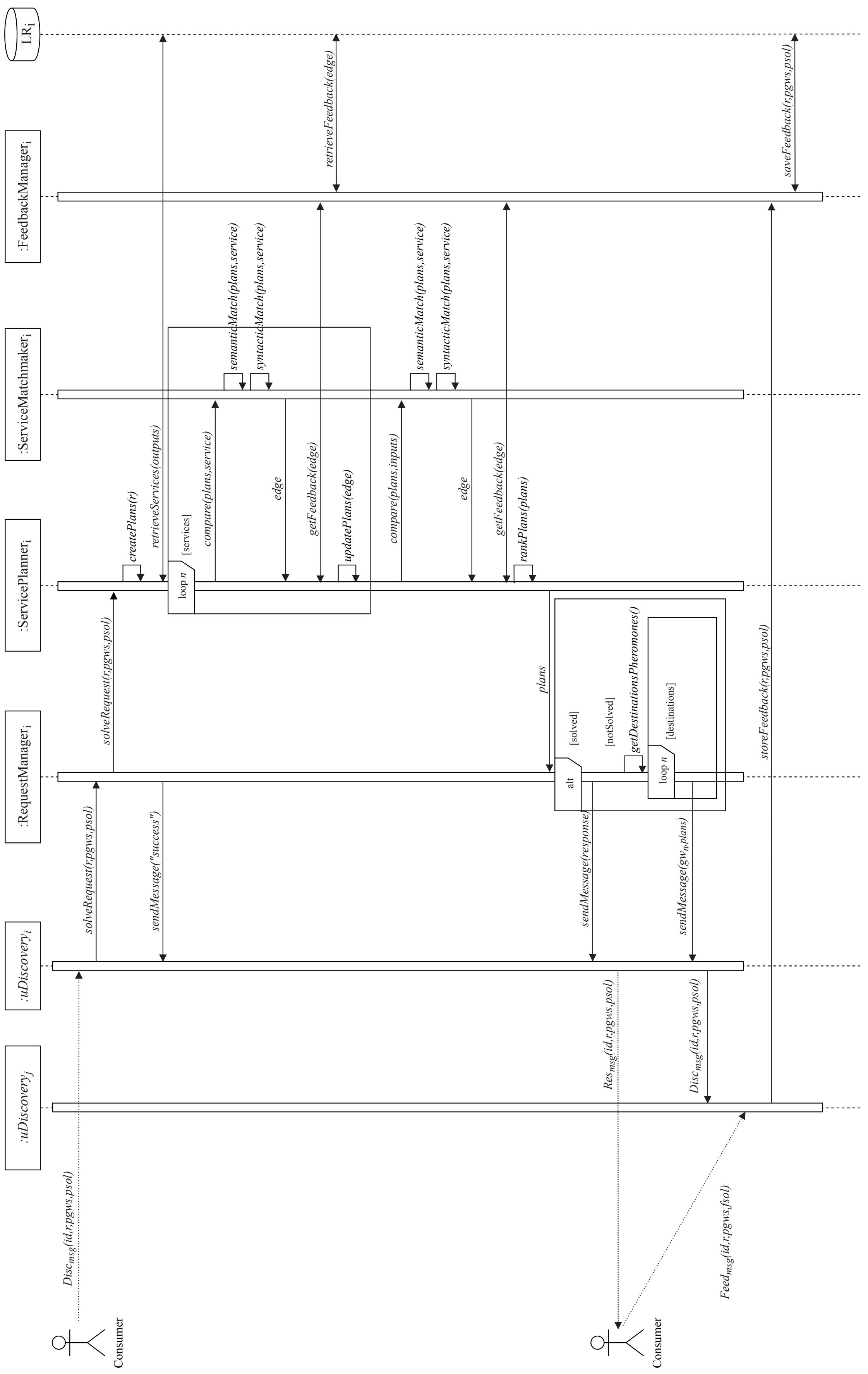

:uDiscovery$_j$
:uDiscovery$_i$
:RequestManager$_i$
:ServicePlanner$_i$
:ServiceMatchmaker$_i$
:FeedbackManager$_i$
LR$_i$
Consumer
Consumer
Disc$_{msg}$(id,r,pgws,psol)
solveRequest(r,pgws,psol)
solveRequest(r,pgws,psol)
createPlans(r)
retrieveServices(outputs)
sendMessage("success")
loop n
[services]
compare(plans,service)
semanticMatch(plans,service)
syntacticMatch(plans,service)
edge
getFeedback(edge)
retrieveFeedback(edge)
updatePlans(edge)
compare(plans,inputs)
semanticMatch(plans,service)
syntacticMatch(plans,service)
edge
getFeedback(edge)
rankPlans(plans)
plans
alt
[solved]
[notSolved]
getDestinationsPheromones()
loop n
[destinations]
sendMessage(response)
sendMessage(gw$_n$,plans)
Res$_{msg}$(id,r,pgws,psol)
Disc$_{msg}$(id,r,pgws,psol)
Feed$_{msg}$(id,r,pgws,fsol)
storeFeedback(r,pgws,psol)
saveFeedback(r,pgws,psol)

Figure 4.8: *uDiscovery* - Discovery Sequence Diagram.

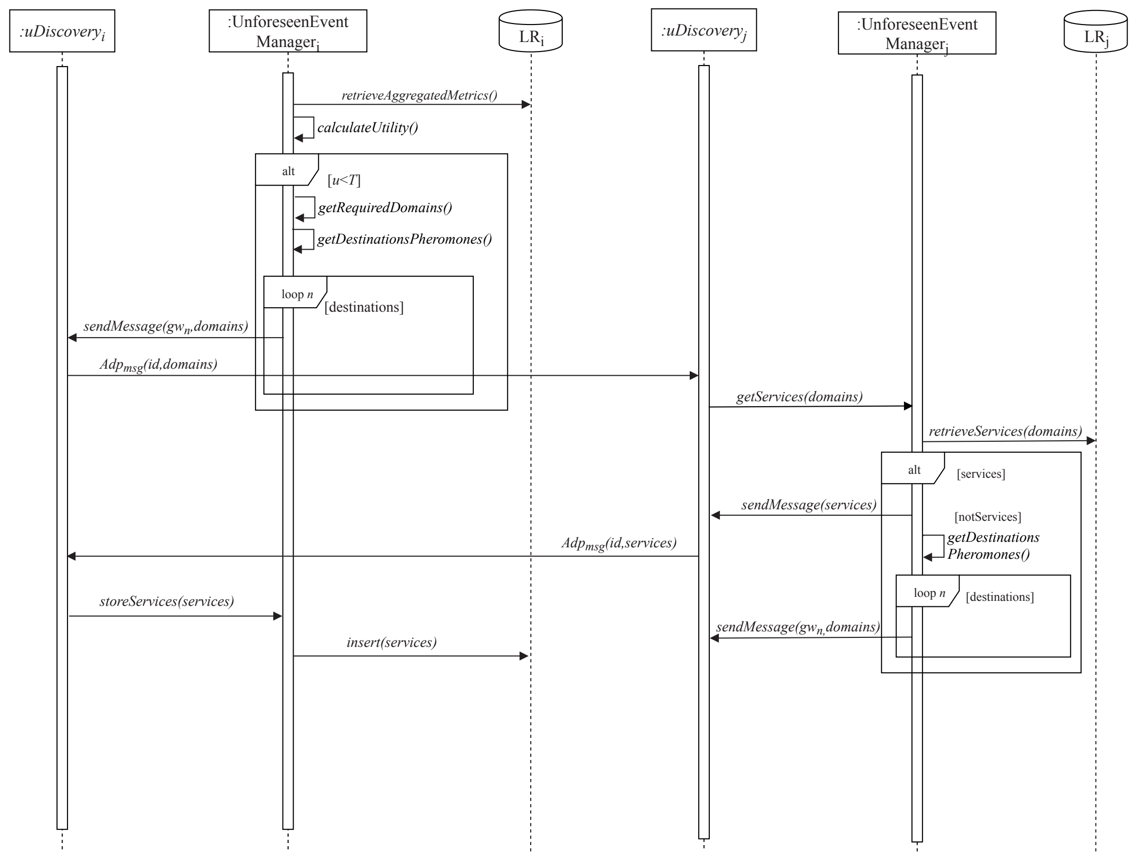


Figure 4.9: *uDiscovery* - Unforeseen Events Management Sequence Diagram.

interface. Otherwise, the manager gets the list of gateways where the request is most likely to be solved using the pheromones information. The manager sends a request message to each candidate through the interface. Consumers can submit their feedback for the discovered services as a boolean mark through the *uDiscovery* interface. This information is stored in the local repository by the feedback manager to be used in future discovery processes by the service planner.

4.3 Self-adaptive Service Management

This section presents the interaction between *uDiscovery* components, data entities, and classes to adapt the organisation of services according to city events.

4.3.1 Unforeseen Events Adaptation

Figure 4.9 presents the behaviour of *uDiscovery* in the presence of unforeseen events based on the Section 3.4.2.1. The unforeseen event manager monitors the metrics in the local repository and computes the utility function. If the utility value is lower than the threshold

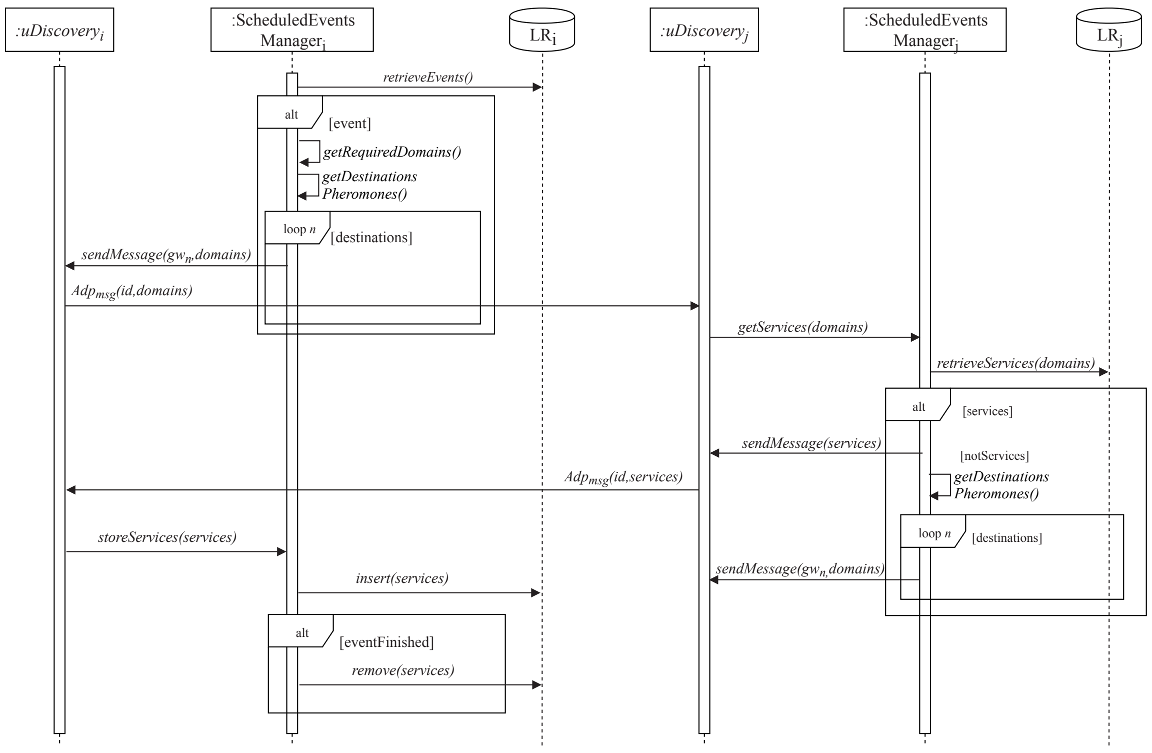


Figure 4.10: *uDiscovery* - Scheduled Events Management Sequence Diagram.

T, the manager defines the required domains, and the gateways that are most likely to have the required service information based on the pheromones information. The manager sends an adaptive message asking for services through the *uDiscovery* interface.

The interface asks for services from the unforeseen event manager, when it gets an adaptive message asking for services from a set of domains. The manager retrieves relevant services from the local repository relating to these domains and responds to the adaptive message through the *uDiscovery* interface, if services are found. Otherwise, it forwards the adaptive message to the most promising gateways according to the pheromones information. *uDiscovery* stores services in the local registry, once an adaptive message with services arrives from other gateways in the network.

4.3.2 Scheduled Events Adaptation

Figure 4.10 illustrates the behaviour of *uDiscovery* when the city has scheduled events. The scheduled events manager monitors the scheduled events (Section 3.4.2.2). If there are events starting soon, the manager defines the required domains according to these events, and sends adaptive messages to the most promising gateways in the network according to the pheromones information. The system follows the same sequence as described in the Figure 4.9 once an message asking for services arrives.

Figure 4.10 also shows the manager's behaviour when services arrive as a response to an adaptive message. The *uDiscovery* interface asks the manager to register the services.

4.3.3 Periodic Events Adaptation

Figure 4.11 presents the behaviour of *uDiscovery* to recognise city patterns and proactively update the organisation of services according to Section 3.4.2.3. The periodic events manager determines the time period and initialises the city MDP, if the period is starting. Otherwise, the manager calculates the current system reward, retrieves the city state, stores the new experience, and learns from historical experiences. Then, the manager gets the action to perform with the current city observation. If the action is to ask for services (i.e., A_1), the manager defines the required domains and the most promising gateways to ask for services according to the pheromones information. The manager sends a request for services to each destination through the *uDiscovery* interface. The manager removes services from the local repository, if the action to perform is A_2. *uDiscovery* stores services in the local registry, once an adaptive message with services arrives from other gateways as in previous adaptive processes.

4.4 Implementation Summary

This chapter presents the implementation details of *uDiscovery*. It starts describing the system architecture where three main components are highlighted. The urban based service manager that initialises the network of gateways, organises service descriptions and receives consumers' requests. The self-adaptive service manager adapts the service organisation according to city events. The heuristic service planner performs the service search in the local repository. These components rely on a data model that represents different system entities, and their attributes. A class diagram describes how *uDiscovery* components interact and uses the data to offer its main functionalities. This interaction is then detailed by sequence diagrams that show the behaviour that the system follows to provide functionalities to initialise gateways, register services, solve requests, and adapt to city events.

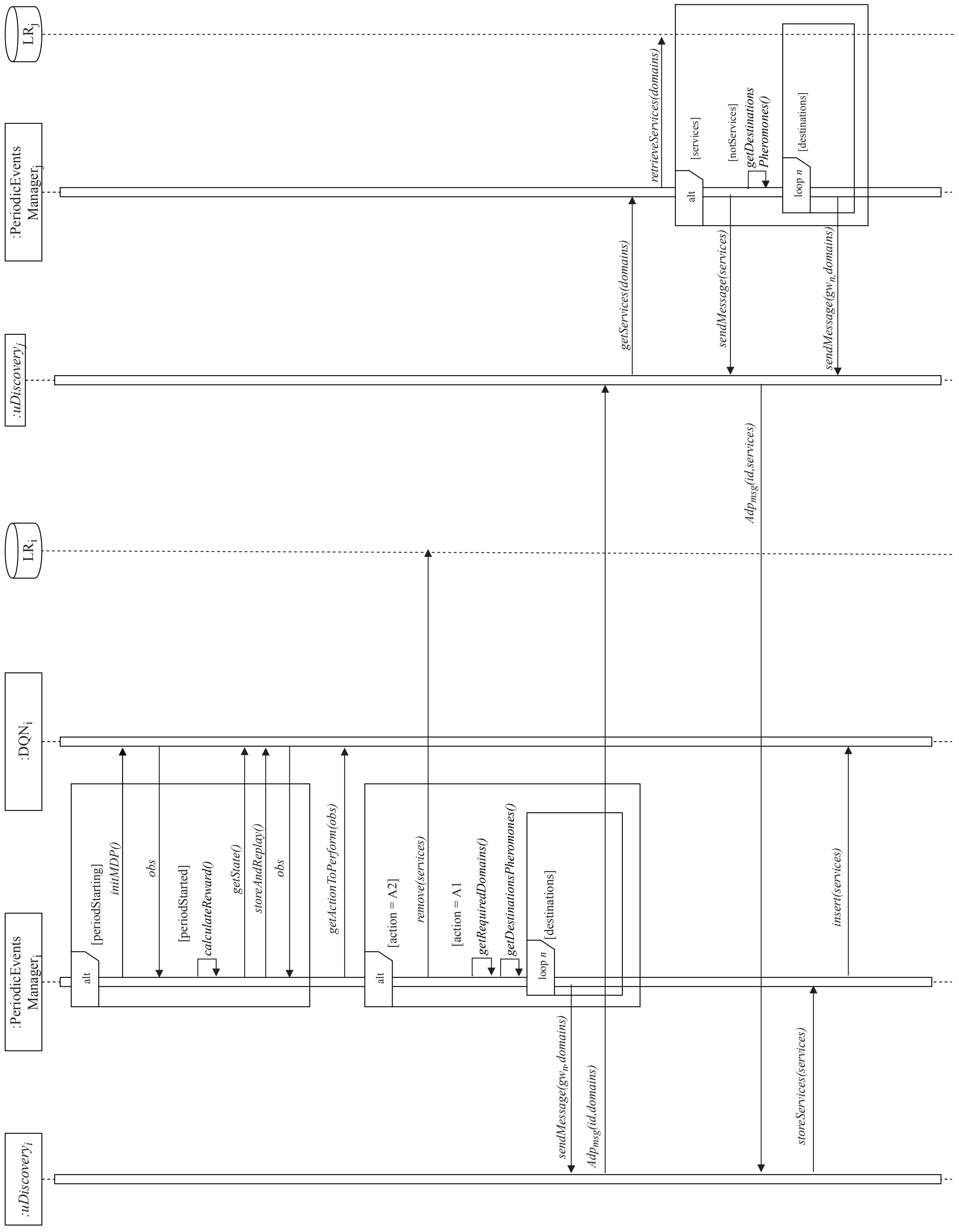


Figure 4.11: *uDiscovery* - Periodic Events Management Sequence Diagram.

Chapter 5

Evaluation

Previous chapters introduced *uDiscovery* design and implementation details. This chapter evaluates to what extent *uDiscovery* improves the efficiency of service discovery in large and dynamic smart city environments. This chapter is organised as follows: Section 5.1 maps evaluation objectives to the research questions, introduces the studies that achieve these objectives, and defines the data used in these studies. Section 5.1.4 presents the statistical tests carried on the results to define parameters values and determine to what extent differences in the results are statistically significant. Section 5.2 presents the simulation studies that achieve evaluation objectives 1 and 2, and Section 5.3 presents the study that achieves the objective 3. Finally, Section 5.4 summarises this chapter.

5.1 Evaluation Approach

This chapter presents the evaluation of $uDiscovery's$ performance, which measures to what extent can the use of urban context to organise services' information improve service discovery efficiency in the presence of a large number of services in smart cities environments **(RQ. 1)**, to what extent can the use of urban context to adapt the services' organisation maintain or improve service discovery efficiency *over time* in dynamic smart cities environments **(RQ. 2)**, and to what extent can the use of consumers' feedback to search for services improve service discovery efficiency, and minimise human input when responding to complex consumer's requirements **(RQ. 3)**. This evaluation performed two types of studies to address the research questions (Table 5.1). These studies are:

5.1.1 Simulation-based Evaluation on Service Discovery Efficiency

Simulation studies rely on the simplification of real-world scenarios by manipulating a set of environment variables to assess an algorithm's performance in different scenarios or under different system configurations [Wohlin et al., 2012]. This evaluation measures $uDiscovery's$ performance through four simulation studies, as follows:

1. **General Service Discovery Efficiency Study:** This study evaluates to what extent *uDiscovery* improves the service discovery efficiency in the presence of a large number of services in smart cities environments, by organising services' information using urban context (**RQ 1**). This study simulates a city where a variable number of gateways manages a variable number of services and receives requests from a simulated consumer that moves through the city, without the presence of any event. The network can be static (i.e., all gateways are fixed), semi-mobile (i.e., half of the gateways are static and half mobile), or fully-mobile (i.e., all gateways are mobile).

2. **Unforeseen Events Study:** This study evaluates to what extent *uDiscovery* improves or maintains the service discovery efficiency *over time*, in the presence of unforeseen events, by adapting the services' organisation based on urban context (**RQ 2**). This study simulates a city environment where an unforeseen event happens in the city and influences consumer's requests. *uDiscovery* is evaluated under a variable number of services in different mobility scenarios (i.e., static network, semi-mobile network, and fully-mobile network).

3. **Scheduled Events Study:** This study evaluates to what extent *uDiscovery* improves or maintains the service discovery efficiency *over time*, in the presence of scheduled events (**RQ 2**). This study simulates a city environment where scheduled events are predefined and influence consumer's requests. *uDiscovery* is evaluated under a variable number of services in different mobility scenarios (i.e., static network, semi-mobile network, and fully-mobile network).

Table 5.1: Mapping of Research Questions to Evaluation Studies.

Research Question	Proposed Study	Performance Metrics	Chapter Section
RQ1: To what extent can the use of urban context to organise services' information improve service discovery efficiency in the presence of a large number of services in smart cities environments?	1. General Service Discovery Efficiency Study	a) Service Discovery Utility b) Rate of Solved Requests c) Search Precision d) Response Time e) Number of Hops f) Exchanged Messages	Section 5.2
RQ2: To what extent can the use of urban context to adapt the services' organisation maintain or improve service discovery efficiency over time in dynamic smart cities environments?	2. Unforeseen Events Study 3. Scheduled Events Study 4. Periodic Events Study	a) Service Discovery Utility b) Rate of Solved Requests c) Search Precision d) Response Time e) Number of Hops f) Exchanged Messages	Section 5.2
RQ3: To what extent can the use of consumers' feedback to search for services improve the service discovery efficiency, and minimise human input when responding to complex consumer's requirements?	5. Prototype Study	a) Search Precision b) Response Time c) Human input	Section 5.3

4. **Periodic Events Support Study:** This study evaluates to what extent *uDiscovery* improves or maintains the service discovery efficiency *over time*, in the presence of periodic events (**RQ 2**). This study simulates a city environment where periodic events are happening and influence consumer's requests. It evaluates *uDiscovery* under a variable number of services in different mobility scenarios (i.e., static network, semi-mobile network, and fully-mobile network).

Evaluation Metrics

The simulation studies use the following metrics to evaluate $uDiscovery's$ performance:

- **Service Discovery Utility:** This metric measures the utility value of the discovery process over time according to the eq 3.10 in Section 3.4.2 of Chapter 3.
- **Rate of Solved Requests:** This metric measures the relation between the number of solved requests and the number of received requests. It reflects the success rate of the discovery approach and varies from 0 to 1, where higher is better.
- **Search Precision:** This metric represents the accuracy of the discovery approach and varies from 0 to 1, the higher the better. It measures the proportion of relevant services returned for a given request and follows the eq 5.1 [Manning et al., 2010].

$$P = \frac{tP}{tP + fP} \tag{5.1}$$

  where, tP is the number of retrieved services that are relevant for the request, and fP is the number of retrieved services that are not relevant.
- **Response Time:** This measures the time that the approach takes to solve a request from when the consumer sends the request message to when the response is delivered back. Lower is better.
- **Number of Hops:** This metric measures the number of hops that the discovery approach needs to solve a request in the distributed network of gateways. This metric, together with the exchanged messages, represents the network usage in a simulation, and lower is better.
- **Exchanged Messages:** This metric measures the number of messages that are exchanged between the network of gateways, representing network usage in the simulations, where lower is better.

5.1.2 Prototype-based Evaluation

The IoT test bed prototype measures to what extent *uDiscovery* improves service discovery efficiency, and minimises human input when responding to complex consumer's requirements **(RQ 3)**. The IoT test bed is composed of a set of Raspberry Pi boards, where each board acts as an IoT gateway that manages services, receives requests from a consumer board, and forwards requests to other gateways. Gateways solve requests using the planning algorithm introduced in Section 3.4.3.

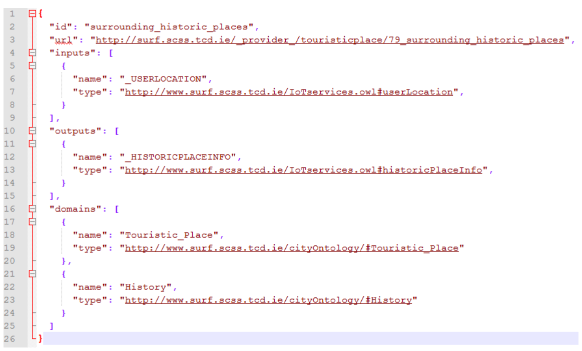
Evaluation Metrics

- **Search Precision:** This is the accuracy of the discovery approach and varies from 0 to 1, as in the simulation studies. It is also defined by equation 5.1.
- **Response Time:** This measures the time that the approach takes to solve a request from when the consumer board sends the request message to when the response is delivered.
- **Human Input:** This is the amount of information that consumers must provide to trigger a discovery process, measured in number of bytes that consumers must input as request and feedback information.

5.1.3 Data Set Definition

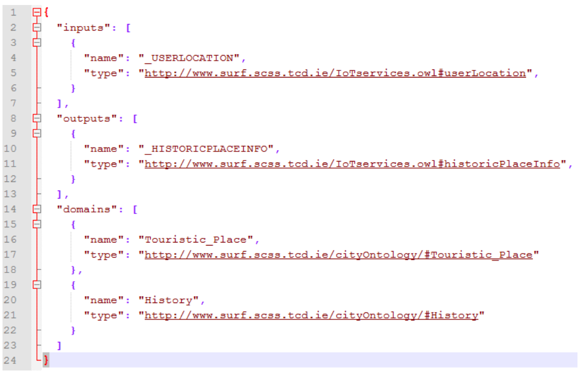
A data set of IoT services descriptions, requests and responses is defined for this work because of the absence of an existing data set in the literature, which represents current and future IoT services offered by cities. This data set has 136 atomic services based on IoT services examples and domains proposed by IoT literature [Al-Fuqaha et al., 2015], and 946 services which are translated from the OWLS-TC V4. These services are used to define mock up services that are used to create environments with a large number of services. Figure 5.1a shows a service description example. This service provides information about historical city places based on user's location. Simulation and prototype studies use this data set in each experiment.

Request and responses were created following the approach proposed by Uribieta et al [Urbieta et al., 2017], which generates requests and responses by running the service discovery process off-line. A planning algorithm [Chen et al., 2016] is executed off-line over the 136 IoT services to discover service plans and identify I/O relations between services (e.g., the output of the service s_1 is the input of the service s_2). A request is created for each plan that the algorithm



```
{
  "id": "surrounding_historic_places",
  "url": "http://surf.scss.tcd.ie/_provider_/touristicplace/79_surrounding_historic_places",
  "inputs": [
    {
      "name": "_USERLOCATION",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#userLocation",
    }
  ],
  "outputs": [
    {
      "name": "_HISTORICPLACEINFO",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#historicPlaceInfo",
    }
  ],
  "domains": [
    {
      "name": "Touristic_Place",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#Touristic_Place"
    },
    {
      "name": "History",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#History"
    }
  ]
}
```

(a) Service Description.



```
{
  "inputs": [
    {
      "name": "_USERLOCATION",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#userLocation",
    }
  ],
  "outputs": [
    {
      "name": "_HISTORICPLACEINFO",
      "type": "http://www.surf.scss.tcd.ie/IoTservices.owl#historicPlaceInfo",
    }
  ],
  "domains": [
    {
      "name": "Touristic_Place",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#Touristic_Place"
    },
    {
      "name": "History",
      "type": "http://www.surf.scss.tcd.ie/cityOntology/#History"
    }
  ]
}
```

(b) Request Description.

Figure 5.1: Data Formats.

discovers, if all services in the plan share at least one domain. Request inputs are the inputs of the first service in the plan, and request outputs are the outputs of the last service in the plan. Atomic services are managed as plans of one service. Figure 5.1b shows a request example. This request asks for a service that provide information about historical places, and receives a location as input. The data set also stores responses for each valid request to measure search accuracy. A service response in an experiment is compared with the predefined responses to determine its relevance. Table 5.2 summarises the number of requests and responses in the data set according to the length of the services plans, which can include from 1 to 5 services. The data set has a total of 363 service requests with 376 relevant responses.

Table 5.2: Requests Data Set Size.

Length of Service Plans	**Number of Requests**	**Relevant Plans**
1	136	136
2	87	91
3	61	62
4	49	53
5	30	34
TOTAL	**363**	**376**

5.1.4 Statistical Analysis

This evaluation applies a Kruskal-Wallis test together with a multiple comparison of ranked means on the evaluation results [McKight and Najab, 2010]. This is a non-parametric test that assesses the differences between three or more independently-sampled groups on a single, non-normally distributed continuous variable. The null hypothesis specifies that these groups are subsets of the same population (i.e., $H0 : (a, b, c, ..., n) \subseteq p$). This test combines the groups into a single group and ranks the variable of interest. These scores, along with the group sizes, are used to calculate the H statistic (eq 5.2)

$$H = N - 1(\frac{gn_n(t_i - T_i)^2}{gsn_n(t_j - T_i)^2}) \tag{5.2}$$

where, n_n is the sample size of the corresponding group, g is the sum of the group n, sn_n is the sum of the corresponding group n, t_i is the average observed rank sums for the group, t_j is the observed rank for an observation for the corresponding group, and T_i is the observed total average rank sums.

This test has three assumptions: the interest variable should be continuous, the independent variable should consist of two or more categorical independent groups, and the observations are independent. The test is used to define the impact of the models' parameters in their performance, and to determine if the differences between evaluated approaches are statistically significant. MATLAB is used to calculate the $\rho - value$ based on the eq 5.2, which must be under 0.01 (i.e., confidence level of 99%) to reject the null hypothesis. The test is applied on the utility values for the simulation studies, and on the response time, search precision, and input size on the prototype-based study.

5.2 Simulation-based Evaluation on Service Discovery Efficiency

This section presents the simulation studies that evaluate $uDiscovery's$ efficiency against baseline approaches in different simulated city environments. It describes the experimental set-up, introduces the evaluated approaches, and discusses the results.

5.2.1 Experimental Set-up

Simonstrator is used to simulate smart city environments [Richerzhagen et al., 2015]. This is a P2P Java-based simulator for mobile and large applications. The experiments were run on the Kelvin system, a high performance compute cluster hosted and managed at the Trinity Centre for High Performance Computing (TCHPC). Each node in the cluster has a Linux OS, 12 2.66GHz Intel processors, and 24GB of RAM[1].

A network of gateways that cover Dublin city centre is simulated (i.e., $2Km^2$ approx.), with static, semi-mobile, and fully-mobile scenarios to determine how mobility affects approaches performance. All gateways are fixed in the static scenario, 50% of gateways are static and

[1]Kelvin Details - `https://www.tchpc.tcd.ie/resources/clusters/kelvin`

50% are mobile in the semi-mobile scenario. All gateways are mobile in the fully-mobile scenario. Static gateways are distributed in a grid topology and mobile gateways follow a social movement pattern provided by Simonstrator, with a speed that varies from $2.7m/s$ (i.e., $10Km/h$ approx) to $13.8m/s$ (i.e., $50Km/h$ approx). The number of services varies from 20 thousand to 100 thousand, increased by 20 thousand in each experiment. A service provider per gateway is simulated to register services. Each provider selects a random description from the services data set (Section 5.1.3), and sends a registration message with the selected description. Simulated providers repeat this process up to the number of registered services in the network reach the desired number.

5.2.2 Baseline Approaches

The simulation studies compare four approaches, two baselines and two $uDiscovery's$ versions, which use the algorithm defined in the Section 3.4.3 to search for services.

- $Location - based\ Approach$: This baseline groups and forwards requests according to services' coordinates. It was implemented following the ideas proposed by Wang et al. [Wang et al., 2015], and Fredj et al. [Fredj et al., 2014].
- $Domain - based\ Approach$: This baseline groups and forwards requests according to services' domains. It was implemented following the ideas proposed by Paganelli et al. [Paganelli and Parlanti, 2012]
- $uDiscovery1$: This version of *uDiscovery* groups services and forwards requests according to the model defined in the Section 3.4.1. But, it does not have the adaptive properties defined in the Section 3.4.2. This variation is used to determine the effect of the adaptive model in *uDiscovery* performance.
- $uDiscovery2$: This approach is the full version of *uDiscovery*. It includes the grouping of services and requests forwarding based on urban context (Section 3.4.1), and the self-adaptive mechanisms described in Section 3.4.2.

Each approach has parameters that influence their performance. A statistical test determines which are the best set of parameters for each approach in the Appendix A. The distance to register services is $100m$ and the limit of hops is 5 for the location-based approach, the number of domains is 5 and the limit of hops is 3 for the domain-based, and the distance to recognise places is $100m$ and the limit of hops is 5 for $uDiscovery's$ versions. This section compares these approaches under these parameters.

Table 5.3: Experiments Parameters for General Service Discovery Efficiency Study.

Length of time for service discovery	8 hours
# of consumer requests	100
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	100; 300; 500
Mobility scenarios	Static; semi-mobile; fully-mobile
Scenarios	1. Services X Mobility 2. Gateways X Mobility
Replication	10 rounds each experiment
Hops limit (See Appendix A)	- 5 for location-based and *uDiscovery* - 3 for domain-based
Distance (See Appendix A)	- 100 for location-based and *uDiscovery*
Number of Domains (See Appendix A)	- 5 for domain-based

5.2.3 General Service Discovery Efficiency Study

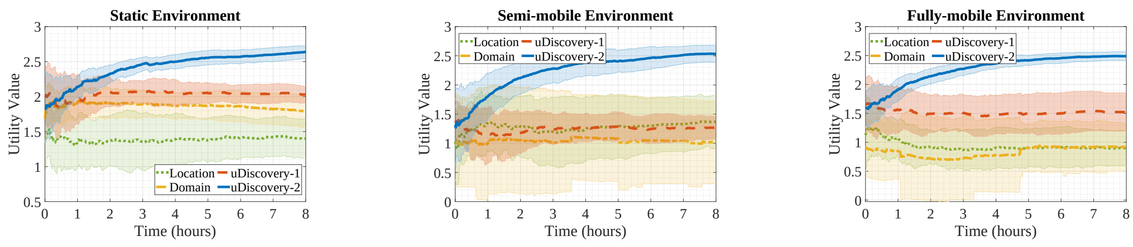
Table 5.3 presents the parameters of the experiments that evaluate and compare the general service discovery efficiency of each approach (i.e., without events in the environment). Each experiment simulates a consumer that performs 100 sequential service requests to a gateway in the network to evaluate the performance with different types of requests. The simulated consumer selects random requests from the data set (Section 5.1.3) according to its location to simulate the influence of city places in consumer's interests. The consumer needs around 8 simulated hours to perform all requests. The experiment's parameters are the number of services, number of gateways, and mobility scenarios. There are two experimental scenarios that combines parameters to illustrate their influence in the discovery efficiency. The first experimental scenario illustrates a variable number of services, in different mobility scenarios, with 500 gateways. The second experimental scenario illustrates a variable number of gateways, in different mobility scenarios, with 100 thousand services. Experiments are replicated 10 times for each experiment. The particular parameters for each approach are selected according to the Appendix A

5.2.3.1 General Service Discovery Efficiency Study Results

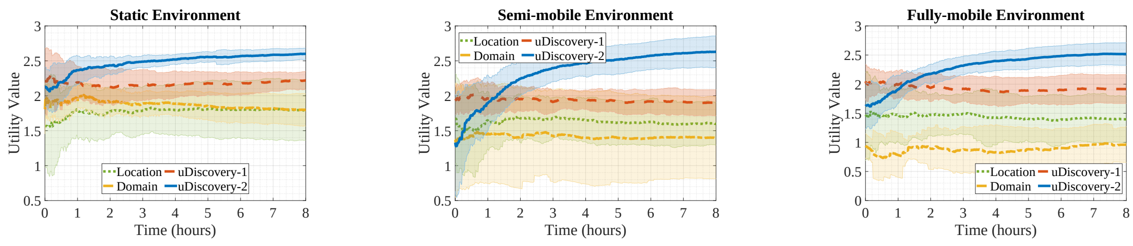
Scenario 1: varying number of services with mobility scenarios

Figure 5.2 shows the utility curve for the experimental scenario 1. *uDiscovery*2 has a better utility than baselines, which is between 2.48, in the worst case, and 2.93, in the best case at hour 8. *uDiscovery*1 is the second best with utility values between 1.26 and 2.36 at hour 8. The utility value varies between 0.9 and 2.25 in the location-based approach, and between 1.02 and 1.9 in the domain-based approach at hour 8. *uDiscovery*2 outperforms the baselines in all cases. It starts with a similar performance to *uDiscovery*1 because it uses the same information to initially group services. However, its utility is improved over time because the self-adaptive model moves services according to Algorithm 5, continuously updating the utility threshold. This threshold is updated when there are no events in the city because even when requests are solved using the semantic overlay, other metrics such as response time or number of hops can be improved by moving services locally. This improvement is more visible when the number of services is low (Figures 5.2a, 5.2b and 5.2c) and gateways are semi or fully-mobile because both number of services and mobility affects the discovery performance of the initial distribution of services. A low number of services implies that it is less likely that a service is registered where it is requested. Mobility implies intermittent gateways which affects discovery because registries are not always available. The number of services and mobility affect all the baselines' performance, but *uDiscovery*2 ends with the best utility value in all cases. The semantic overlay provides enough information to ask for services from the right gateways, despite number of services. The maintenance of such an overlay includes updating each gateway's local routing table, which allows mobile gateways to be considered as services carriers, taking advantage of their movement through the city.

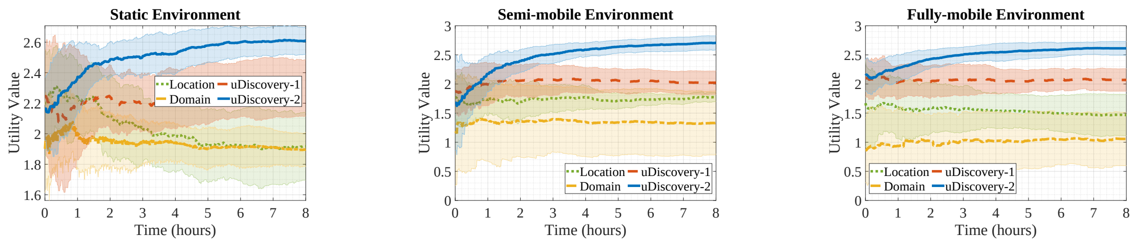
As described in Section 3.4.2, the utility function encapsulates the rate of solved requests, search precision, discovery response time, and number of hops. Figure 5.3 shows these metrics for the experimental scenario 1. Each graph shows the median value from the 10 rounds of each experiment with their quartiles 1 and 3. Figure 5.3a shows the **rate of solved requests**. *uDiscovery*2 maintain a rate of solved requests close to 1 (i.e., it solves almost every received request) despite the number of services and the mobility scenario. *uDiscovery*1 has a close performance to *uDiscovery*2 in the static environment, which shows the advantage of using urban-context to drive service discovery. But its rate of solved requests decays to a minimum value of 0.69 with 20 thousand services in the fully-mobile scenario. *uDiscovery*2 addresses the mobility negative impact with the properties of the self-adaptive approach (Section 3.4.2). The location-based approach has an acceptable performance in the static scenario, and varies from 0.8, with 20 thousand services, to 0.96 with 100 thousand services. However, the location-based approach is affected by gateways' mobility. The rate of solved requests varies from around 0.5, with 20 thousand services, to 0.8 with 100 thousand services for both semi-mobile and fully-mobile environments. The domain-based approach presents the worst



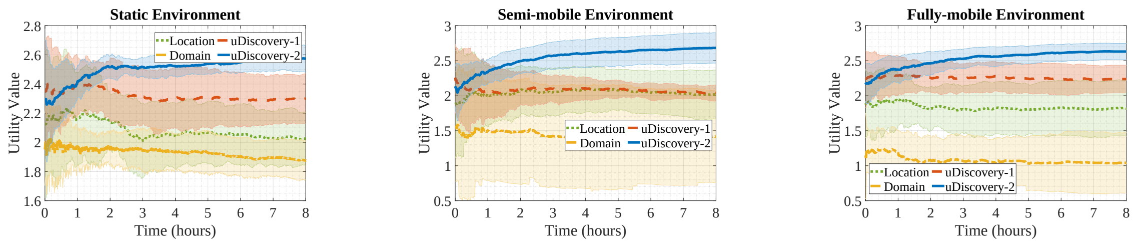
(a) 20,000 Services



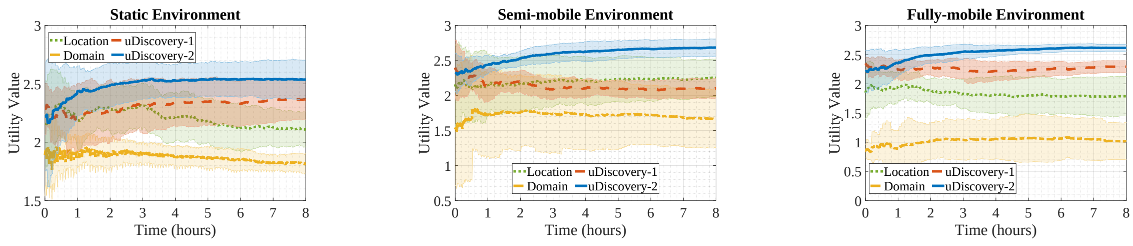
(b) 40,000 Services



(c) 60,000 Services

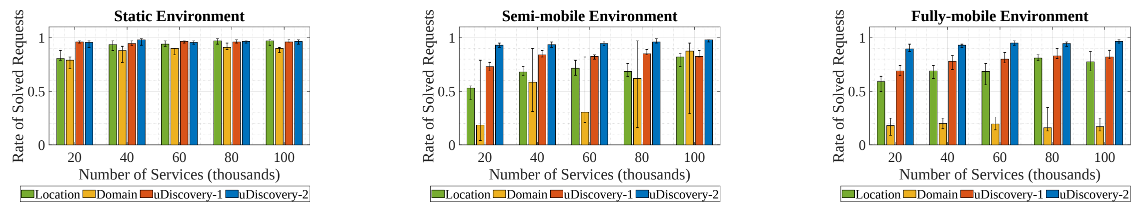


(d) 80,000 Services

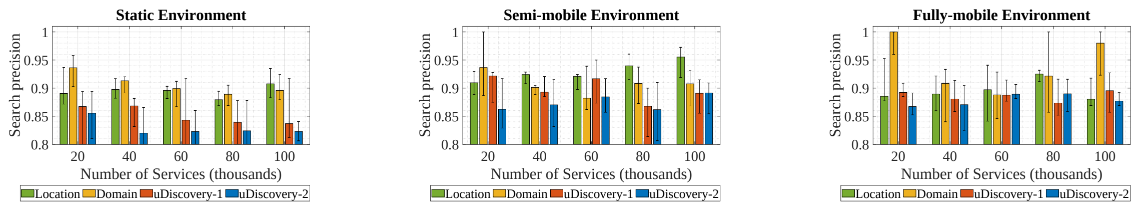


(e) 100,000 Services

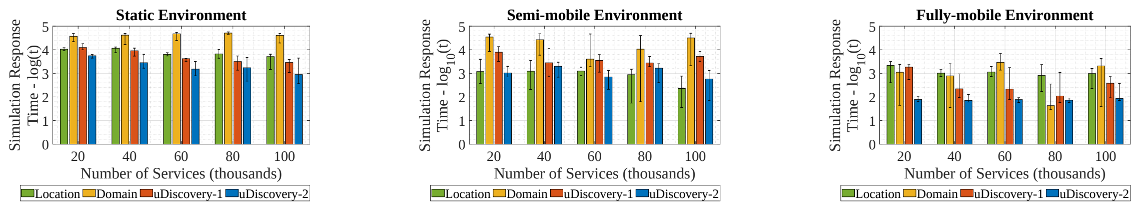
Figure 5.2: Utility Function: General service discovery efficiency with variable number of services and different mobility scenarios.



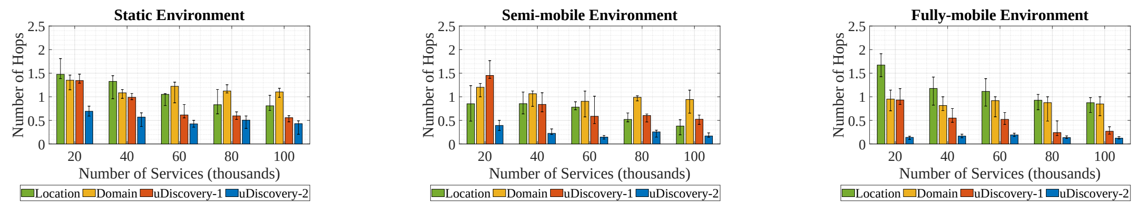
(a) Solved Requests



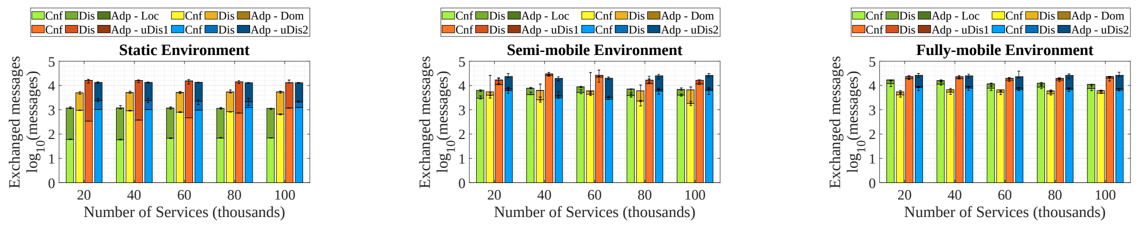
(b) Search Precision



(c) Response Time



(d) Number of Hops



(e) Exchanged Messages

Figure 5.3: Service Discovery Metrics - General service discovery efficiency with variable number of services and different mobility scenarios.

performance as it is highly affected by mobility scenarios. It has a similar performance to the location-based approach in the static environment where the rate varies from 0.69, with 20 thousand services, to 0.88 with 100 thousand, reaching a maximum of 0.91 with 80 thousand services. However, this rate decays to values around 0.2 in mobile scenarios.

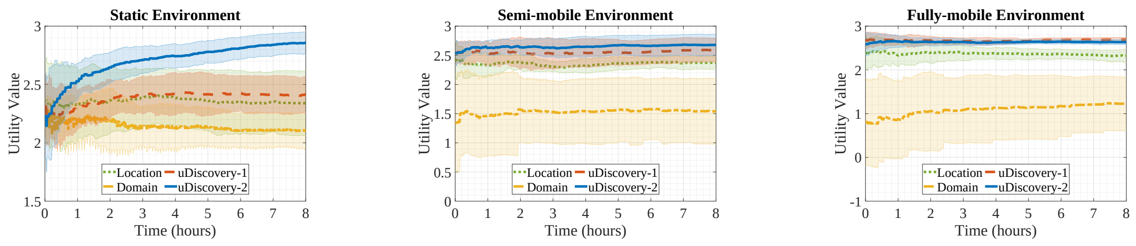
Figure 5.3b illustrates the median **search precision** of each approach. The location and domain-based approaches have better precision than *uDiscovery*1 and *uDiscovery*2. The high precision in the location-based approach (i.e., from 0.88 to 0.96), and in the domain-based approach (i.e., from 0.86 to 0.97) contrasts with their lower rate of solved requests, and is mainly because, with a lower number of solved requests, there is less chance of discovering non relevant services. This trade-off also explains why *uDiscovery*1 and *uDiscovery*2 have lower precision. A higher rate of solved requests affects the search precision because it is more likely to retrieve non relevant services when the rate is higher. $uDiscovery1's$ precision varies from 0.84 to 0.92, $uDiscovery2's$ precision varies from 0.82 to 0.89.

Figure 5.3c illustrates the median simulated **response time** for each approach. The simulated response time is in a logarithmic scale to ease the graph visualisation, and does not represent or approximate times from the real world. However, it still allows the comparison of the approaches' latency in the simulation. Section 5.3 presents a more realistic evaluation of the search latency on real devices. *uDiscovery*2 has the lowest simulated response time in all the scenarios. This low latency is achieved even while it solves more requests than all the baselines, thanks to the semantic overlay and the movement of services between gateways, which puts the right service at the right place at the right time. *uDiscovery*2 has an advantage over *uDiscovery*1, although both use the same semantic overlay, because the movement of services allows more requests to be solved locally, which takes less time. The location and domain-based approaches have the highest latency because they do not always have services in place and need to forward requests to other gateways in the network. This is related to the average number of hops in Figure 5.3d, when the location or domain based approaches have the highest latency, they also need more hops to solve requests. Figure 5.3d presents the median **number of hops** to solve a request. *uDiscovery*2 outperforms the baselines because when services are moved, most of the requests are solved locally (i.e., the median number of hops varies from 0.1, in the best case, to 0.7, in the worst). For all approaches, the number of hops is higher when there are low numbers of services in the environment (e.g., 20 thousand services), and the number of hops decreases when the number of services increases. This is because a larger number of services implies more replication, and more requests can be solved locally. The location, domain, and *uDiscovery*1 approaches have a higher number of hops. However, *uDiscovery*1, as *uDiscovery*2, solves more requests, which indicates that request forwarding based on city context is more effective.

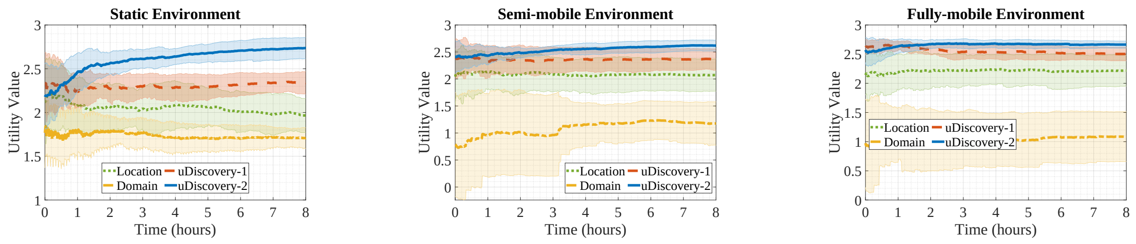
Figure 5.3e presents the median **number of messages** in the network in a logarithmic scale. It shows the median configuration messages, discovery messages, and adaptive messages for each approach. The number of registered services does not have a considerable impact in the number of exchanged messages, but the mobility scenarios does. The results for *uDiscovery*2 illustrate the cost of good performance in previous metrics, which is more configuration, discovery and adaptation messages. *uDiscovery*2 exchanges around 13,000 messages in the static scenario, 22,000 in the semi-mobile environment, and 25,000 in the fully-mobile. *uDiscovery*1 exchanges around 15,000 messages in the static scenario, 24,000 in the semi-mobile, and 26,000 in the fully-mobile scenario. *uDiscovery*2 needs less messages than *uDiscovery*1 in some of the cases because *uDiscovery*2 solves requests locally once services are moved from other gateways, and *uDiscovery*1 needs to forwards messages per each request when this cannot be solved locally. The location and domain-based approaches need less messages, although they offer a poor service discovery efficiency from the rate of solved requests perspective. The location-based approach exchanges around 1,000 messages in the static scenario, 7,000 messages in the semi-mobile scenario, and 15,000 messages in the fully-mobile. The domain-based approach exchanges around 5,000 messages in the static scenario, and 6,000 messages in the semi-mobile and the fully mobile scenario. The proportion of configuration messages is smaller in the static scenario because there are not mobile gateways to be maintained, and each gateway has more information about other gateways in the network where to forward discovery messages. The proportion of configuration messages increases in the semi-mobile and fully-mobile scenarios because of the maintenance of mobile gateways. The proportion of discovery messages decreases because gateways have less knowledge about the dynamic networks.

Scenario 2: varying number of gateways with mobility scenarios

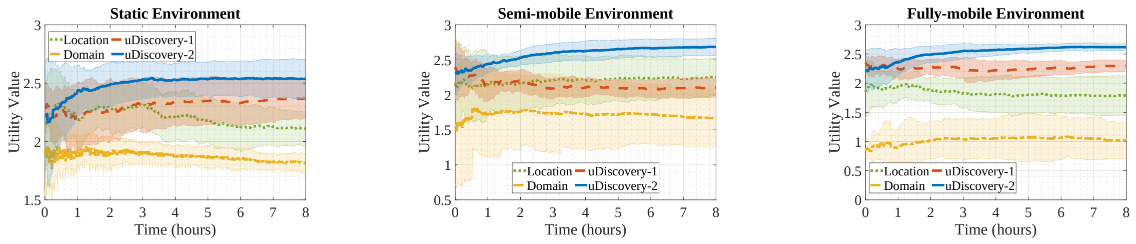
Figure 5.4 presents the utility with variable number of gateways in different mobility scenarios with 100 thousand services. It shows that *uDiscovery*2 has the best utility in all scenarios with exception of the fully-mobile environment with 100 gateways, where *uDiscovery*1 has a similar behaviour and even has a higher utility value than *uDiscovery*2 at hour 8 (i.e., 2.69 for *uDiscovery*1, and 2.64 for *uDiscovery*2). The utility curve is similar between approaches (i.e., location-based, *uDiscovery*1, and *uDiscovery*2) with lower number of gateways (i.e., 100, and 300) because the number of services in each gateway is bigger, so each approach



(a) 100 Gateways



(b) 300 Gateways

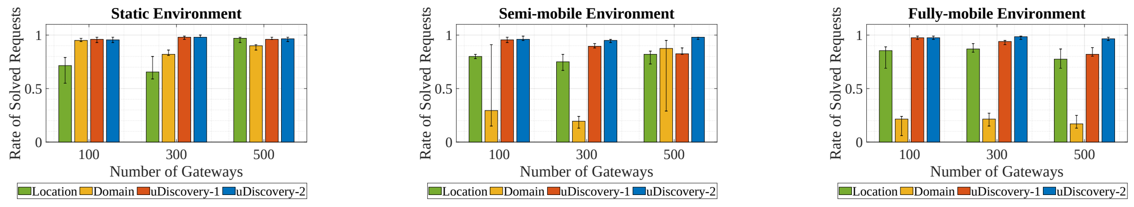


(c) 500 Gateways

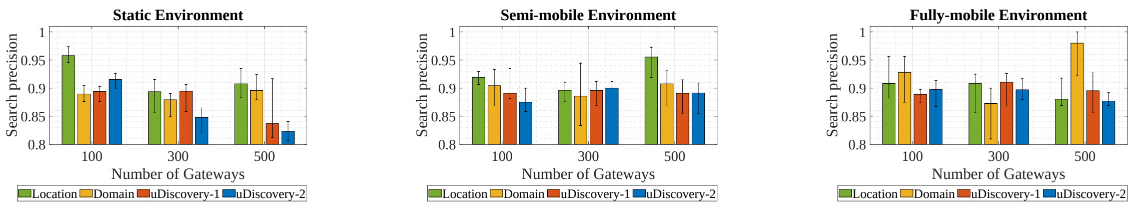
Figure 5.4: Utility Function: General service discovery efficiency with variable number of gateways and different mobility scenarios.

is likely to solve more requests. The rate of solved requests in Figure 5.5a supports the similarity between results from $uDiscovery1$ and $uDiscovery2$. They have a similar rate of solved requests with 100 and 300 gateways. This figure also explains why $uDiscovery2$ has a better utility curve with 500 gateways, as $uDiscovery2$ solves more requests than all the baselines. Figure 5.5d explains the similarity between the utility curves from $uDiscovery2$ and the location-based approach as both uses fewer number of hops to discover services with 100 gateways. This figure also shows that $uDiscovery2$ has a clear advantage in all static scenarios, because of the difference in the number of hops. There is a considerable difference between the number of hops for $uDiscovery2$ (i.e., close to 0), and the number of hops for $uDiscovery1$ and the domain-based approach in the static scenario. This difference is reduced in the semi-mobile and fully-mobile scenarios, which also explains why the utility curves are similar in these scenarios with 100 and 300 gateways. The domain-based approach has the worst utility curve in all cases. It solves less requests, and needs more time and hops.

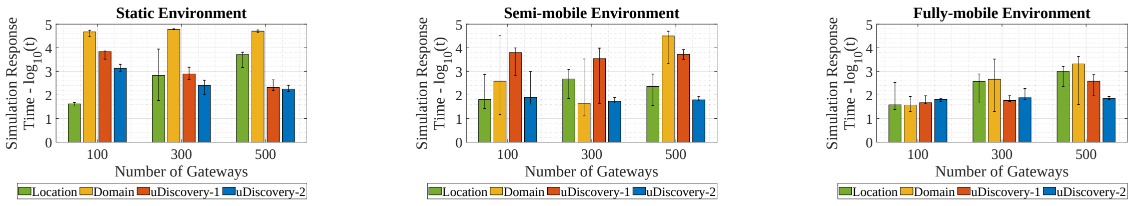
Figure 5.5 shows the utility function's constituent service discovery metrics for each approach



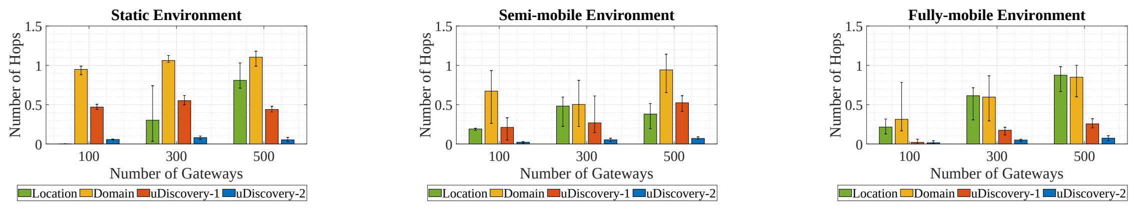
(a) Solved Requests



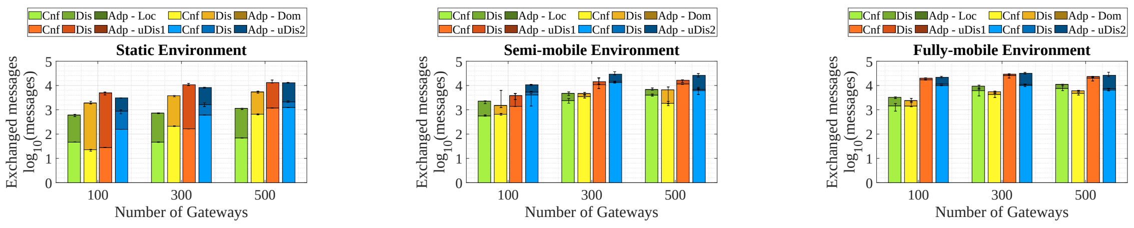
(b) Search Precision



(c) Response Time



(d) Number of Hops

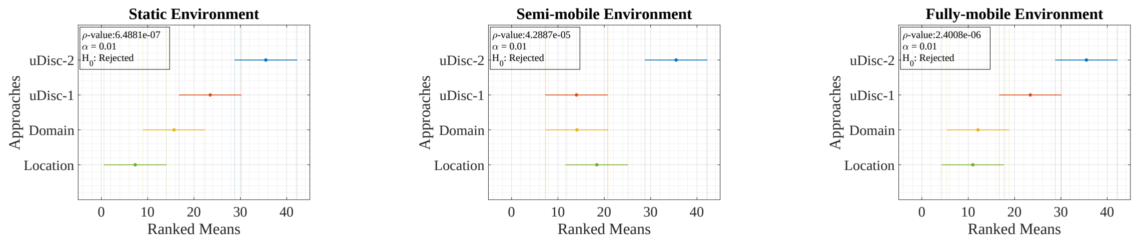


(e) Exchanged Messages

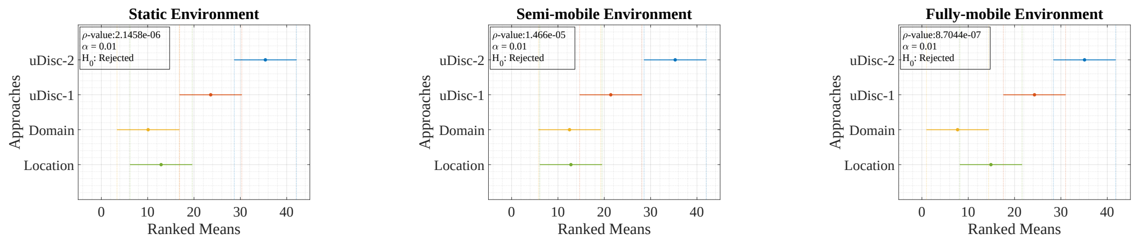
Figure 5.5: Service Discovery Metrics - General service discovery efficiency with variable number of gateways and different mobility scenarios.

in the experimental scenario 2. The **rate of solved requests** (Figure 5.5a) is not affected by the number of gateways, but it is impacted by the mobility scenarios, similarly to the experiments with variable number of services (Figure 5.3a). **Search precision** has also a similar behaviour as in previous experiments. Approaches that solve fewer requests have a higher precision (i.e., location and domain-based approaches). *uDiscovery*1 and *uDiscovery*2 have an acceptable precision (i.e., from 0.82 to 0.91) and a high rate of solved requests (i.e., above 0.8 for *uDiscovery*1, and above 0.94 for *uDiscovery*2) (Figure 5.5b). Both **response time** and **number of hops increase** with the size of the network for the location and domain-based approaches (Figures 5.5c and 5.5d). Gateways in these approaches have more information about other gateways as to where to forward requests, when the size of the network increases. Response time and hops keep constant for *uDiscovery*1 and *uDiscovery*2, despite the number of gateways because the *ant colony* forwarding mechanism limits the request forwarding to a subset of relevant gateways in both *uDiscovery* versions (Section 3.4.1).

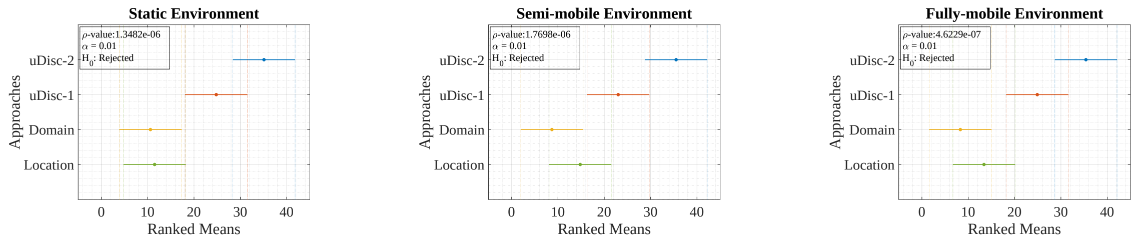
The **number of exchanged messages** increases according to the network size and the mobility scenarios in each approach (Figure 5.5e). This increment is because mobile gateways require more configuration messages for their maintenance, and the number of configuration messages in the initialisation process increases with the number of gateways. The location-based approach exchanges around 600 messages with 100 gateways in the static scenario, 2,200 in the semi-mobile, and 3,000 in the fully mobile. The same approach exchanges around 1,141 messages with 500 gateways in the static scenario, 6,000 in the semi-mobile, and 13,000 in the fully-mobile. The domain-based approach has a similar behaviour. This approach exchanges from around 1,800 messages, in the static scenario with 100 services, to 6,000 messages in the fully-mobile. *uDiscovery*2 exchanges around 3,000, 8,000, and 13,000 messages with 100, 300, and 500 gateways respectively in the static scenario; around 9,000, 29,000, and 27,000 messages with 100, 300, and 500 gateways in the semi-mobile environment; and around 20,000, 30,329, and 27,000 messages with 100, 300, and 500 gateways in the fully-mobile environment. *uDiscovey*1 has a similar performance as the exchanged messages varies from 5,000, with 100 gateways in the static scenario, to 22,000 with 500 gateways in the fully-mobile scenario. The cost of a better performance in previous metrics for both versions of *uDiscovery* is a higher number of exchanged messages, as previously in Figure 5.3e). Similarly, The proportion of configuration messages increases when there are mobile gateways, and the proportion of discovery messages decreases when gateways have less knowledge about the network.



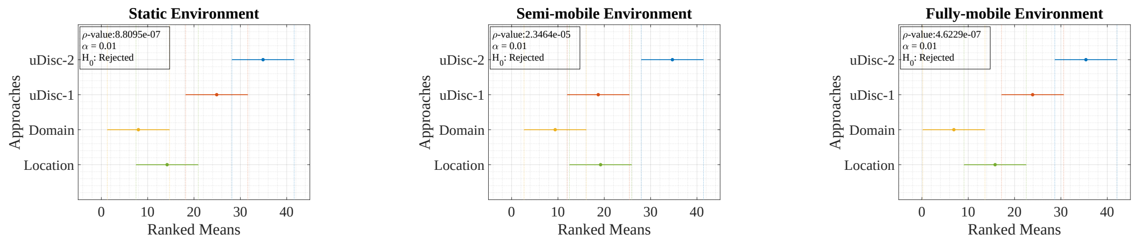
(a) 20,000 Services



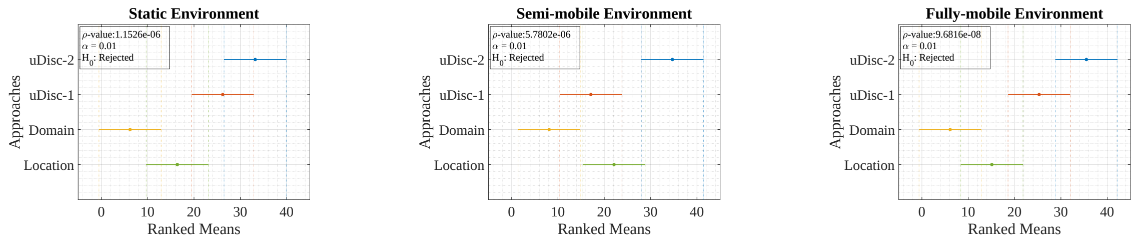
(b) 40,000 Services



(c) 60,000 Services



(d) 80,000 Services



(e) 100,000 Services

Figure 5.6: Statistical Test on General Service Discovery Efficiency with Variable Number of Services: Multiple Comparison of Ranked Means.

Statistical Analysis

A Kruskal-Wallis test was performed on the utility value at hour 8 in each experiment to. This test determines if there are statistically significant differences between the approaches' performance. The null hypothesis of this test specifies that samples of the utility value from each approach are subsets from the same population (i.e., $H_0 : (a, b, c, ..., n) \subseteq p$) with confidence interval $\alpha = 0.01$. Figure 5.6 presents the results from this test. Each sub-figure shows the comparison of the ranked means, the ρ-value, and the decision (i.e., to reject or not reject H_0). The projected lines from each ranked mean shows whether two approaches are statistically significantly different. Approach A and B are not different, if a projected line from approach A intersect the ranked mean of approach B.

H_0 is rejected in all cases because there are significant differences between approaches' performance. *uDiscovery*2 is statistically significantly different from location-based, and domain-based approaches in all scenarios, with the exception of the semi-mobile environment with 100 thousand services. The ranked means have an overlap in this case, which made the differences not significant. This is because the location-based approach has a rate of solved request close to the $uDiscovery2's$ rate, with a lower latency. The efficiency of *uDiscovery*2 is statistically significantly better than the location-based and domain-based approaches in the other 14 cases. *uDiscovery*2 is statistically significantly better than *uDiscovery*1 in 4 out of 15 cases. The means of *uDiscovery*1 and *uDiscovery*2 overlap in most of the cases. Although they are not statistically different, the small overlaps can indicate that *uDiscovery*2 is more efficient than *uDiscovery*1.

Discussion

The experiments on the general service discovery efficiency evaluated *uDiscovery* in a city simulated environment. Two versions are evaluated to observe the effect of adaptive properties in the discovery performance. *uDiscovery*1 organises services and forwards requests based on urban context (Section 3.4.1). *uDiscovery*2 adds adaptive properties based on the model in Section 3.4.2. Results show that both versions of *uDiscovery* can improve the discovery efficiency compared with baselines. Both have better utilities values which reach a maximum of 2.93 for *uDiscovery*2, and a maximum of 2.34 for *uDiscovery*1. These utility values show that *uDiscovery*1 and *uDiscovery*2 solve more requests (i.e., rate of solved requests close to 1) with high search precision (i.e., between 0.8 and 0.92) and lower latency. *uDiscovery*2 needs less hops to discover services than *uDiscovery*1, but it exchanges more messages for adaptation purposes. *uDiscovery*1 and *uDiscovery*2 have in overall better

Table 5.4: Experiments Parameters for Unforeseen Events Study.

Length of time for service discovery	8 hours
Event start time	Hour 7
# of consumer requests	170
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	500
Mobility scenarios	Static; semi-mobile; fully-mobile
Experiments Scenarios	1. Services X Mobility
Replication	10 rounds each experiment
Hops limit (See Appendix A)	- 5 for location-based and uDiscovery - 3 for domain-based
Distance (See Appendix A)	- 100 for location-based and uDiscovery
Number of Domains (See Appendix A)	- 5 for domain-based

performance with regard to the utility value, rate of solved requests, and search latency. However, both versions of *uDiscovery* need to exchange more messages, which might be a serious drawback in scenarios where the network congestion is prioritised against the other metrics.

The number of gateways do not have an impact on the approaches' performance with regard to the rate of solved requests and the search precision. However, the discovery latency, number of hops and exchanged messages increase with the network size. Following studies consider the largest network size in the experiments (i.e., 500 gateways), to evaluate approaches' performance under the most challenging conditions with regard to latency, number of hops and exchanged messages. Results of this study address the question **RQ 1** in the Section 1.2.4. The inclusion of urban context improves service discovery efficiency in smart cities scenarios.

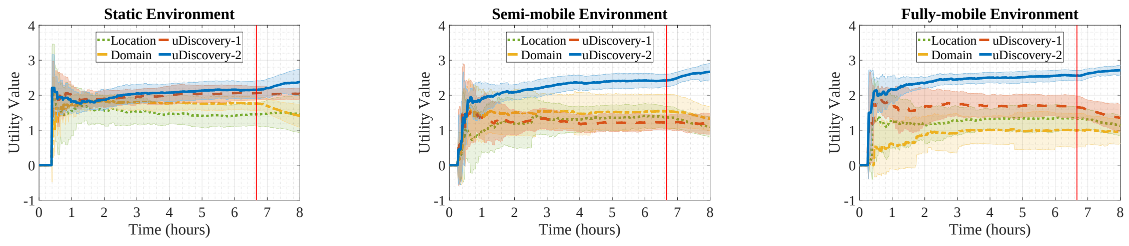
5.2.4 Unforeseen Events Study

This study measures the discovery efficiency of each approach in the presence of an unforeseen event, according to the metrics defined in Section 5.1.1. Table 5.4 presents the experiments design that this study follows. Each experiment simulates 8 hours of services requests to have enough time to simulate one unforeseen event and analyse the approaches' performance with and without the event. Each period has an unforeseen event that starts before hour 7 (i.e., after 400 minutes). The consumer requests around 170 services in each experiment, with requests made according to its location before the event, and then requests made for services relating to domains not managed by the gateway, after the event starts. The period between requests is shorter after the event starts and that is why the number of requests is greater than the number of requests in the previous study (Section 5.2.3). This shorter period also generates a small variability in the number of requests, which is handled by the experiments replication. The experiments' parameters are the number of services, and the mobility scenarios, which are combined in one experimental scenario to determine the influence of each variable in the service discovery process. Each experiment is replicated 10 times. Particular parameters of each approach are selected according to the Appendix A.

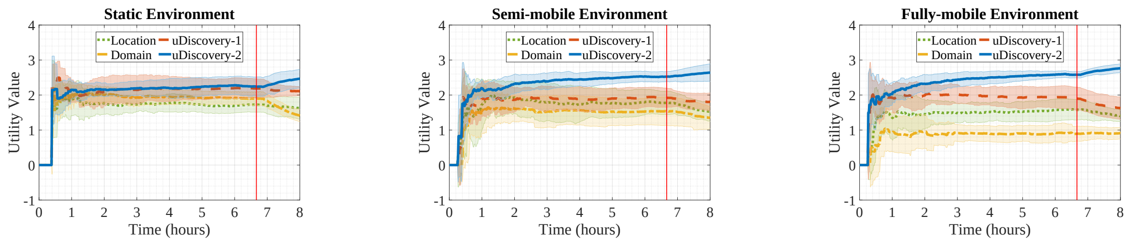
5.2.4.1 Unforeseen Events Study Results

Figure 5.7 illustrates the utility function for each approach in this study. *uDiscovery*2 has a better utility in all scenarios as its value is between 2.43, in the worst case, and 2.88, in the best case, at hour 8. *uDiscovery*1 is the second best with utility values between 1.17 and 2.24 at hour 8. The utility value is between 1.03 and 2.18 in the location-based approach, and between 0.91 and 1.57 in the domain-based approach at hour 8. The behaviour of the approaches is similar to the study on general efficiency (Section 5.2.3), until the event starts. $uDiscovery2's$ utility value is initially improved, and then maintained, in all cases while the baselines' utility is negatively impacted after the event with no subsequent improvement, in all cases. The improvement in the utility curve of *uDiscovery*2 is expected because once *uDiscovery*2 moves the services related to the event, the requests are solved in a more efficient way. The domain and urban-based approaches have the worst performance because they either cannot solve the requests after the event starts, or they have to forward them to other gateways, adding latency and network overhead. This is supported by Figure 5.8c and 5.8d which show that the location and domain-based approaches need more time and hops to solve requests than *uDiscovery*2. The utility curve of the location-based approach is less affected by the event than the domain-based and *uDiscovery*1, because the gateway could have services related to the unforeseen event by mere randomness. The difference between *uDiscovery*2 and the rest of the approaches is bigger when the number of services is lower (i.e., 20 and 40 thousand) and there are mobile gateways. The exchange of services between gateways enables a higher rate of solved requests (Figure 5.8a), with lower latency (Figure 5.8c), and less number of hops (Figure 5.8d) because requests are solved locally.

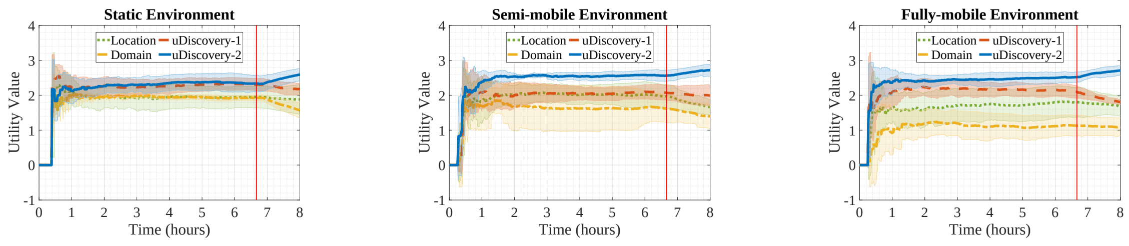
Figure 5.8 presents the service discovery metrics for each approach with a variable number of services, in different mobility environments, and in the presence of an unforeseen event. The **rate of solved requests** is similar for *uDiscovery*1 and *uDiscovery*2 in the static



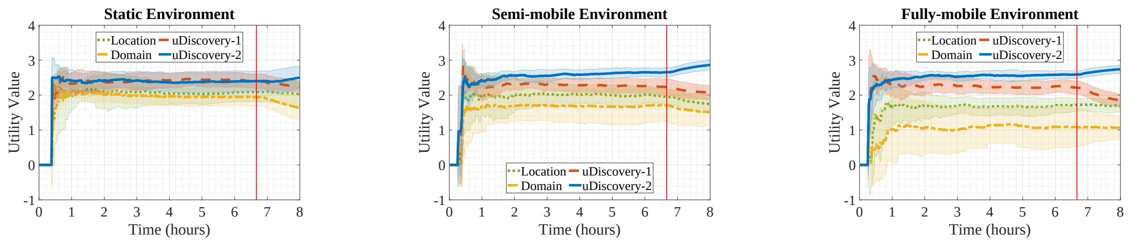
(a) 20,000 Services



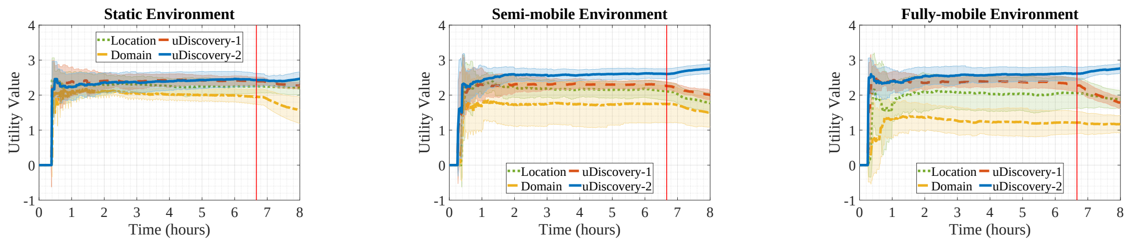
(b) 40,000 Services



(c) 60,000 Services

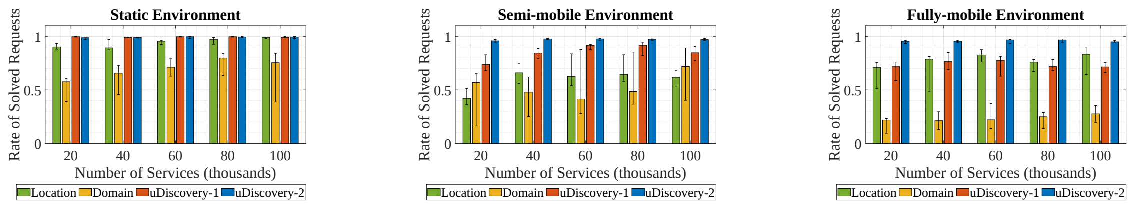


(d) 80,000 Services

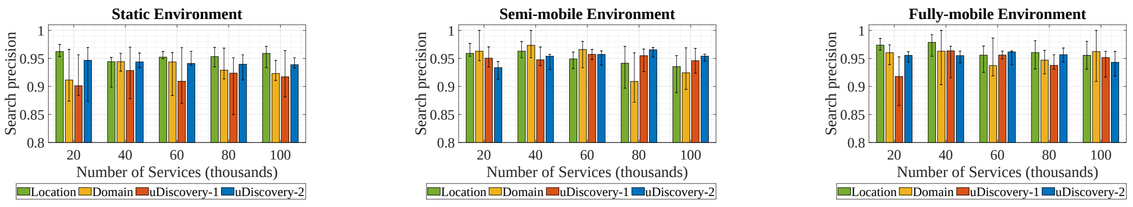


(e) 100,000 Services

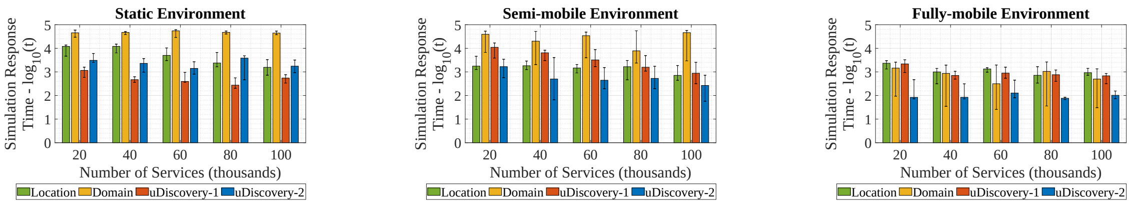
Figure 5.7: Utility Function: Unforeseen Events Study.



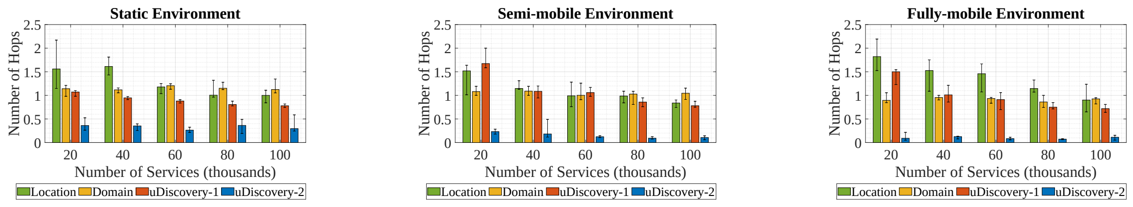
(a) Solved Requests



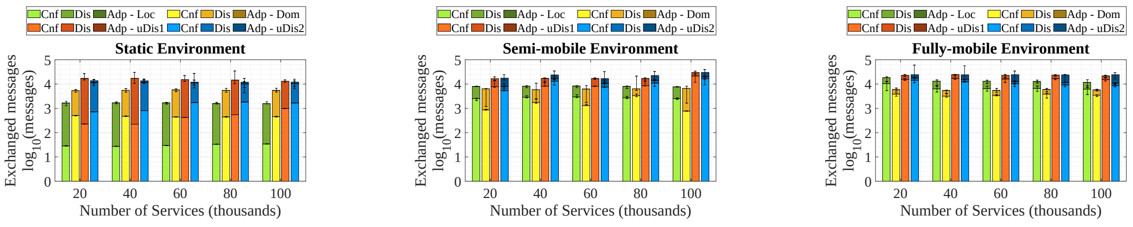
(b) Search Precision



(c) Response Time



(d) Number of Hops



(e) Exchanged Messages

Figure 5.8: Service Discovery Metrics - Unforeseen Events Study.

environment (i.e., close to 1). The rate of solved requests is also high for the location-based approach in the static scenario. It varies from 0.9, with 20 thousand services, to around 1.0 with 100 thousand services. The domain-based approach has the lowest rate in the static scenario. It varies from 0.6, with 20 thousand services, to 0.8 with 100 thousand. The unforeseen event does not affect the rate of solved requests for $uDiscovery1$, with different numbers of services, or the location-based approach, with a high number of services, because they have knowledge about other gateways and solve the requests by forwarding them. They need a higher number of hops, as Figure 5.8d illustrates. The rate of solved requests is affected in all the approaches, with exception of $uDiscovery2$, because they have less information about other gateways. $uDiscovery2$ is not affected by mobile gateways because this approach moves services to the local registry. These services are accessible even when the gateway, from where they were moved, is not available because of its mobility.

The location and domain-based approaches have the highest **search precision** scores (i.e., all of them equal or more than 0.95). This is because they solve less requests, and there is less chance to retrieve non relevant services. $uDiscovery2$ has a better search precision (i.e., greater or equal to 0.93) than in the previous study on general discovery efficiency (Figure 5.3b). Once the event starts and services are moved, it is more likely that these requests, which are similar (i.e., belong to the same domain), are solved with the right set of services. The simulated **response time** is lower for $uDiscovery2$ in the semi-mobile and fully-mobile scenarios because requests are solved locally, and other approaches take longer time forwarding requests to other gateways with changing, and partial, network information (Figure reffig:timeScenario1). The location and domain-based have higher latency because gateways do not have enough service information in the local repository and need to forward more requests. Figure 5.8d shows that $uDiscovery2$ always needs fewer hops than the other approaches, because services are moved locally, specially when there are mobile gateways where the number of hops is close to 0 (i.e., the approach solves the request, where it is received).

Figure 5.8e shows the **number of exchanged messages** for each approach in the different scenarios. The cost of a more efficient service discovery is a higher number of exchanged messages for $uDiscovery1$ and $uDiscovery2$, as in the previous study. Similarly, there are more configuration messages in all approaches when there are mobile gateways, and less discovery messages because limited knowledge about the dynamic network. The location and domain-based approaches exchange less messages, although they discover less services in overall. The location-based approach exchanges around 1,600 messages in the static scenario, 8,000 in the semi-mobile, and 15,000 in the fully-mobile. The domain-based approach exchanges a similar number of messages for different cases (i.e., around 6,000), but the proportion of configuration and discovery messages varies from static to semi and fully-mobile scenarios. This low number of exchanged messages in the domain-based approach is because the limit of hops is lower than in other approaches according to the Appendix A. $uDiscovery1$ exchanges around 17,000 messages in the static scenario, 20,000 in the semi-mobile, and 22,000 in the fully-mobile. $uDiscovery2$ exchanges around 14,000, 22,000, and 24,000 in the static, semi-mobile and fully-mobile scenarios respectively. $uDiscovery2$ exchanges less messages than $uDiscovery1$ in the static scenario because requests are solved locally, when services are moved between gateways, and $uDiscovery1's$ gateways have more information about other gateways as where to forward discovery messages.

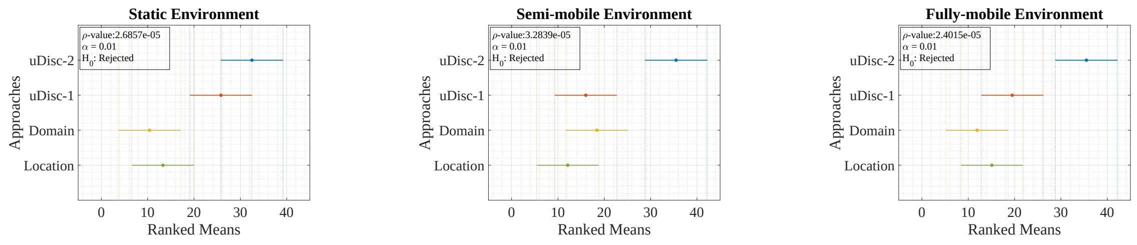
Statistical Analysis

A Kruskal-Wallis test is performed on the utility value at hour 8 in each experiment. This test is used to identify whether the performance of evaluated approaches is statistically significantly different. The null hypothesis of this test specifies that samples of the utility value of each approach are subsets from the same population (i.e., $H_0 : (a, b, c, ..., n) \subseteq p$) with confidence interval $\alpha = 0.01$. The projected lines from each ranked mean shows whether two approaches are statistically significantly different, as in previous study.

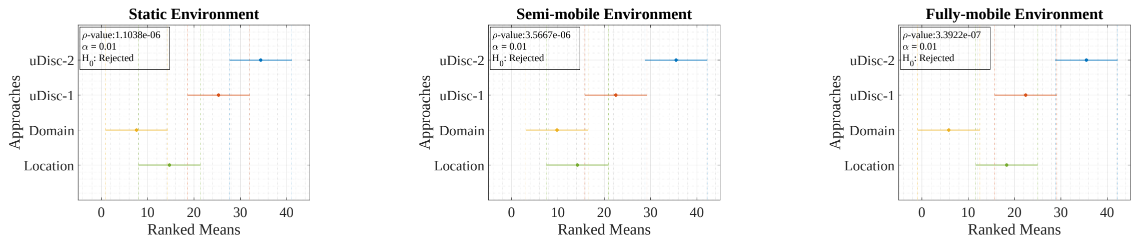
Figure 5.9 presents the results from this test. H_0 is rejected in all the cases, which means that there are statistically significant differences between approaches performance. $uDiscovery2$ is statistically different from the location and domain-based approaches in 13 out of 15 cases. The location-based approach and $uDiscovery2$ are not significant different in the static scenarios with 80 and 100 thousand services, where there is a high availability of services and gateways information. $uDiscovery2's$ performance is statistically significantly better than the location and domain-based approaches in all the other scenarios. $uDiscovery2$ is statistically significantly better than $uDiscovery1$ in 4 out of 15 cases. The difference is more notorious when there are mobile gateways because services might not be available, and $uDiscovery1$ has less accurate information about other gateways in the network. Although they are not statistically different, the overlaps are small and can still indicate that $uDiscovery2$ is more efficient than $uDiscovery1$ in mobile scenarios.

Discussion

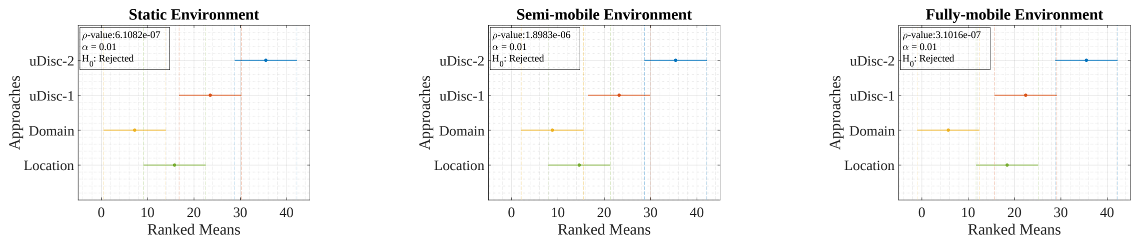
This study evaluated $uDiscovery1$ and $uDiscovery2$ in the presence of unforeseen events in a simulated city environment. Results show that $uDiscovery2$ responds to the unfore-



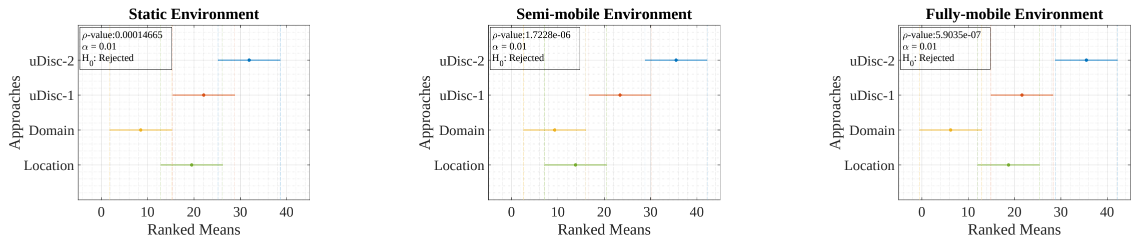
(a) 20000 Services



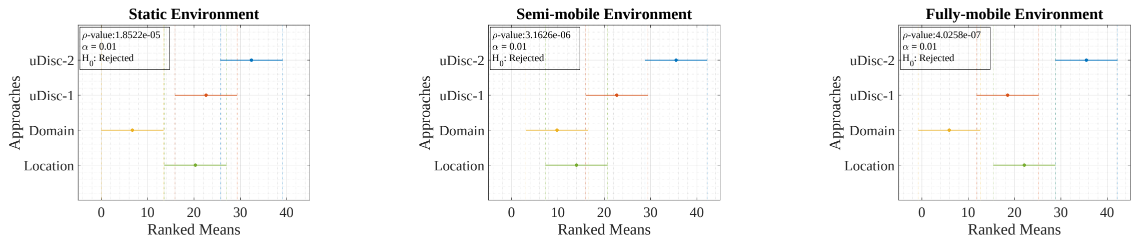
(b) 40000 Services



(c) 60000 Services



(d) 80000 Services



(e) 100000 Services

Figure 5.9: Statistical Test Unforeseen Events Study: Multiple Comparison of Ranked Means.

Table 5.5: Experiments Parameters for the Scheduled Events Study.

Length of time for service discovery	10 hours
Events duration	1 hour
Events start time	First event: Hour 3 Second event: Hour 6 Third event: Hour 9
# of consumer requests	500
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	500
Mobility scenarios	Static; semi-mobile; fully-mobile
Scenarios	1. Services X Mobility
Replication	10 rounds each experiment
Hops limit (See Appendix A)	- 5 for location-based and uDiscovery - 3 for domain-based
Distance (See Appendix A)	- 100 for location-based and uDiscovery
Number of Domains (See Appendix A)	- 5 for domain-based

seen event and keeps a more efficient service discovery than baselines. *uDiscovery*2 solves more requests (i.e., rate of solved requests around 0.95 despite gateways mobility) with less number of hops (i.e., close to 0 hops), and offers a high search precision (i.e., over 0.9) with lower latency than baselines. *uDiscovery*1 also has a better performance than baselines, but *uDiscovery*2 is better because of the adaptive properties (Section 3.4.2), which can be observed in rate of solved requests of each approach when there are mobile gateways. Both versions of *uDiscovery* improve service discovery efficiency at the cost of exchanging more messages, which can be an issue in scenarios where network efficiency is more important than other metrics. The location and domain-based approaches exchange around 15,000 and 6,000 messages in the worst case respectively. *uDiscovery*1 and *uDiscovery*2 exchange more than 20,000 under the same conditions. Results from this study address the questions **RQ 1**, and **RQ 2** in the Section 1.2.4. The inclusion of urban context improves the service discovery efficiency in smart cities scenarios. It is complemented by adaptive properties that respond to city changes and maintains more efficient service discovery over time.

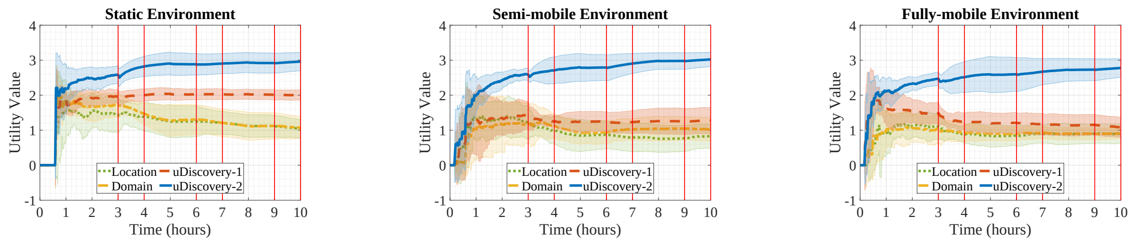
5.2.5 Scheduled Events Study

This study measures the discovery efficiency of each approach according to the metrics defined in the Section 5.1.1, in the presence of scheduled events. Table 5.5 presents the experiments design that this study follows. Each experiment simulates 10 hours of requests to have enough time to simulate more than one scheduled event (i.e., 3 scheduled events). Each event lasts 1 hour: the first event starts at hour 3, the second at hour 6, and the third at hour 9. The consumer performs around 500 requests to a gateway in each experiment to evaluate the approaches performance with different type of requests between events, and when the event is happening. These requests are made according to consumer's location between events, and requests made for services relating to domains not managed by the gateway when an event is happening. The experiments' parameters are the number of services, and the mobility scenarios, which are combined to determine their influence the discovery performance. Each experiment is replicated 10 times.

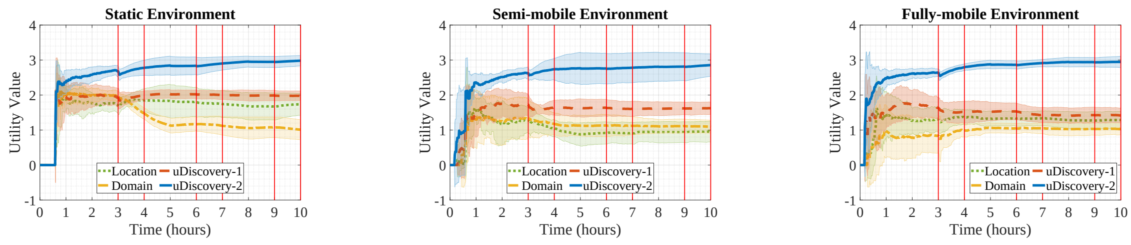
5.2.5.1 Scheduled Events Study Results

Figure 5.10 illustrates the utility function for each approach in the presence of scheduled events. *uDiscovery*2 handles the scheduled events better than all the baselines. The adaptation increases the utility value because the gateway has the set of services ready to solve the requests before the events start. The utility value has its highest increment after the first event starts, which is expected because once the second and third events start the metrics are already high and the effect of the adaptation is to keep them at that level. The utility also increases between events because of the effect of Algorithm 5, which improves the utility continuously by updating the threshold. The baselines are affected by the events, as their utility value falls when an event starts. *uDiscovery*2 has the best utility values from 2.7, in the worst case, to 3.04 in the best case, at hour 10. This value varies from 1.11 to 2.01 for *uDiscovery*1. The location-based approach has an utility value that varies from 0.85 to 1.96, and the domain-based has an utility that varies from 0.99 to 1.05, at hour 10. Similarly to the unforeseen events study (Section 5.2.4.1), the domain-based and urban-based approaches are more affected by the events because they do not have the services in place for discovery and must forward requests to other gateways. Figure 5.11d shows that the location, domain-based and *uDiscovery*1 approaches need more hops compared against *uDiscovery*2. The location-based approach keeps a constant, but lower, utility value despite the events in some cases because the gateway can have services related to events by mere randomness. *uDiscovery*2 has the best utility values despite gateways' mobility because services are available locally, as explained in the unforeseen events study (Section 5.2.4.1).

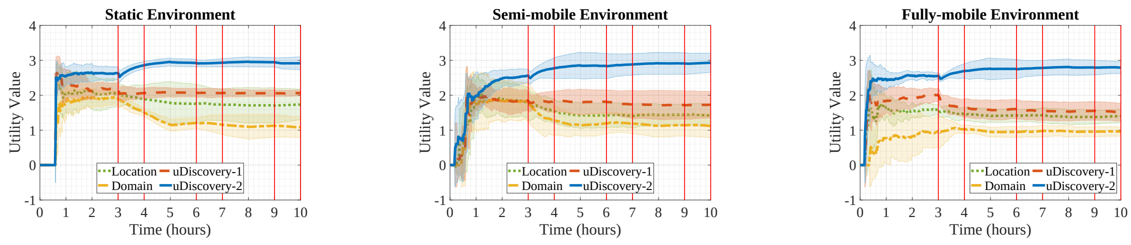
Figure 5.11 presents the detailed metrics for each approach in the presence of scheduled events. *uDiscovery*2 has the best **rate of solved requests** (i.e., close to 1) despite the number of services and the mobility scenarios (Figure 5.11a). *uDiscovery*1 has the second



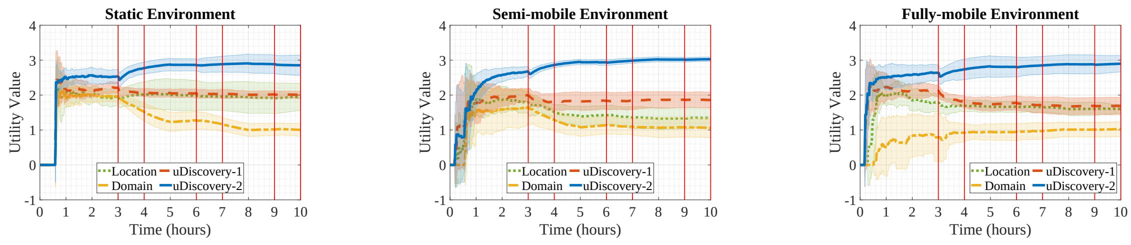
(a) 20,000 Services



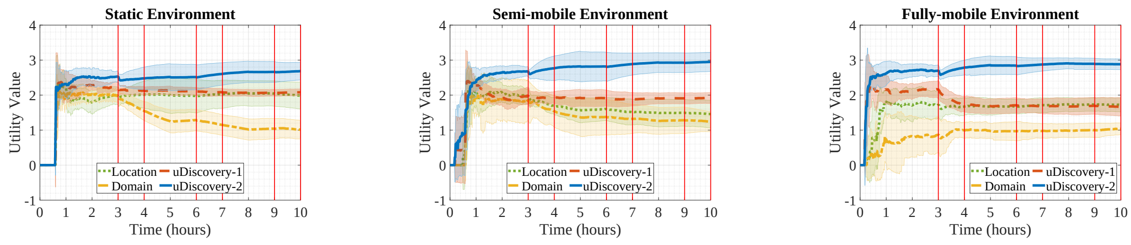
(b) 40,000 Services



(c) 60,000 Services

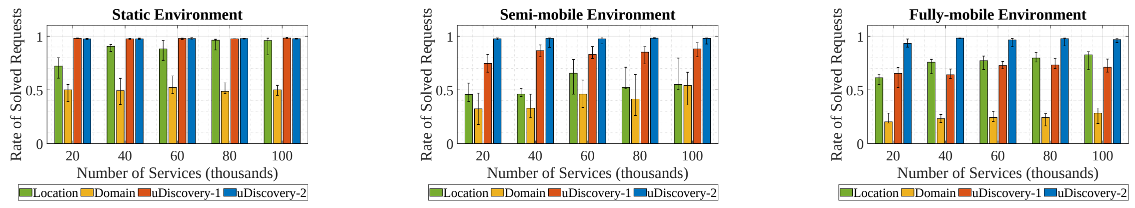


(d) 80,000 Services

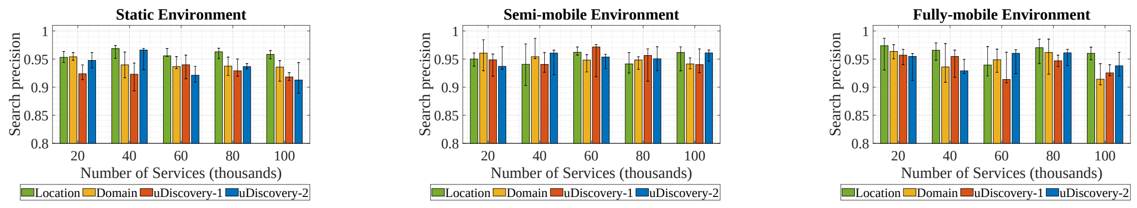


(e) 100,000 Services

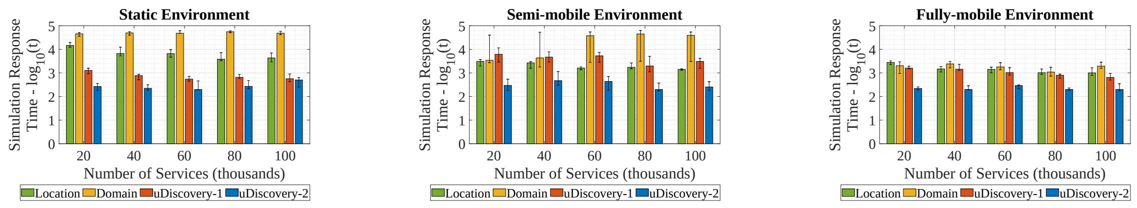
Figure 5.10: Utility Function - Scheduled Events Study.



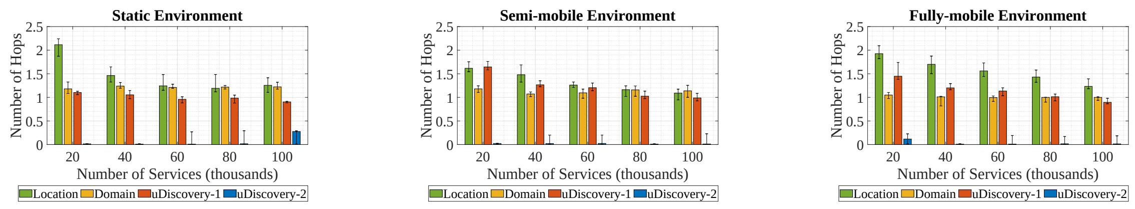
(a) Solved Requests



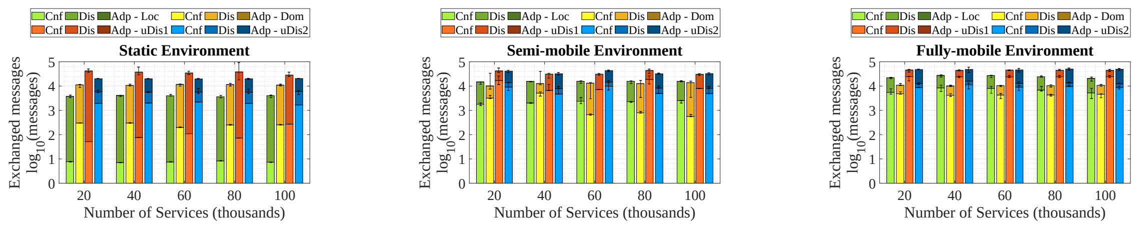
(b) Search Precision



(c) Response Time



(d) Number of Hops



(e) Exchanged Messages

Figure 5.11: Service Discovery Metrics - Scheduled Events Study.

best rate of solved requests. Its rate is close to 1.0 in the static scenario, but it is impacted by mobile gateways in the semi and fully-mobile scenarios. The location-based approach reaches a similar rate of solved requests in the static scenario with 100 thousand services. But, it is also impacted in the mobility scenarios. The location-based and $uDiscovery1$ approaches have a good rate of solved requests in static scenarios, but they need more time and hops to discover services (Figures 5.11c and 5.11d). The domain-based approach has the lowest rate of solved requests (i.e., under 0.5) in the different mobility scenarios with variable number of services. Figure 5.11b shows the search precision for different approaches under different circumstances. All approaches have a **search precision** above 0.9, but there is no approach better than others in most of the cases. This is caused because of the request randomness and the events that favour $uDiscovery2$ over others. However, $uDiscovery1$ and $uDiscovery2$ have a similar search precision to the location and domain-based approaches even when they solve more requests.

$uDiscovery2$ has the lowest simulated **search latency** in most of the scenarios followed by $uDiscovery1$, the location-based approach, and the domain-based approach (Figure 5.11c). $uDiscovery2$ spends less time to discover services because it solves requests locally. Other approaches spend more time because they forward requests to other gateways, according to Figure 5.11d, which illustrates the median **number of hops** that each approach needs to solve requests. The location-based approach needs more hops followed by $uDisocvery1$ and the domain-based approach, which need a similar number of hops. The median number of hops for $uDiscover2$ is close to 0, which demonstrates that requests are solved locally. Figure 5.11e presents the exchanged messages for each approach. $uDiscovery2's$ good performance in previous metrics costs the highest number of exchanged messages, as in previous studies. All approaches need more messages compared against previous studies because the experiments of this study last longer. The location-based approach exchanges around 4,000, 15,000, and 24,000 messages in the static, semi-mobile and fully-mobile scenarios respectively. The domain-based approach exchanges a similar number of messages in different scenarios, which vary from 10,000 to 12,000. The domain-based approach is more efficient in terms of network usage because the limits of hops is lower than in the other approaches according to the Appendix A. $uDiscovery1$ also exchanges a similar number of messages in different scenarios (i.e., around 40,000). This is because in static scenarios gateways send more discovery messages because they have more knowledge about the network, and in the mobile scenarios the maintenance of mobile gateways require more configuration messages. $uDiscovery2$ requires a higher number of messages than $uDiscovery1$ in the mobile scenarios (i.e., around

45,000), because *uDiscovery*2 adds adaptation messages to the configuration, and discovery messages. However, *uDiscovery*2 needs less messages in the static scenarios because gateways solve requests locally, once services are moved. The proportion of configuration messages is lower in the static environment for all approaches because there are not mobile gateways to maintain, as in previous studies.

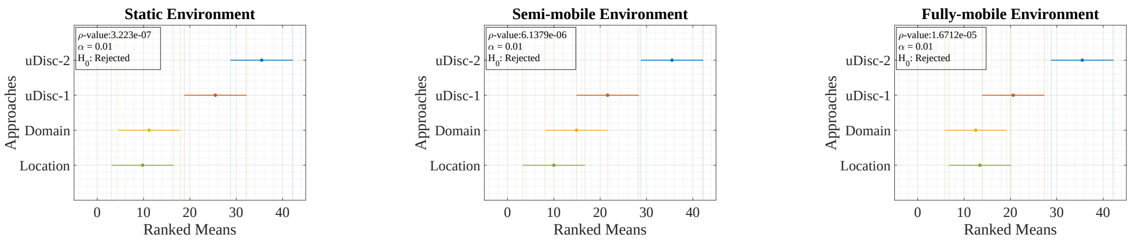
Statistical Analysis

A Kruskal-Wallis test was performed on the utility value of different approaches in this study. This test is used to identify whether the performance of evaluated approaches is statistically significantly different. The null hypothesis of this test specifies that samples of the utility value of each approach are subsets from the same population (i.e., $H_0 : (a, b, c, ..., n) \subseteq p$) with confidence interval $\alpha = 0.01$. The projected lines from each ranked mean shows whether two approaches are statistically significantly different, as in previous studies.

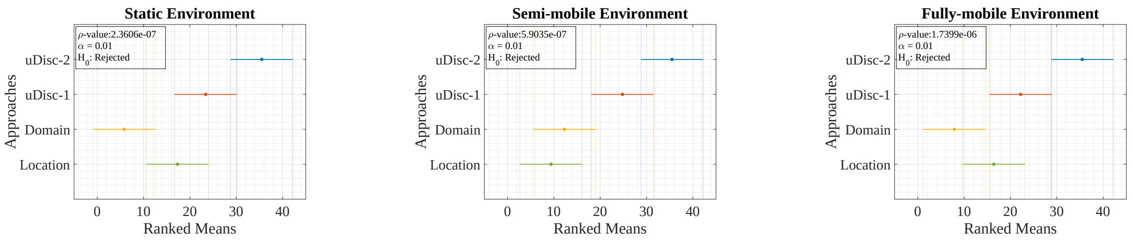
Figure 5.12 presents the results from the test in this study. H_0 is rejected in all the cases which means that there are statistically significant differences between approaches' performance. *uDiscovery*2 is statistically different from the location and domain-based approaches in all the cases. This means that $uDiscovery2's$ performance is statistically significantly better than location and domain-based approaches in all the scenarios. *uDiscovery*2 is statistically significantly better than *uDiscovery*1 in 5 out of 15 cases. Although *uDiscovery*1 and *uDiscovery*2 are not significantly different, *uDiscovery*2 can be considered more efficient than *uDiscovery*1 because of the small overlaps between their ranked means in all cases. Adaptive properties complement the use of urban information to offer a more efficient service discovery.

Discussion

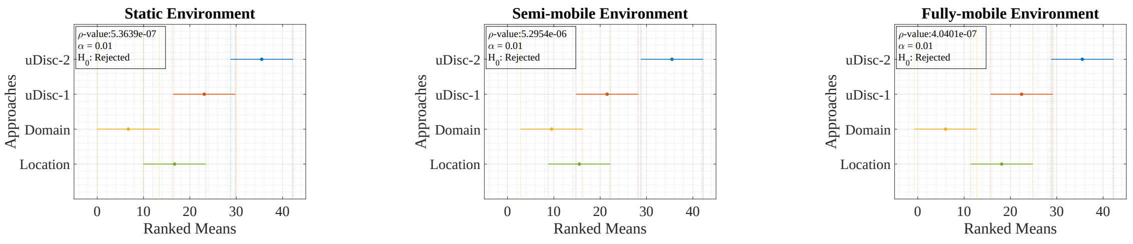
This study evaluated *uDiscovery* in the presence of scheduled events in a simulated city environment. Results show that *uDiscovery* responds to the scheduled events and improves the discovery efficiency over time. *uDiscovery*1 and *uDiscovery*2 have a better performance than baselines, but *uDiscovery*2 is better because of the adaptive properties (Section 3.4.2). *uDiscovery*2 reaches a rate of solved requests close to 1, despite the mobility scenarios, offers a search precision over 0.9 in all cases, has lower latency than all approaches, and solves requests in the gateway that receives them (i.e., 0 hops). *uDiscovery* improves the discovery efficiency at the cost of more messages exchanged in the network, which might be a concern in discovery efficiency is measure from the network usage perspective. Both versions of *uDiscovery* requires more than 40,000 messages in the worst case, the location-based



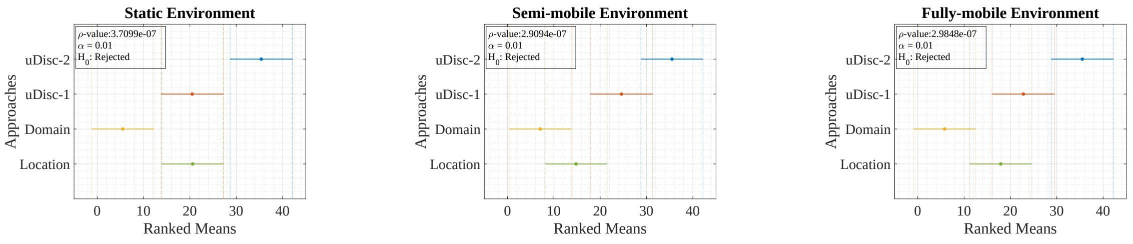
(a) 20,000 Services



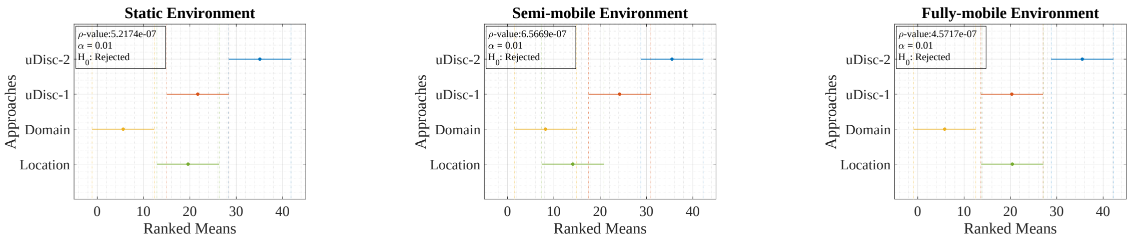
(b) 40,000 Services



(c) 60,000 Services



(d) 80,000 Services



(e) 100,000 Services

Figure 5.12: Statistical Test Scheduled Events Study: Multiple Comparison of Ranked Means.

Table 5.6: Experiments Parameters to Evaluate *uDiscovery*.

Length of time for service discovery	30 periods of 6 hours
Events duration	1 hour
Events start time	First event: Hour 2 each period Second event: Hour 4 each period
# of services	20,000; 40,000; 60,000; 80,000; 100,000
Learning rate	0,1; 0,01; 0,001
# of hidden nodes	12; 24; 36
# of gateways	500
Mobility scenarios	Static; semi-mobile; fully-mobile
Scenarios	1. Services X Mobility X Learning Rate X Hidden Nodes
Replication	10 rounds each experiment
Hops limit (See Appendix A)	5
Distance (See Appendix A)	100m

requires 24,000, and the domain-based 12,000. Results from this study address the questions **RQ 1**, and **RQ 2** in Section 1.2.4. The inclusion of urban context improves the service discovery efficiency in smart cities scenarios. It is complemented by adaptive properties that respond to city changes and keeps more efficient service discovery over time.

5.2.6 Periodic Events Study

This study measures the discovery efficiency of each approach according to the metrics defined in the Section 5.1.1 in the presence of periodic events. The study defines two experiments: the first one to explore the behaviour of *uDiscovery*2 with different parameters in the model that manages periodic events (Section 3.4.2), and the second one to compare the baselines behaviour in the presence of periodic events.

Table 5.6 introduces the design of the first experiment which simulates 30 periods of 6 hours in a city to evaluate if *uDiscovery* is able to manage periodic events by learning from requests patterns. Each period has two simulated peak hours when a consumer requests services relating to the same domain in each period. The first peak hour starts at hour 2 and ends at hour 3 in each period. The second peak hour starts at hour 4 and ends at hour 5 in each period. The consumer requests services randomly according to its location between peak hours. The experiments parameters are the number of services, mobility scenarios, number of hidden nodes in the neural network, and the learning rate of the model (i.e., α), which are combined to determine the influence of each variable in the service discovery efficiency. Each experiment is replicated 10 times. *uDiscovery*2 uses Algorithm 7 in Section 3.4.2

Table 5.7: Experiments Parameters for Approaches' Periodic Events Management.

Length of time for service discovery	30 periods of 6 hours
Events duration	1 hour
Events start time	First event: Hour 2 each period Second event: Hour 4 each period
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	500
Mobility scenarios	Static; semi-mobile; fully-mobile
Scenarios	1. Services X Mobility
Replication	10 rounds each experiment
Hops limit (See Appendix A)	- 5 for location-based and uDiscovery - 3 for domain-based
Distance (See Appendix A)	- 100 for location-based and uDiscovery
Number of Domains (See Appendix A)	- 5 for domain-based

to monitor discovery performance and try to learn the city patterns from periodic events. *uDiscovery*2 integrates the DL4J[2] library with Simonstrator to use DL4J's implementation of the DQN algorithm. Additional *uDiscovery′s* parameters are selected according to the Appendix A.

Table 5.7 illustrates the design of the second experiment, which compares the performance of different approaches when managing periodic events. This simulates 30 periods of 6 hours in the city as the previous experiment, to compare *uDiscovery′s* performance against other approaches under the same conditions. The experiments' parameters are the number of services, and mobility scenarios, which are combined in to determine the influence of each variable in the service discovery efficiency. Each experiment is replicated 10 times and sets the number of hidden nodes and learning rage for *uDiscovery*2 according to the results in the previous experiment. The rest of approaches' parameters are selected according to the Appendix A.

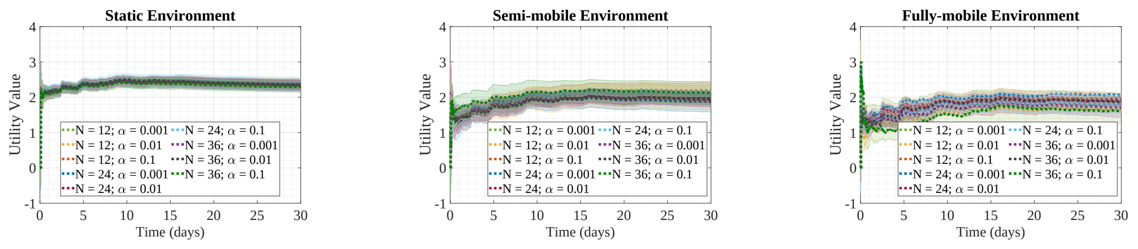
5.2.6.1 Periodic Events Study Results

Figure 5.13 shows *uDiscovery*2*′s* behaviour with different hidden nodes in the neural network, and different learning rates in the different scenarios. As illustrated, the number of hidden nodes and the learning rate do not have a considerable impact on the utility function. This curve follows the same pattern for different configurations in all the cases. The static scenarios show a decreasing trend in the utility value after 10 periods. The semi-mobile and fully-

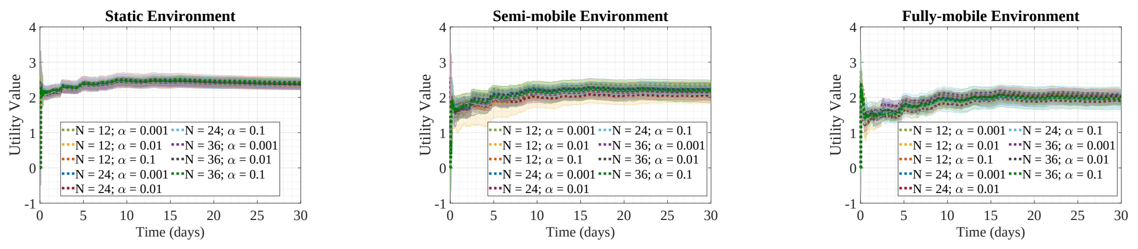
[2]DL4J - `https://deeplearning4j.org/`

mobile scenarios show a constant value after 15 periods. The difference between static and mobile scenarios is because the negative impact of the decreasing rate of solved requests is more notable in static scenarios, where the approach starts with a better utility value. Moreover, the system may ask for services from different gateways each time because gateways may be mobile. However, there is fluctuation in the utility value over time in all cases, and after a while the trend of the curve flattens and starts to be negative in some cases. There is no continuous improvement in the utility value in any of the cases because the algorithm does not make the same decision under the same conditions. A larger fluctuation behaviour is expected as it represents the learning phase when $uDiscovery2$ makes different decisions to evaluate rewards. Then, a constant improvement is expected as $uDiscovery2$ should make better decisions based on the historical rewards and continue learning from the environment. The observed utility curves mean that $uDiscovery2$ does not have a good, current knowledge of the environment, despite the neural network parameters. This limitation is mainly caused by the complexity of the system that the self-adaptive algorithm tries to learn (i.e., a city). Consumers do not request the same services at the same time of the day, and such randomness cannot be fully-learned by the proposed approach. One alternative is to remove such randomness and set a static sequence of requests which is repeated every period. However, this alternative does not reflect the real world. Further research is needed to explore reinforcement learning algorithms in a deeper way, or alternative approaches to identify emergent behaviours in smart cities, which enable models to address such complex environments.

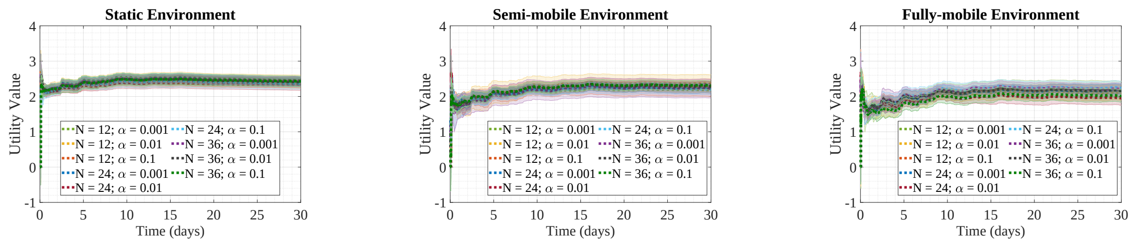
Figure 5.14 compares how different approaches perform when there are periodic events. The neural network in $uDiscovery2$ is configured with 12 nodes and $\alpha = 0.1$. $uDiscovery2$ has the best utility values, which vary from 1.93 to 2.48, followed by $uDiscovery1$ with an utility value that varies from 1.09 to 2.16. The utility value of the location-based approach varies from 0.4 to 1.58, and the domain-based utility value varies from 0.63 to 1.03. The utility curve increases its value in the first periods for $uDiscovery2$, but after a while it is maintained constant or decreases. $uDisovery2$ does not improve the efficiency over time, although it has a better utility than all other approaches. Periodic events have significant effect on the performance of the baselines. The location and domain-based approaches have a decreasing trend for most cases. The urban-based approach has some scenarios where the value decreases but it also manages to keep the utility constant in most cases. $uDiscovery2$ is still better than all the baselines, although it is not able to fully learn the city environment. The advantage emerges because the self-adaptive approach uses the urban-context to drive



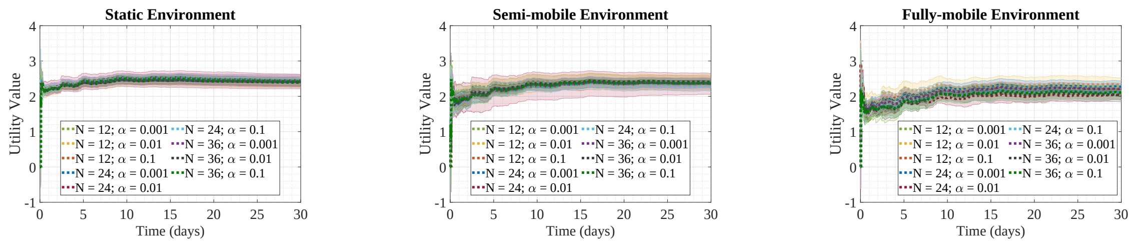
(a) 20,000 Services



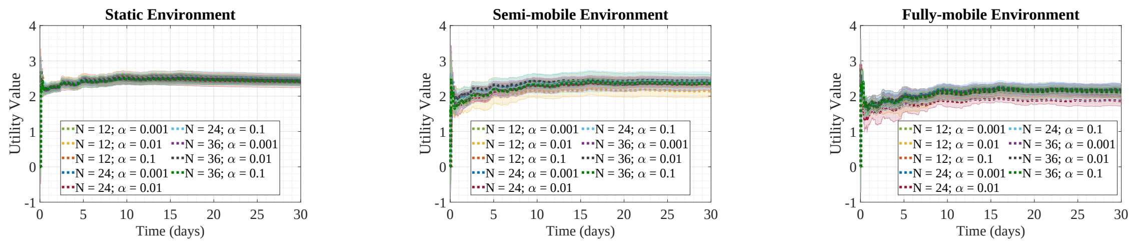
(b) 40,000 Services



(c) 60,000 Services

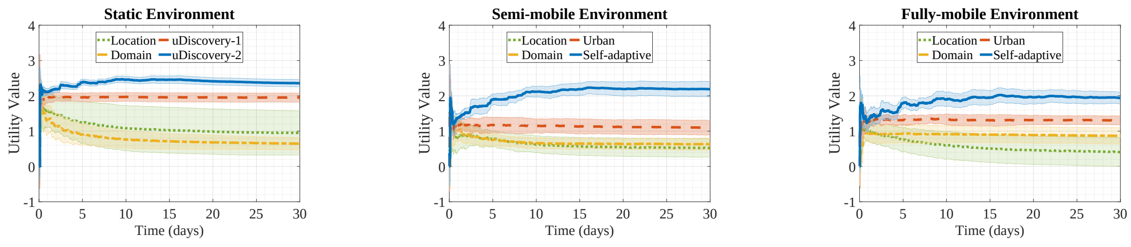


(d) 80,000 Services

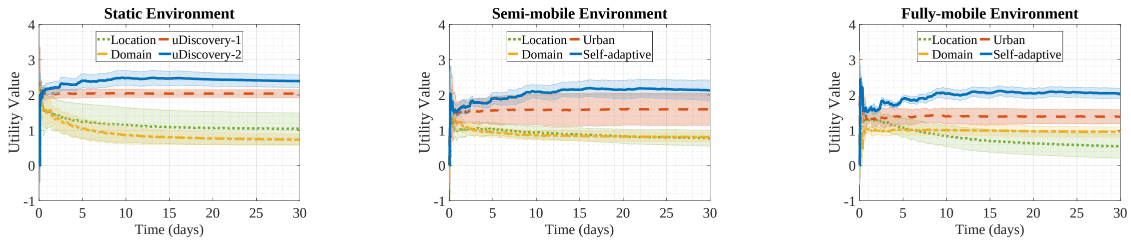


(e) 100,000 Services

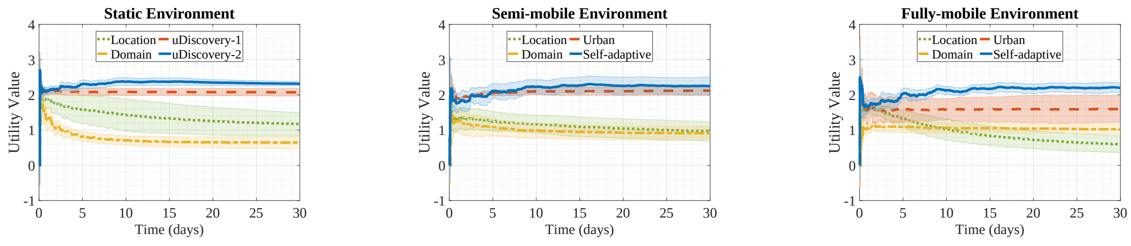
Figure 5.13: Utility Function: Periodic Events Study with Different Neural Network Configurations.



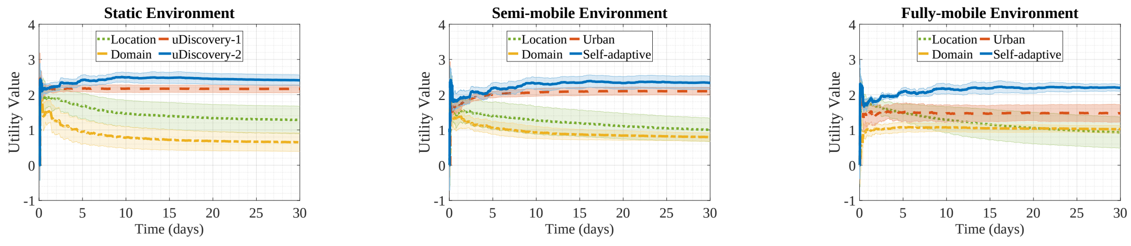
(a) 20,000 Services



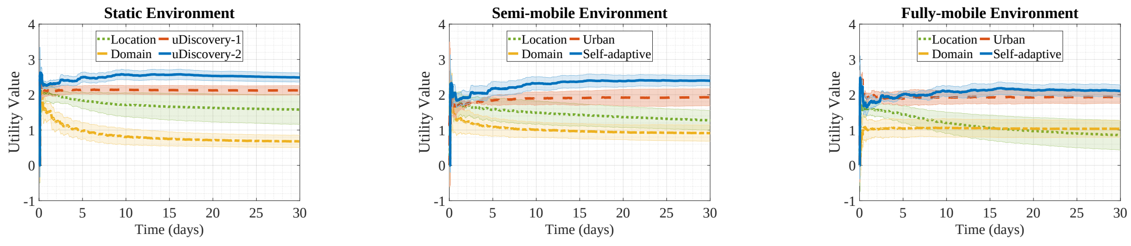
(b) 40,000 Services



(c) 60,000 Services

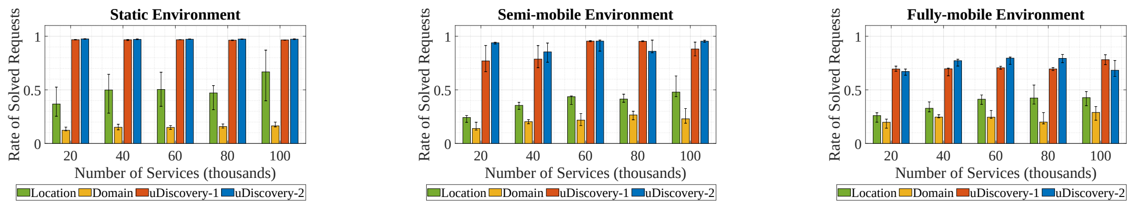


(d) 80,000 Services

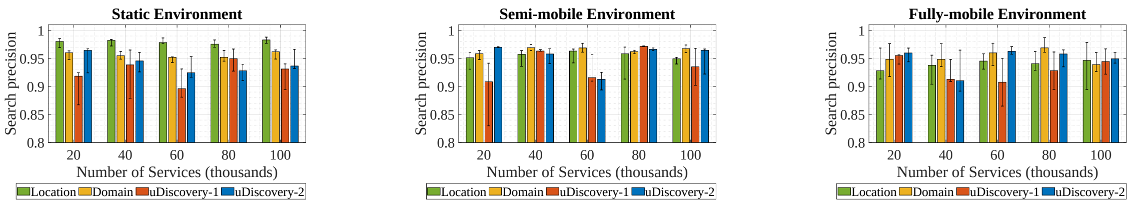


(e) 100,000 Services

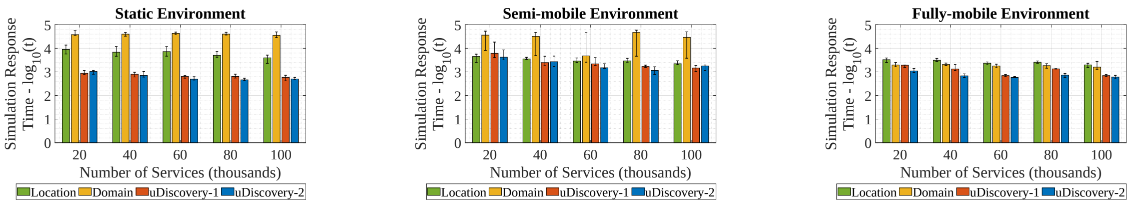
Figure 5.14: Utility Function - Periodic Events Study with different approaches.



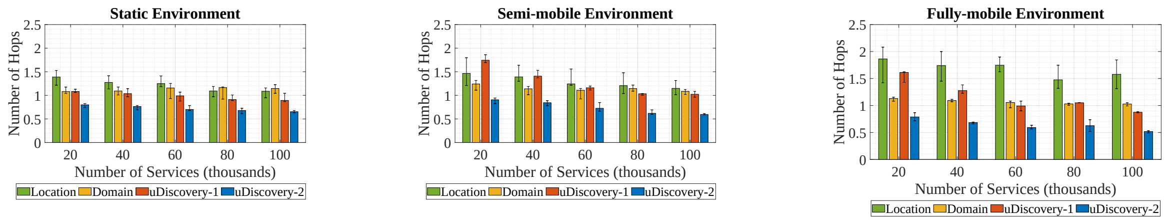
(a) Solved Requests



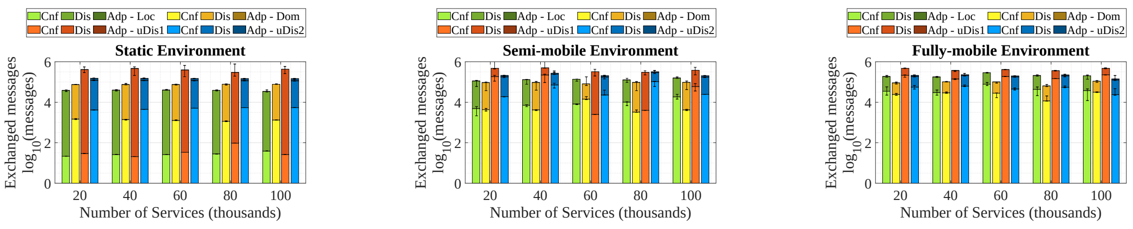
(b) Search Precision



(c) Response Time



(d) Number of Hops



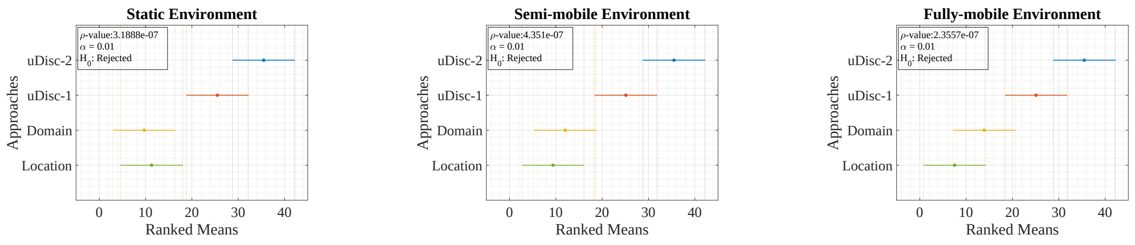
(e) Exchanged Messages

Figure 5.15: Service Discovery Metrics - Periodic Events Study with different approaches.

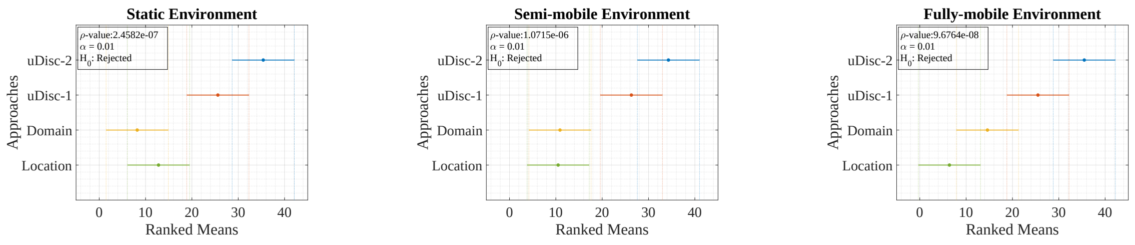
the discovery, and it manages to make the right decision some times (i.e., moving the right services at the right time).

Figure 5.15 presents the service discovery metrics for different approaches when there are periodic events. *uDiscovery*1 and *uDiscovery*2 have a similar **rate of solved requests** in different mobility scenarios (Figure 5.15a). This rate is higher in the static environment (i.e., close to 1) because gateways have more information to forward requests to the relevant gateways. This rate is impacted in both *uDiscovery*1 and *uDiscovery*2 by mobile gateways, because the information about other gateways is smaller and less accurate in these environments (e.g., the rate varies from 0.7 to 0.8 in the fully-mobile environment). The rate of solved requests of the location-based approach is under 0.7 in all cases, but it improves in the presence of more services. The domain-based approach has the worst rate of solved requests as in previous studies (i.e., under 0.4 in all cases). The **search precision** for different approaches follows a similar behaviour as in the performance studies of unforeseen and scheduled events. The location and domain-based approaches have an advantage in the static environment, but none of the approaches can be considered better in semi-mobile and fully-mobile scenarios. *uDiscovery*1 and *uDiscovery*2 offer a lower search precision in some cases, but they still provide a higher rate of solved requests. The simulated **response time** (Figure 5.15c) and **number of hops** (Figure 5.15d) have an interesting outcome. *uDiscovery*1 and *uDiscovery*2 have a similar latency in all scenarios, though *uDiscovery*2 needs more hops to discover services compared against previous studies. This means that *uDiscovery*2 solves less requests locally, the adaptation processes fail to move the required services, and *uDiscovery*2 uses the urban context to forward requests to other gateways as *uDiscovery*1 does.

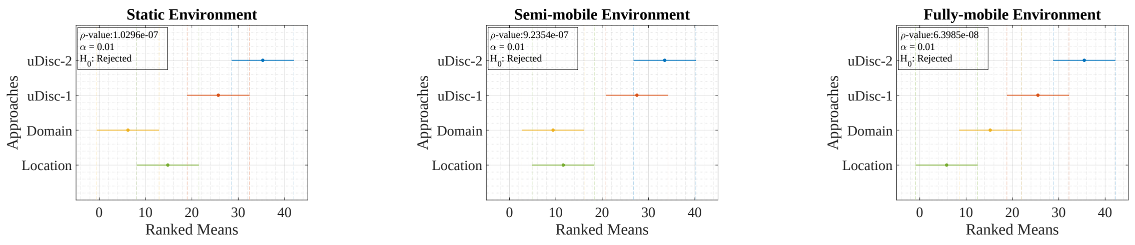
The median of exchanged messages follows the same pattern as in previous studies (Figure 5.15e). *uDiscovery*1 and *uDiscovery*2 exchange more messages than the location and domain-based approach. This is the cost of a more efficient service discovery reflected in a larger rate of solved requests. Location-based approach exchanges around 38,000 messages in the static scenario, 120,000 in the semi-mobile, and more than 200,000 in the fully mobile. The domain-based approach has the best efficiency in terms of network usage because of the lower limit of hops, as in previous studies. This approach exchanges from around 77,000 messages in the static environment, to around 90,000 in the fully-mobile. *uDiscovery*1 exchanges more than 300,000 messages in all the cases, and *uDiscovery*2 around 150,000 in the best case and 250,000 in the worst. The advantage for *uDiscovery*2 is because some times the adaptive manager makes the right decision and moves services to solve requests locally.



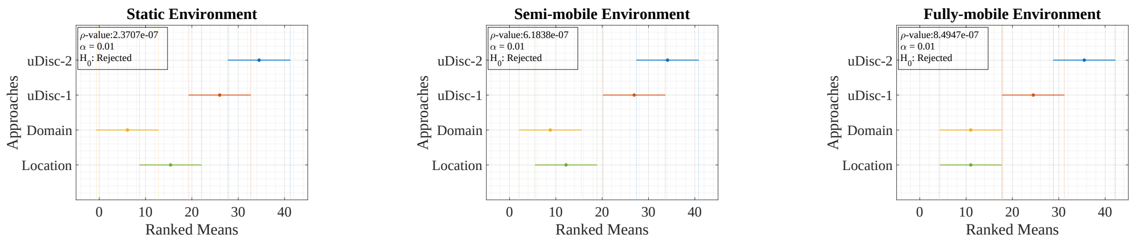
(a) 20,000 Services



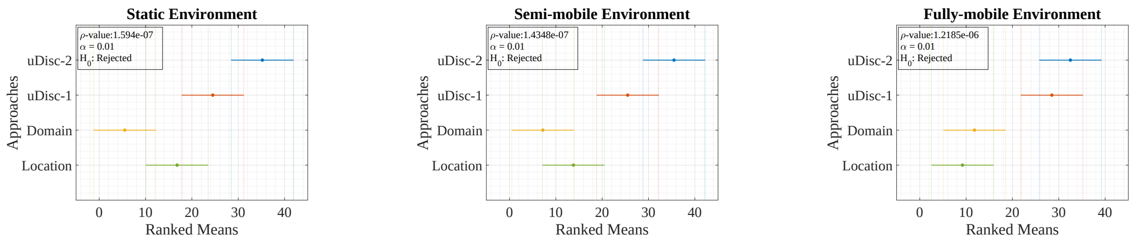
(b) 40,000 Services



(c) 60,000 Services



(d) 80,000 Services



(e) 100,000 Services

Figure 5.16: Statistical Test Periodic Events Study: Multiple Comparison of Ranked Means.

Statistical Analysis

A Kruskal-Wallis test was performed on the utility value of different approaches in this study. This test is used to identify whether the performance of evaluated approaches is statistically significantly different. The null hypothesis of this test specifies that samples of the utility value of each approach are subsets from the same population (i.e., $H_0 : (a, b, c, ..., n) \subseteq p$) with confidence interval $\alpha = 0.01$. The projected lines from each ranked mean shows whether two approaches are statistically significantly different, as in previous studies.

Figure 5.16 presents the results from the test in this study. H_0 is rejected in all the cases which means that there are statistically significant differences between approaches performance. *uDiscovery*2 is statistically different from the location and domain-based approaches in all cases. *uDiscovery*1 is statistically different from the location and domain-based approaches in 6 out of 15 cases. *uDiscovery*1 and *uDiscovery*2 have a more similar performance than in previous studies, as their ranked means have bigger overlaps. This is because *uDiscovery*2 does not move services properly and uses urban context information to forward requests to relevant gateways, making its behaviour similar to $uDiscovery1's$.

Discussion

This study evaluated *uDiscovery*1 and *uDiscovery*2 in the presence of periodic events in a simulated city environment. Results show that *uDiscovery*2 does not learn from the environment, and cannot handle periodic events (i.e., the discovery efficiency does not improve over time). It was expected at least a more random behaviour, which implies that the adaptive manager was learning, but results show a flatter curve after few periods instead. This behaviour is observed in every experiment despite the number of hidden nodes and learning rates, which influence the DQN algorithm used by *uDiscovery*. This limitation can be caused by the complexity of city environments where services are requested randomly, but also because of the limitation of current learning techniques. Further research is needed to explore RL algorithms in a deeper fashion, propose possible extensions, and experiment with variable environment complexity to determine to what extent current learning methods can handle cities scenarios.

However, *uDiscovery*2 still has better performance than the location and domain-based approaches (e.g., rate of solved requests close to 1). This is because *uDiscovery*2 uses urban context to forward requests where they are most likely to be solved, and sometimes makes the right decision to move services. *uDiscovery*1 and *uDiscovery* solve more requests than baselines because of the use of urban context, but fail to manage periodic city events. The

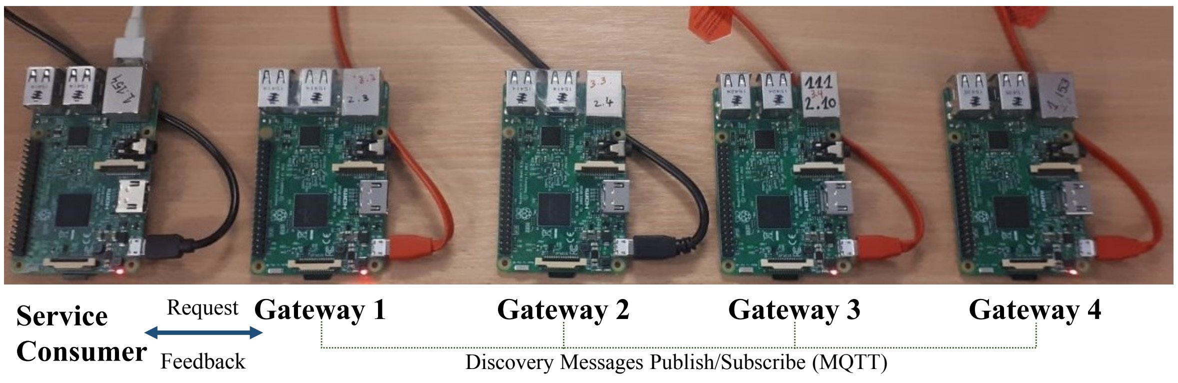


Figure 5.17: IoT Test bed.

cost of *uDiscovery* performance is again the number of exchanged messages. The domain-based approach uses the network in a more efficient way as it needs much less messages than others approaches (i.e., around 100,000 in the worst case). However, it is not able to offer an efficient discovery according to the other metrics (e.g., rate of solved requests). Results from this study address the questions **RQ 1**, and **RQ 2** in the Section 1.2.4.

5.3 Prototype-based Evaluation

This section presents the prototype-based study that evaluates to what extent *uDiscovery* improves the service discovery efficiency, and minimises human input when responding to complex consumer's requirements **(RQ 3)**. This section compares *uDiscovery's heuristic planner* against baseline approaches and discusses the results.

5.3.1 Experimental Set-up

A prototype of *uDiscovery's* heuristic planner (Section 3.4.3), and baselines are implemented for this evaluation. These prototypes are implemented in Python 3.5 and deployed in an IoT testbed with 5 Raspberry Pi3 enabling the evaluation of *uDiscovery* in resource constraint devices, which run Raspbian, have 1GB of RAM and an SD card with 16GB. Each board sends messages to other boards using a MQTT broker through a WiFi MANET. One of the boards is the consumer and sends requests and feedback to the other boards which act as IoT gateways (Figure 5.17). Each board has a Mongo 2.4 database where it stores services, requests data, and consumers' feedback.

5.3.2 Baseline Approaches

The prototype-based study evaluates and compares three different approaches to search for services and support service composition. The approaches are:

Table 5.8: Experiments Design Composition Support Study.

Experiment	The consumer requests a service.
Experiments' Parameters	- Number of Services: 2, 4, 6, 8, and 10 thousand - Plan Length: 1, 2, 3, 4, and 5
Parameters Combinations	1. Services X Plan Length
Replication	100 requests in each experiment

- *Classic Backward Planning Approach* : This baseline uses a backward planning algorithm to discovery services that constitute service compositions and is based in the discovery components of the work of Chen et al. [Chen et al., 2016].
- *Conversation based Approach* : This baseline uses service conversations to support service composition and is based on the discovery components of the work of Uribieta et al. [Urbieta et al., 2017].
- *uDiscovery Planner* : This prototype implements the heuristic service planner proposed in the Section 3.4.3 of this thesis. Specific parameters of this model were defined by the study in Appendix A for this comparison. The $feedbackThreshold$ is 1.0, the $functionalThreshold$ is 1.0, and the K-value is 5.

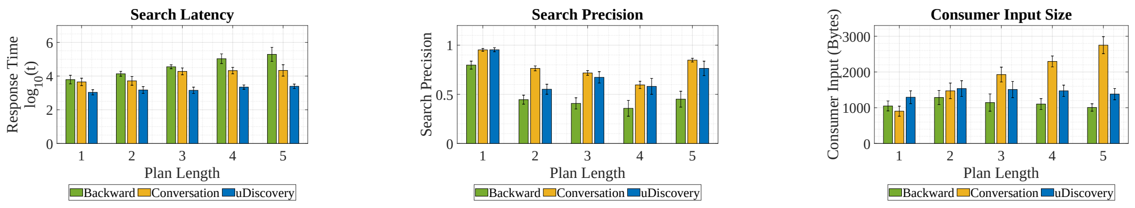
5.3.3 Prototype-based Study

This study measures the search efficiency of each approach (i.e., backward planning, conversation-based, and *uDiscovery planner*) with variable number of services and plan length according to the metrics defined in the Section 5.1.2. Table 5.8 presents the experiments design of this study. The consumer requests a service in each experiment. The experiment's parameters are number of services, and plan length, which are combined to evaluate and compare approaches' performance. Each experiment (i.e., a service request) is replicated 100 times.

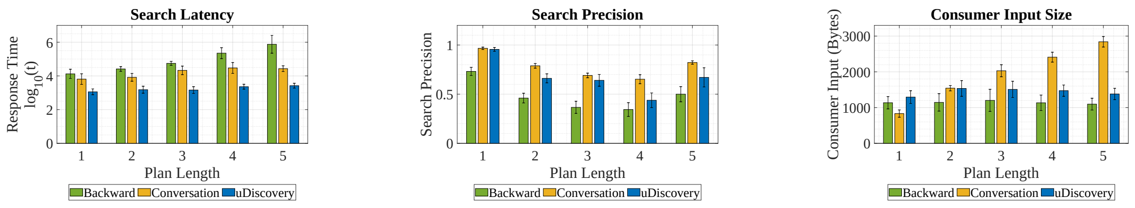
5.3.3.1 Prototype-based Study Results

Figure 5.18 illustrates this study results. Each row in the figure corresponds to a different number of services, and each column corresponds to a metric (i.e., search latency, precision, and input size). Each sub-figure plots metric against the plan length variable.

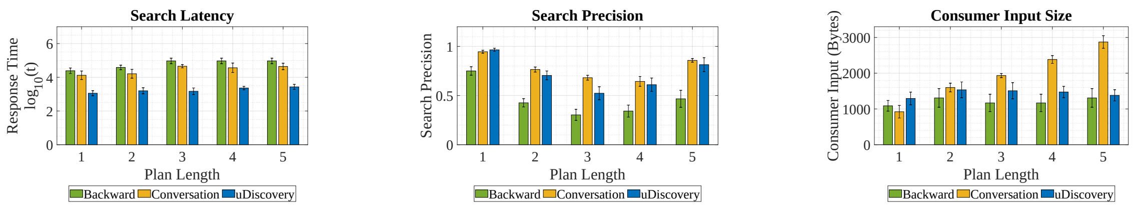
Search latency measures the response time in milliseconds. Figures in the first column show the median response time in logarithmic scale to facilitate their analysis. The backward planning approach has the worst latency in all cases. The response time grows with the plan length. The response time is 7 seconds for plan length 1, 14 seconds for length 2, 36



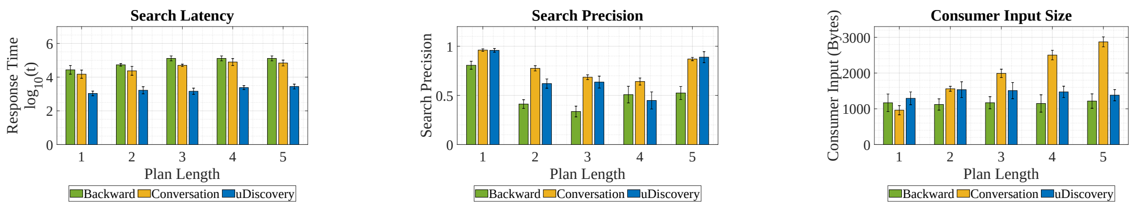
(a) 2,000 Services



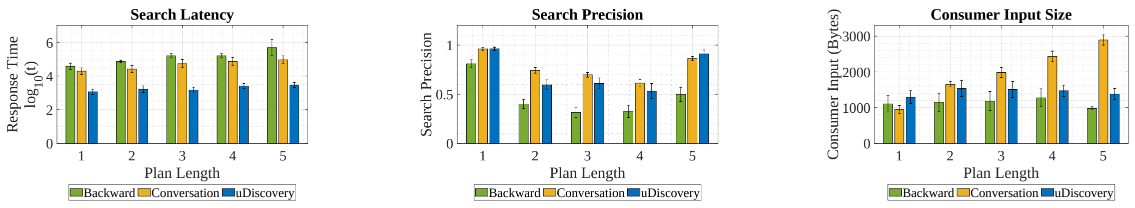
(b) 4,000 Services



(c) 6,000 Services



(d) 8,000 Services



(e) 10,000 Services

Figure 5.18: Prototype-based Study Results.

seconds for length 3, 2.14 minutes for length 4 and 4.61 minutes for plan length 5 when there are 2,000 services. The number of services also affects the response time in the backward planning approach. The response time is 40 seconds for length 1, 1.22 minutes for length 2, 2.72 minutes for length 3, 6.17 minutes for length 4, and 12 minutes for length 5 when there are 10,000 services. This high latency is because the backward planning approach performs and exhaustive search through all the services in the registry, exploring all the possible combinations. The conversation-based approach has the next higher latency, which is also affected by the plan length. The response time is 4.9 seconds for plan length 1, 6.5 seconds for length 2, 20 seconds for length 3, 23 seconds for length 4 and 28 seconds for length 5 when there are 2,000 services. The number of services also impacts the latency of the conversation-based approach because it matches all services in the repository. The response time is 20 seconds for plan length 1, 27 seconds for length 2, 1.02 minutes for length 3, 1.49 minutes for length 4 and 1.85 minutes for length 5 and 10,000 services.

uDiscovery has the lowest response time because of the progressive search that limits the number of explored services according to the functional ranking and the consumer's feedback (Section 3.4.3). Neither the plan length or the number of services have the same impact in *uDiscovery′s* performance. The response time is 1.08 seconds for plan length 1, 1.51 seconds for length 2, 1.48 seconds for length 3, 2.18 seconds with length 4, and 2.5 seconds for length 5 when there are 2,000 services. The response time is 1.12 seconds for plan length 1, 1.6 seconds for length 2, 1.55 seconds for length 3, 2.5 seconds for length 4 and 2.75 for length 5 when there are 6,000 services. And, the response time is 1.16 seconds for plan length 1, 1.59 seconds for length 2, 1.52 seconds for length 3, 2.55 seconds for length 4 and 2.95 seconds for length 5 and 10,000 services.

The conversation-based approach offers the best precision in most of the cases. This precision is different for each plan length because of the data set variability. Requests with plan length 3 and 4 get the lowest search precision (i.e., around 0.7 for length 3, and 0.6 for length 4). The number of services does not affect the accuracy of the conversation-based approach. It varies from 0.59 with 4,000 services and plan length 4 to 0.9668 with 6,000 services and plan length 1. *uDiscovery′s* precision is close to the conversation-based approach in most of the cases, and slightly superior in some of them. The difference between *uDiscovery* and the conversation-based approach is smaller when there are more services in the repository because *uDiscovery* explores more services and selects the most promising based on the functional ranking and the feedback information (Section 3.4.3). The search precision is 0.9533 for plan length 1, 0.5516 for plan length 2, 0.6729 for plan length 3, 0.581 for plan length 4, and 0.76

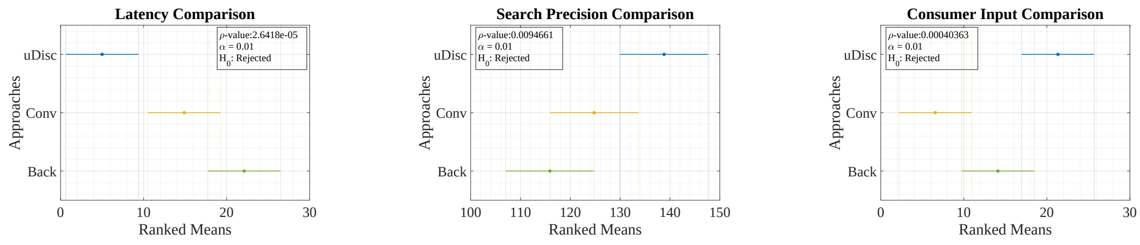
for plan length 5 with 2,000 services. The precision is 0.9626 for plan length 1, 0.5956 for plan length 2, 0.6102 for plan length 3, 0.532 for plan length 4, and 0.9103 for plan length 5 with 10,000 services. The backward planning approach has the worst precision because it has limited information to drive the service search. Search precision varies from 0.3154 with 10,000 services and plan length 3 to 0.8095 with 10,000 services and plan length 1.

The backward planning approach needs less input from consumers than *uDiscovery* and the conversation-based approach. It varies from 979 bytes to 1309 bytes regardless of the plan length and the number of services in the registry. *uDiscovery* also keeps a constant number of bytes (i.e., around 1500), as input from consumers, regardless of the plan length and the number of services. The input size is larger in *uDiscovery* compared to the backward planning approach because of the feedback provided by the consumer after the search process. The conversation-based approach needs more input from consumers as the plan length increases. The input size is lower than the size of backward planning and *uDiscovery* inputs when the plan length is 1 (i.e., around 900 bytes). This size increases to around 1600 bytes for plan length 2, 1900 bytes for plan length 3, 2433 for plan length 4, and 2894 for plan length 5. The number of services does not affect the input required by the conversation-based approach.

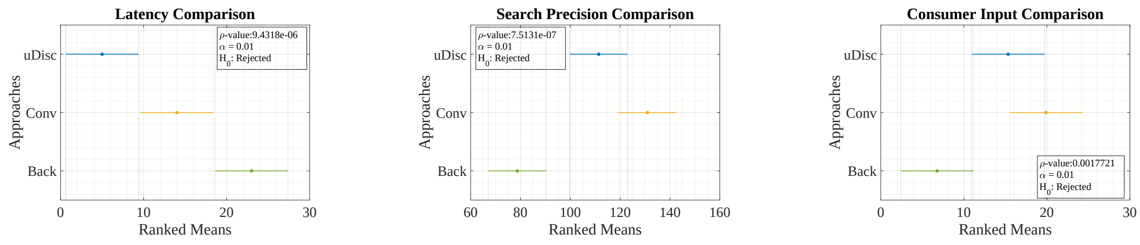
Statistical Analysis

A Kruskal-Wallis test was performed on the different metrics with 10,000 services in the distributed repository. This test is used to identify statistically significant differences in approaches performance with regard the search efficiency with variable plan length. The null hypothesis specifies that samples from results of each metric for each approach are subsets from the same population (i.e., $H_0 : (a, b, c, ..., n) \subseteq p$) with confidence interval $\alpha = 0.01$.

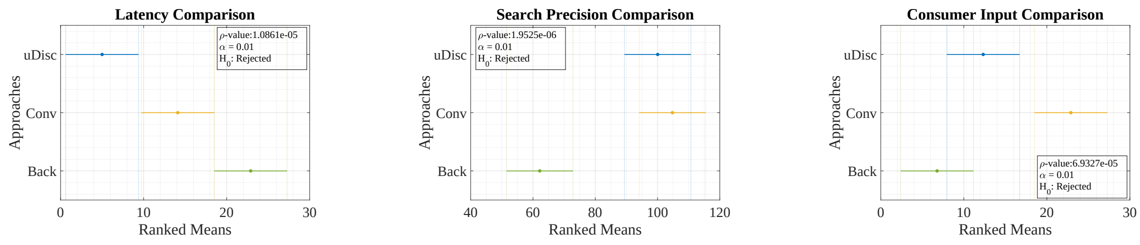
Figure 5.19 presents the results from the test in this study. Each row in the figure corresponds to the plan length, and each column to each metric (i.e., search latency, search precision, and consumer input size). H_0 is rejected in all cases for the comparison of approaches latency, which means that there are statistically significant differences between approaches performance. *uDiscovery*2 is statistically different from the conversation-based and the backward planning approaches in all cases as it always offers the lowest latency. The difference between *uDiscovery* and the next approach in the rank (i.e., the conversation-based approaches) increases with the plan length, thanks to the progressive search that limits the explored services and combinations to the most promising ones. There are significant differences between



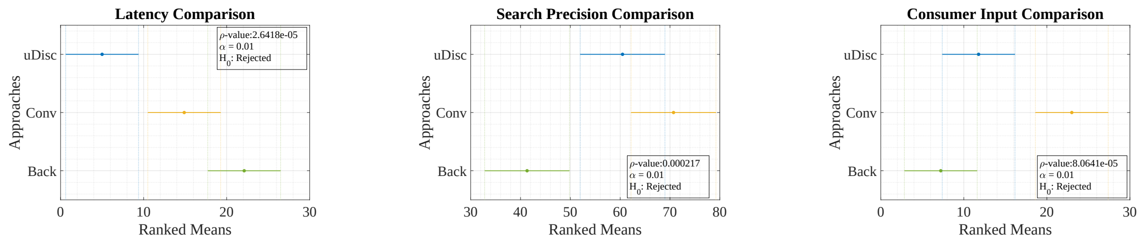
(a) Plan Length 1



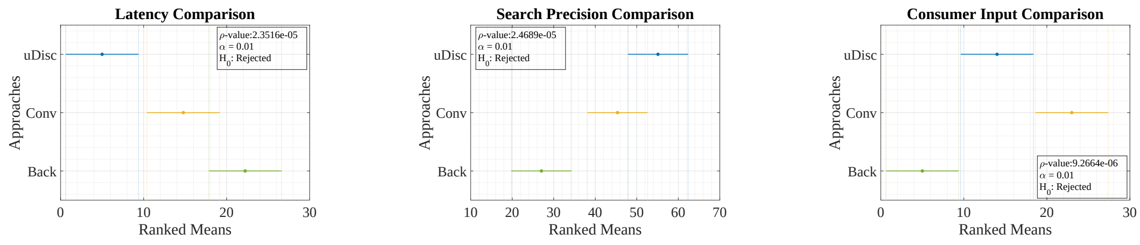
(b) Plan Length 2



(c) Plan Length 3



(d) Plan Length 4



(e) Plan Length 5

Figure 5.19: Statistical Test Prototype-based Study: Multiple Comparison of Ranked Means.

approaches with regard to the search precision in all cases (i.e., H_0 is rejected). Such differences are mainly because the search accuracy of the backward planning approach is too low compared to *uDiscovery* and the conversation-based approach. $uDiscovery's$ search precision is not statistically significantly different from the conversation-based approach in any of the cases. This means that *uDiscovery* manages to offer similar search precision to the conversation-based approach because of the inclusion of the consumer's feedback and the functional ranking.

H_0 is also rejected in all cases for the comparison of approaches input size, which means that there are statistically significant differences between approaches inputs size. $uDiscovery2$ and the backward planning algorithm do not have differences in most of the cases, with exception of the scenario for plan length 5. The conversation-based approach is statistically different from *uDiscovery* because the size of requests in the conversation-based approach is less than the size of requests in *uDiscovery*. This is because requests in the conversation-based approach only specify one task (i.e., plan length 1). Then, *uDiscovery* is not statistically significantly different from the $conversation-based$ approach for plan length 2, because their requests have a similar size. This is because requests in conversation-based specify information about two tasks (i.e., plan length 2). *uDiscovery* is statistically significantly different from the conversation-based approach when the plan length is greater or equal to 3 because the size of requests in *uDiscovery* is less than the size of requests in the conversation-based approach. This is because requests in conversation-based specify more information as the number of tasks increases. The ranked mean of the conversation-based approach moves from the left to the right when the plan length increases, which illustrates the linear relation between the input size and the plan length. The ranked means of the backward planning approach and *uDiscovery* do not move, which illustrates that the required input keeps constant regardless of the plan length, in both approaches. *uDiscovery* offers the lowest search latency, with a similar search precision and less consumer's input compared to the conversation-based approach, according to this analysis.

Discussion

This study evaluated the *uDiscovery* service planner in an IoT test bed with variable number of services and plan length. Results show that *uDiscovery* searches for services with the lowest latency (i.e., close to 3 seconds for length plan 5 and 10,000 services), and equals, or in some cases overcomes, the accuracy of the conversation-based approach by requiring less input from consumers (i.e., around 1,000 bytes despite the plan length). *uDiscovery* improves the search efficiency based on a progressive search that limits the explored services and combinations to the most promising ones using consumers' feedback and the functional ranking. These results address the question **RQ 3** in the Section 1.2.4.

5.4 Evaluation Summary

This chapter presented *uDiscovery's* evaluation, which includes a set of simulations and a prototype-based study. The research questions and objectives drive the evaluation to measure to what extent *uDiscovery* improves the efficiency of the discovery of services in smart city environments.

Simulations studies evaluate *uDiscovery's* performance in smart city simulated environments with regard to the utility function defined in the Section 3.4.2, the rate of solved requests, search precision, simulated response time, number of hops to discovery a service, and the exchanged messages. *uDiscovery* is compared against two approaches: one that drives the service discovery based on the services location, and another one that drives the service discovery based on services domains. These studies address the questions **RQ 1**, and **RQ 2** in Section 1.2.4. Their results show that the inclusion of urban context improves the service discovery efficiency in smart cities scenarios. It is complemented by adaptive properties that respond to city changes and keeps more efficient service discovery over time. *uDiscovery* solves more requests with high search precision, lower latency and less number of hops in scenarios with variable number of services, and gateways. *uDiscovery* keeps a better performance over time in the presence of unforeseen, scheduled, and periodic events. However, it fails to learn from periodic events and proactively adapt services organisation because of the environment complexity. Further research is needed to handle complex environments such as smart cities where learning approaches need to consider a large number of features.

A prototype-based study evaluates the performance of *uDiscovery* on an actual IoT test bed. It measures the search latency, precision, and input size of *uDiscovery* for requests that require service composition support. This study compares *uDiscovery* heuristic planner against a backward planning, and a conversation-based approach. Results show that *uDiscovery* offers the lowest latency, and equals the accuracy of the conversation-based approach by requiring less input from consumers. *uDiscovery* offers a more efficient service search than the backward planning approach, but requires more information from consumers in the form of feedback. This study addresses the question **RQ 3** in the Section 1.2.4.

Chapter 6

Conclusion

This thesis investigated the service discovery process in large and dynamic smart city environments. It identified the limitations of current service discovery approaches, and proposed *uDiscovery*, a distributed urban-centric model for service discovery in smart cities. *uDiscovery* organises and searches for services based on urban-context, adapts such service organisation to respond to city events, and drives a progressive search based on consumers' feedback. This thesis evaluated *uDiscovery* using both simulated city environments, and an IoT test bed under varying number of services, and different mobility scenarios. Results were compared with existing service discovery approaches. This rest of this chapter summarises this thesis contributions, limitations, and future work.

6.1 Thesis Summary

Introduction Chapter 1 described the research challenges that motivate this thesis with regard to the service discovery in smart city environments. This chapter analysed existing solutions, identified research gaps, and defined the research questions of this thesis. The chapter identified that current service discovery approaches have performance issues discovering services in large and dynamic IoT environments. Thereafter, this chapter proposed a hypothesis to address these issues, which states that the inclusion of city and citizens' context can improve the discovery efficiency in smart cities. The chapter also presented the thesis objectives, and the approach to investigate the proposed hypothesis. Finally, this chapter outlined the contributions, and limitations of this thesis.

State of the Art Chapter 2 analysed how the state of the art in service discovery addresses the challenges of large and dynamic smart city environments. This chapter studied how current solutions organise services' information, manage consumers requests, perform service planning, and handle dynamic environments. Three gaps are highlighted from this analysis. Current approaches organise services according to their attributes in static structures. These structures do not provide enough information to support efficient service discovery in large environments. Current planning approaches either require high human input or might present performance issues with regard to search latency and accuracy. And, current self-adaptive approaches respond to devices or network changes, but they do not consider changes in the IoT environments where they work, causing outdated architectures.

Design Chapter 3 defined the design objectives of this thesis based on the research questions, and hypothesis outlined in Chapters 1 and 2. This chapter also introduced the design decisions of *uDiscovery*, and presented its detailed design. *uDiscovery* puts the right service in the right place in preparation for discovery, and forwards consumers' requests where they are most likely to be solved using urban context. This context is extracted from city places information, which defines the relevance of gateways to manage services and solve requests. *uDiscovery* uses a bio-inspired method on top of the urban-based service organisation to propagate the information in an environment where each gateway has a partial knowledge about the network. *uDiscovery* adapts the service organisation by sensing the impact of city events in the discovery performance, and moving services between gateways. *uDiscovery* searches for services in local repositories using a heuristic planner based on consumers' feedback. This planner drives a progressive search that avoids the exploration of incorrect combinations of services, and explores correct combinations according to how well they meet requests' functional requirements.

Implementation Chapter 4 presented the implementation details of *uDiscovery*. This chapter described *uDiscovery's* architecture where the urban-based service manager organises services information, and manages consumers' requests. The self-adaptive service manager responds to city events by reorganising services information. And, the heuristic service planner performs a progressive search to support composition of services with minimal human input. The chapter illustrated the structure, behaviour, and interactions of *uDiscovery* components to realise the design specified in Chapter 3.

Evaluation Chapter 5 evaluated to what extent *uDiscovery* improves the service discovery efficiency in smart city environments according to the defined research questions (Chapter 1). This evaluation included simulations studies and a prototype-based study. Simulations evaluated *uDiscovery's* performance in smart city simulated environments, and the prototype evaluated *uDiscovery′s* performance in an actual IoT test bed. Results show that *uDiscovery* improves service discovery efficiency in smart city environments compared with baselines, at the cost of more exchanged messages. *uDiscovery* also improves search process efficiency in the IoT test bed, reducing human input. Section 6.2 discusses these results to highlight *uDiscovery′s* contributions and limitations.

6.2 Discussion

uDiscovery improves the service discovery efficiency in simulated smart city environments compared with existing approaches. *uDiscovery* offers a better discovery efficiency at the cost of more messages exchanged between gateways. This section outlines first the contributions of this thesis and then discusses *uDiscovery′s* limitations.

6.2.1 Thesis Contributions

This thesis outlines three contributions to the body of knowledge. The first contribution is the use of urban context to manage services and requests in smart cities. Previous service discovery approaches manage services and requests according to services' attributes in structures that might fail to provide enough information to drive efficient service discovery in smart cities. *uDiscovery* formalises city concepts in a knowledge model that drives a more informed discovery of services. *uDiscovery* distributes service information to gateways that are relevant according to their surrounding places based on the formalised knowledge model. It puts information about the right service in the right place, and enables a smart replication of services in preparation for discovery. *uDiscovery* also uses the urban context to manage service requests, but this is not enough because gateways have partial knowledge about other gateways in the network. *uDiscovery* solves this issue by using a bio-inspired method to forward requests to the most relevant gateways. Section 5.2 reports how well *uDiscovery* works with regard to this contribution in simulated smart city environments. The limitations of this contributions are discussed in the Sections 6.2.2 and 6.2.4.

The second contribution is the adaptation of services' organisation according to city events. Previous research in service discovery for IoT proposes adaptive approaches that react to changes in the "thing" properties (e.g., battery level) or the network topology. These approaches do not consider the changes of the real-world environments with which IoT services interact (e.g., a smart city). *uDiscovery* proposes a novel self-adaptive discovery model for smart cities that offers the right service in the right place, at *the right time*. Service information distribution evolves according to city events which may be unforeseen (e.g., flash flooding), or foreseen (e.g., a cultural event). *uDiscovery* formalises the temporal dimension of the city in the knowledge model and proposes mechanisms to identify and react to city events by exchanging services between IoT gateways. It creates an IoT network where distributed gateways collaborate between each other to improve the discovery efficiency over time. Section 5.2 reports how well *uDiscovery* works with regard to this contribution in simulated smart city environments. The limitations of this contributions are discussed in the Sections 6.2.3 and 6.2.4.

The third contribution is a progressive service search that discovers services that constitute compositions. Previous service planning approaches are not suitable for smart cities because large environments affect their performance, and they either require high level of inputs from consumers, or have search accuracy issues. *uDiscovery* proposes a heuristic service planner that uses consumer feedback to improve search efficiency, minimising human input. Consumer feedback (i.e., historical data from previous searches) determines if a discovered plan was correct or incorrect and is used to improve search accuracy. The latency of the process is reduced using two strategies. First, the model explores only the most promising plans (i.e., search space reduction). Second, the model avoids wasting time on the less promising plans (i.e., progressive search). *uDiscovery* defines the structures to manage consumers' information and the algorithms to include this knowledge in a progressive search. Section 5.3 reports how well this contribution works in a real IoT test bed. The limitations of this contributions are discussed in the Section 6.2.5.

6.2.2 Urban-context Dependency

uDiscovery uses city places information as urban context because this information can be used to infer consumers' needs in a smart city (Section 3.3). *uDiscovery* depends on this urban context to organise services and manage requests. Simulation studies evaluated *uDiscovery* and show that the inclusion of urban context improves the service discovery efficiency compared against baseline approaches. Results from these studies address questions **RQ 1**, and **RQ 2** in Section 1.2.4. However, *uDiscovery* is not suitable for IoT environments that do not provide the required urban information. For example, indoor IoT scenarios such as smart buildings [Wang et al., 2015, Bovet and Hennebert, 2014] or ambient computing environments [Mokhtar et al., 2010, Görgü et al., 2017], and smaller networks of sensors (i.e., WSNs) [Butt et al., 2013, Perera et al., 2014b, Fredj et al., 2014]. Nonetheless, it is important to note that the *heuristic planning* model can be used in domains different to smart cities. It is a general planner that creates services plans based on a semantic matchmaking of services interfaces (i.e., inputs and outputs parameters). Moreover, the proposed city knowledge model is independent from the service discovery domain, and can be used in different service-oriented processes (e.g., service placement, or service execution) to support smart city applications.

6.2.3 Periodic Events Management

The simulation studies show that the *uDiscovery's* self-adaptive model improves service discovery efficiency over time and responds to city unforeseen and scheduled events (Section 5.2). Results from these studies address the question **RQ 2** in Section 1.2.4. However, results also show that *uDiscovery* cannot adapt services organisations in a proactive fashion. The proposed method fails to identify city periodic events as it cannot identify patterns in the consumer's requests, and make the right decision accordingly. This limitation can be caused by the complexity of city environments where services are requested randomly, but also because of the limitation of current learning techniques. Further research is needed to explore learning algorithms in a deeper fashion, propose possible extensions, and experiment with variable environment complexity to determine to what extent current learning methods can handle cities scenarios.

6.2.4 Network Efficiency

The simulation studies show that *uDiscovery* improves service discovery efficiency in terms of rate of solved requests, simulated response time, and number of hops to discover services by maintaining a high search accuracy (Section 5.2). *uDiscovery* achieves this performance as a result of the urban-based structure of gateways that organises services (Section 3.4.1.3), the bio-inspired mechanism that manages requests (Section 3.4.1.4), and the adaptation properties that respond to city events (Section 3.4.2). The simulation studies also show that the cost of this performance is a higher number of exchanged messages between gateways in the network. The number of configuration messages is higher because of the initialisation and maintenance processes that spread gateways' information in the network, and manage gateways' mobility. The number of discovery messages is also higher because each gateway has more information about other gateways than in baselines, so each gateway has more destinations to send messages. It is important to note that the adaptation processes have the most significant impact in the network efficiency. *uDiscovery* sends much more messages than others when implementing adaptive properties (Section 5.2).

6.2.5 Interface-based Planning

uDiscovery depends on a goal-driven approach to discover services that constitutes compositions. This approach matches services inputs and outputs to combine services that meet consumers' requests, which are also described by inputs and outputs as functional requirements (i.e., interface-based). The prototype-based study evaluated the heuristic planner that *uDiscovery* uses to search for services in local repositories (Section 3.4.3). This study shows that *uDiscovery* improves the local search efficiency, despite the plan length, by driving a progressive search based on consumers' feedback. Results from this study address the question **RQ 3** in Section 1.2.4. However, interface-based approaches implement semantic matching, which relies on central ontologies that annotate inputs and outputs. Such ontologies and annotations require human experts to define them. This constraint might be an issue in large IoT environments where heterogeneous providers might not agree in one single service representation, and language description. *uDiscovery's* heuristic planner cannot discover services that do not follow the service representation illustrated in Figure 3.1a.

6.2.6 Concurrent Consumers' Requests

The simulation studies consider scalability from the perspective of large number of services. *uDiscovery* improves the discovery efficiency in the presence of variable number of services compared against baselines (Section 5.2). But, different scalability issues can be caused because of concurrent consumers performing a large number of requests at the same time. Simulation studies are limited to a single consumer that requests for services in a sequential fashion. Although the period between requests varies when there are city events, these studies do not show the impact of concurrent requests in *uDiscovery*. However, the distributed nature of *uDiscovery* can enable the response to multiple requests by different gateways in the network as an alternative to respond to concurrent consumers' requests.

6.3 Future Work

This thesis shows that *uDiscovery*, a distributed urban-centric model for service discovery, can improve the discovery efficiency compared with existing approaches in smart city environments, thought with some limitations. Future research directions are:

1. **Emergent Behaviour in Complex Systems:** The evaluation illustrated that *uDiscovery* fails to learn from city patterns to proactively adapt services organisation 5. This limitation can be caused because of the complexity of city environments, where random

behaviour can not be captured by the proposed method. Future work will focus on exploring in a deeper way current reinforcement learning algorithms and experimenting with possible adaptations, and alternatives to detect emergent behaviour in complex systems [O'Toole et al., 2017]. A first step towards the proactive organisation of services could be the experimentation with variable complexity to determine to what extent the environment complexity can be handled by learning methods.

2. **Physical Service Placement:** *uDiscovery* uses urban context (i.e., city places, and city events) to organise services descriptions in search spaces that evolve according to city changes. The physical deployment of services based on the *uDiscovery′* context model is an interesting area that needs further research. It would improve the performance of IoT-based smart city applications, by moving and deploying services on fog nodes, which are closer to final users. However, new research challenges emerge with regard to the migration of code between fog nodes, and the automatic deployment of services from heterogeneous technologies. For example, the distributed maintenance of services deployed on wide geographic areas, the transfer of large chunks of code to be deployed, and the deployment of services in heterogeneous platforms.

3. **Inclusion of Non-Functional Requirements:** *uDiscovery* searches for services based on a goal-driven approach. It generates service plans by matching inputs and outputs, which specify consumer's functional requirements. Consumers also have non-functional requirements which are critical in IoT because services might have variable and unpredictable QoS [White et al., 2017]. Further research is needed to include these non-functional parameters in an efficient service search that also considers QoS trade-off.

4. **Proactive Service Discovery:** *uDiscovery* discovers services in a reactive fashion as it needs a consumer request to trigger the search process. Proactive suggestion of services is needed in large IoT environments to realise the vision of future pervasive cyber-physical systems [Wang and Chow, 2016]. Further research is needed to provide proactive service discovery in smart cities [Wang and Chow, 2016]. Proactive discovery stores and formalises users information to personalise service suggestions [Zhou et al., 2016]. However, the large number of consumers represents a significant challenge for current approaches.

5. **Heterogeneous Services Management:** *uDiscovery* registers services descriptions which are annotated by specific ontologies, and follow a specific format. However, IoT

service providers can be reluctant to follow a specific description format or even provide a description. Further research is needed to manage heterogeneous IoT services in *uDiscovery*. An alternative is the automatic enrichment of services descriptions [Cassar et al., 2014, Liu et al., 2019] from the IoT services behaviour using machine learning techniques.

6. **Real Environment Implementation:** This thesis evaluates *uDiscovery* in simulated smart city environments with a data set which was created as part of this work. Future work will focus on the deployment of *uDiscovery* in real smart city environments with real services and descriptions. Further research is needed to create IoT environments similar to the proposed in this thesis, and to create data sets of services descriptions according to the services oriented computing future trends.

Appendix A

Algorithms' Parameters

Each approach evaluated in the Section 5 has a set of particular parameters that influence their performance. This appendix illustrates this influence and selects the best set of parameters for each approach. The parameters are as follows:

- Location-based Approach: This approach uses the distance between gateways and services' locations to decide whether to register a service, and the number of hops to limit the sending of messages in the network and avoid overhead.
- Domain-based Approach: This approach uses the number of domains to decide the set of services domains that each gateway is responsible for (i.e., the domains of services to store), and the number of hops to limit the sending of messages in the network and avoid overhead.
- *uDiscovery*: This approach uses the distance between gateways and city places' locations to decide whether a place surrounds a gateway, and the number of hops to limit the sending of messages in the network and avoid overhead.
- Heuristic Service Planner: Each approach in the simulation uses a planner to search for services in local repositories. This planner uses the $feedbackThreshold$, $functionalThreshold$, and K value parameters to drive progressive searches on the services repository.

A.1 Location-based Approach Parameters

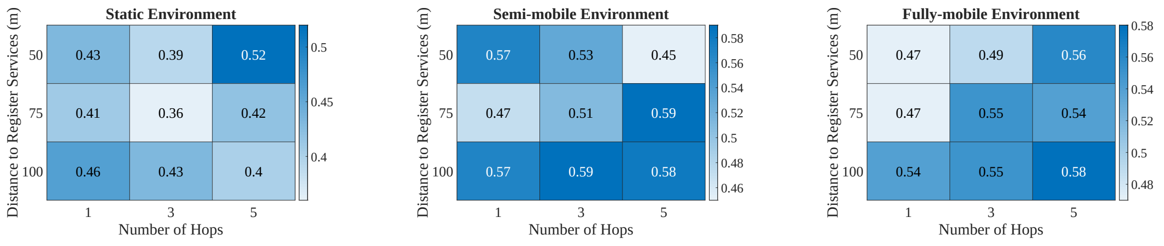
Table A.1 describes the experiment design that is used to select the best set of parameters for the location-based approach. Each experiment simulates 8 hours of services requests where a simulated consumer performs 100 requests. The experiments' parameters are the number

Table A.1: Experiments Parameters Location-based Approach

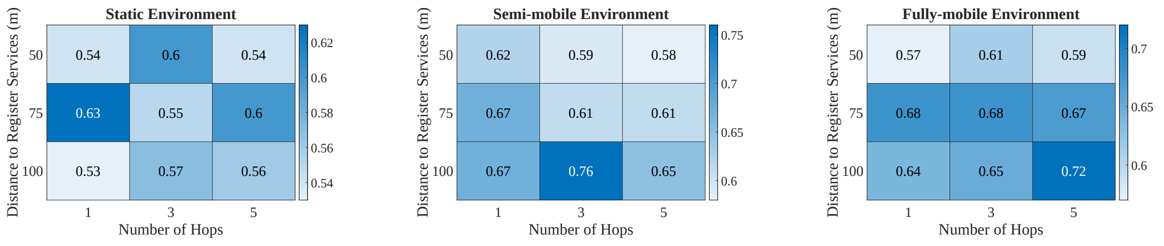
Length of time for service discovery	8 hours
# of consumer requests	100
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	100; 300; 500
Mobility scenarios	Static; semi-mobile; fully-mobile
Distance to register services	50m; 75m; 100m
Hops limit	1; 3; 5
Scenarios	Services X gateways X mobility X distance X hops
Replication	10 rounds each experiment

of services, the number of gateways, the mobility scenarios, the distance to register services, and the hops limit. They are combined in an scenario that evaluates the influence of each parameter. Each experiment is repeated 10 rounds.

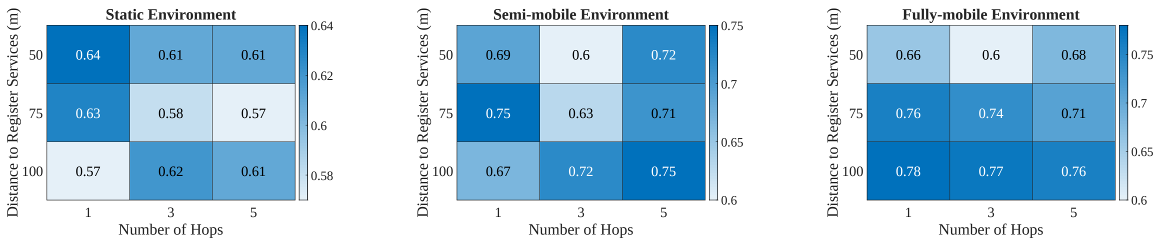
Figures A.1, A.3, and A.5 show heat maps of the rate of solved requests for different combinations of parameter under different number of services, mobility scenarios, and gateways. FigureA.1 shows the rate of solved request with 100 gateways. This rate varies from 0.36, when distance equal to 75m and hops limit 3 with 20 thousand services, to 0.85, when distance equal to 100m and hops 5 with 80 thousand services. The rate of solved requests increases when the number of gateways increase and there are mobile gateways. Mobile gateways carry services through a network where gateways are further from each other. Gateways are closer when the network size increases (i.e., 300 and 500 gateways), and so the rate of solved requests also increases. The maximum rate of solved requests is 0.87, with 300 gateways, and 0.96 with 500. Both cases when the distance is equal to 100m the hops limit 5 (Figures A.3e, and A.5e). A Kruskal-Wallis test was performed on the rate of solved requests for each combination to analyse parameters influence. Figures A.2, A.4, A.6 present a multi comparison of ranked means that compares different combinations of the approach parameters (i.e., hops limit, and distance to register a service). Figure A.2 shows the comparison with 100 gateways, there are not significant differences in most of the cases but it changes when the network size increases. Figure A.4 and A.4 illustrate the comparison when the number of gateways is 300 and 500. The combination of 5 hops and 100m get the highest mean in most of the cases. The best set of parameters for the location-based approach are 5 hops and 100 m, and so these were used in the baseline simulations for the evaluation in Chapter 5.



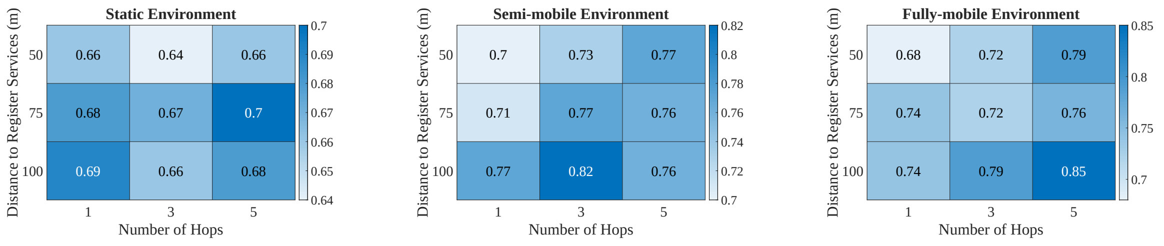
(a) 20000 Services



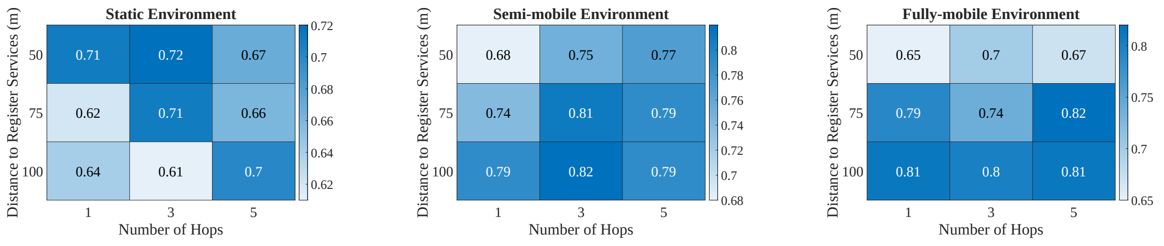
(b) 40000 Services



(c) 60000 Services

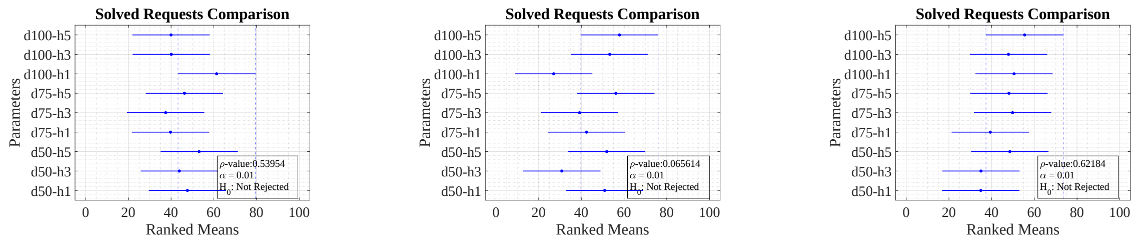


(d) 80000 Services

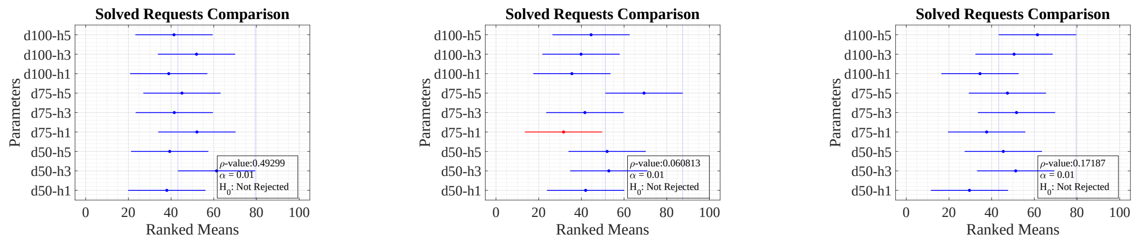


(e) 100000 Services

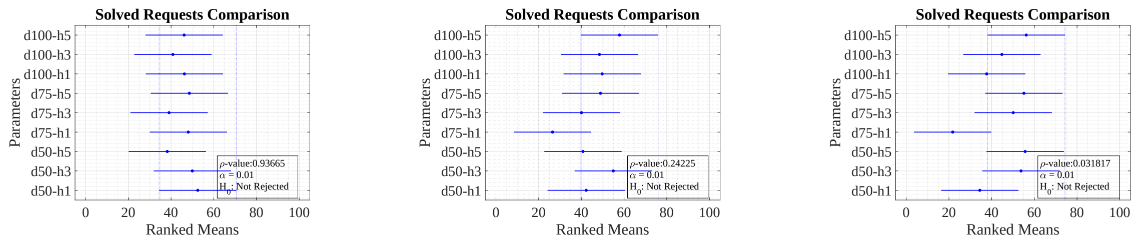
Figure A.1: Location-based Approach: Rate of solved requests heat maps with 100 gateways.



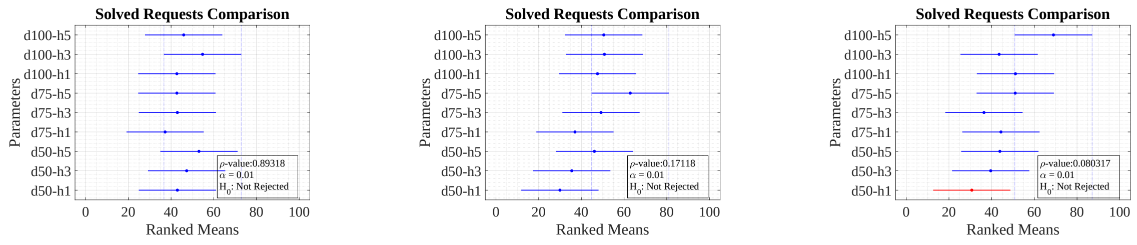
(a) 20000 Services



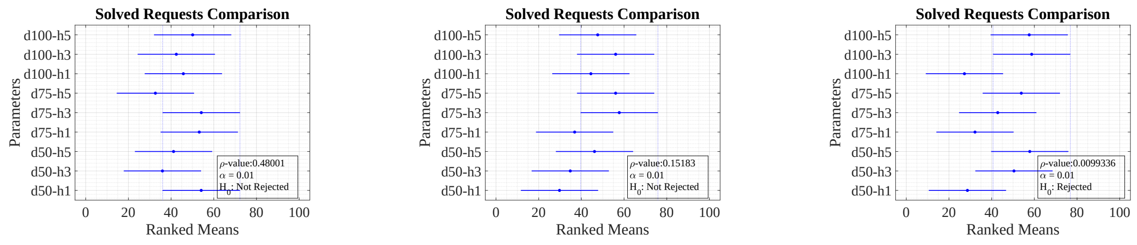
(b) 40000 Services



(c) 60000 Services

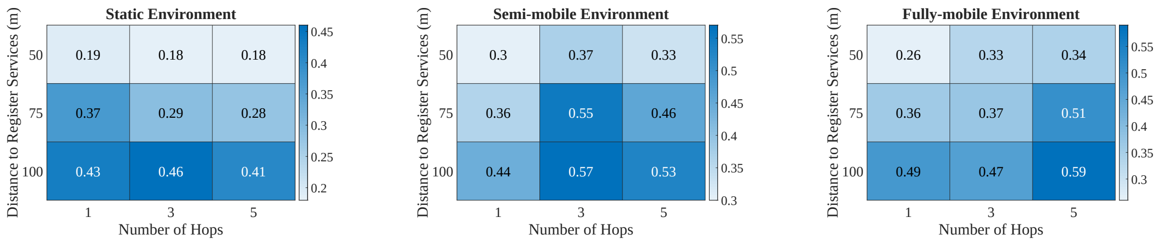


(d) 80000 Services

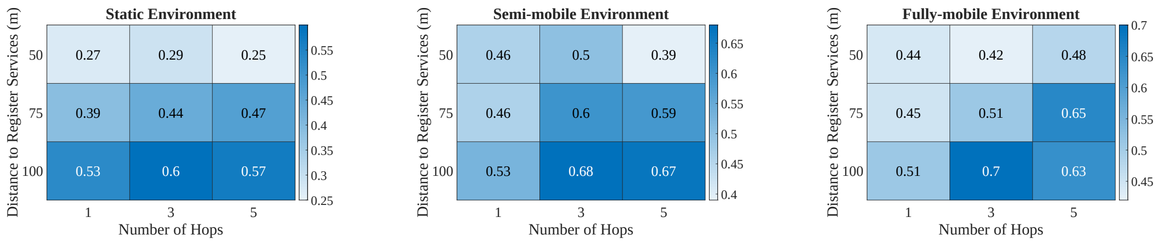


(e) 100000 Services

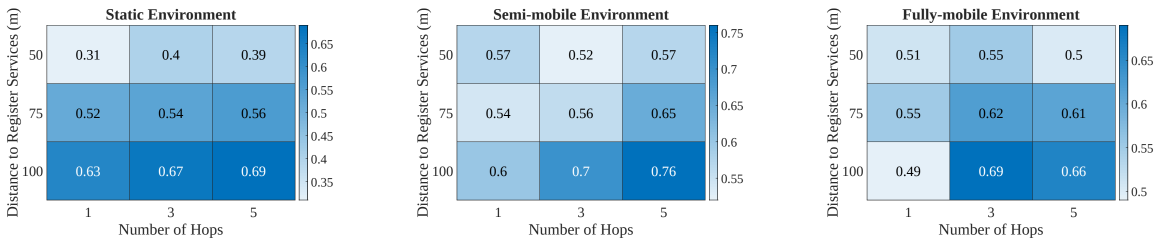
Figure A.2: Location-based Approach: Comparison of solved requests with 100 gateways.



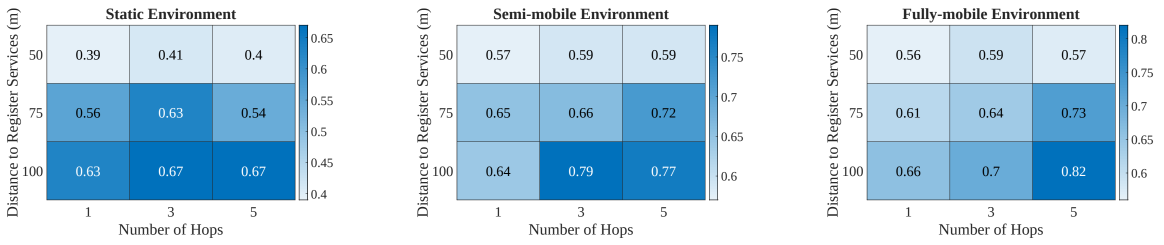
(a) 20000 Services



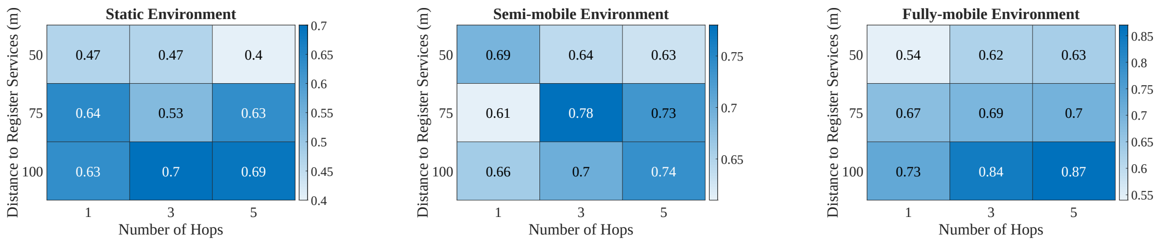
(b) 40000 Services



(c) 60000 Services

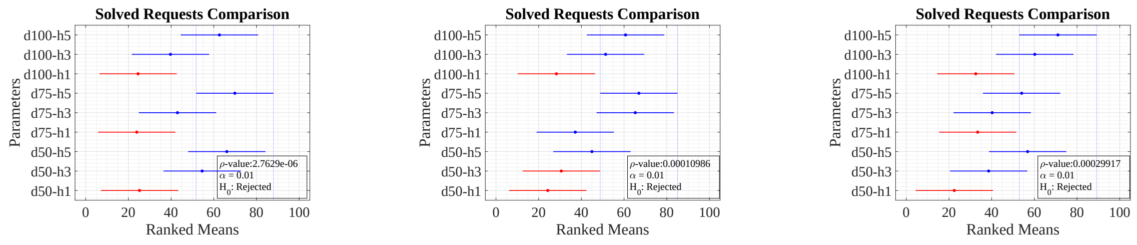


(d) 80000 Services

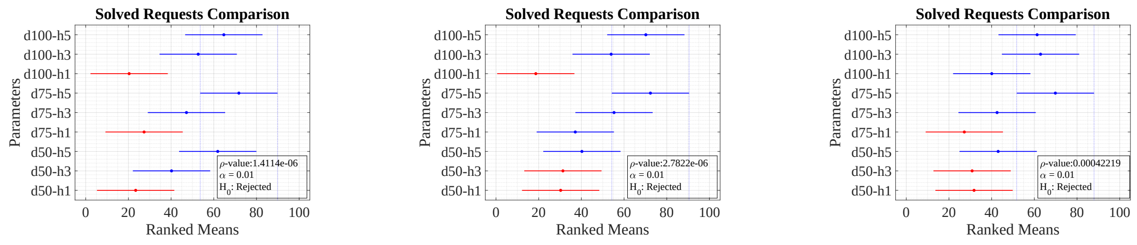


(e) 100000 Services

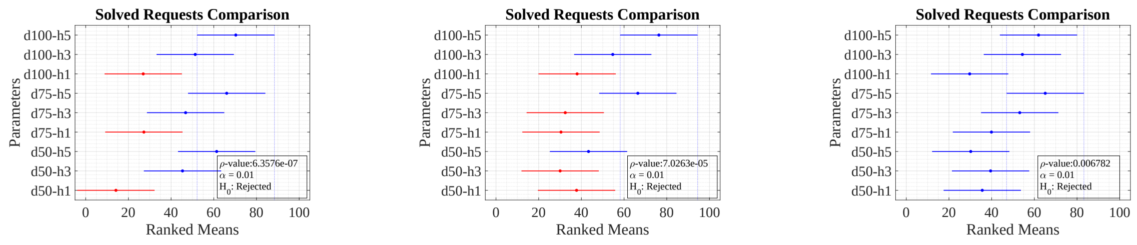
Figure A.3: Location-based Approach: Rate of solved requests heat maps with 300 gateways.



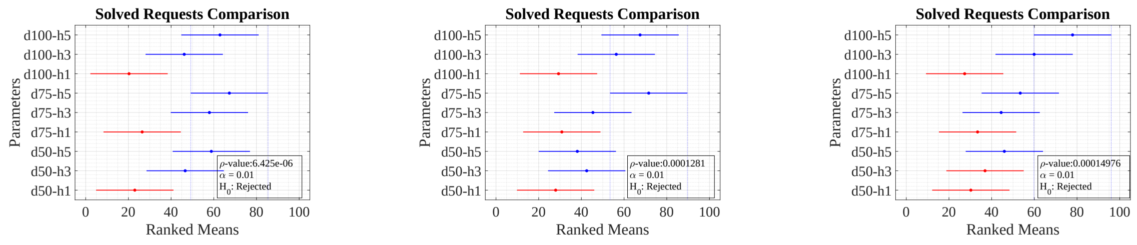
(a) 20000 Services



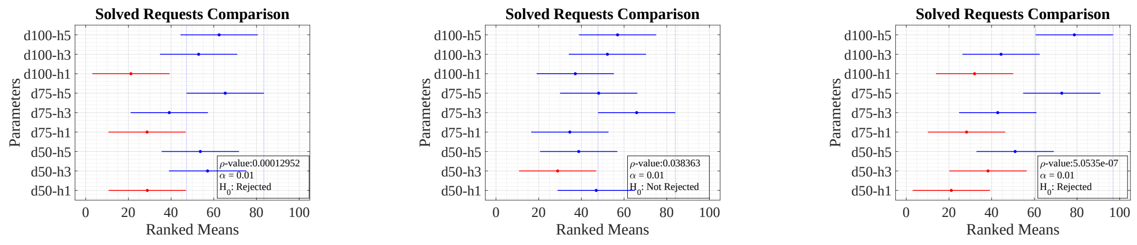
(b) 40000 Services



(c) 60000 Services

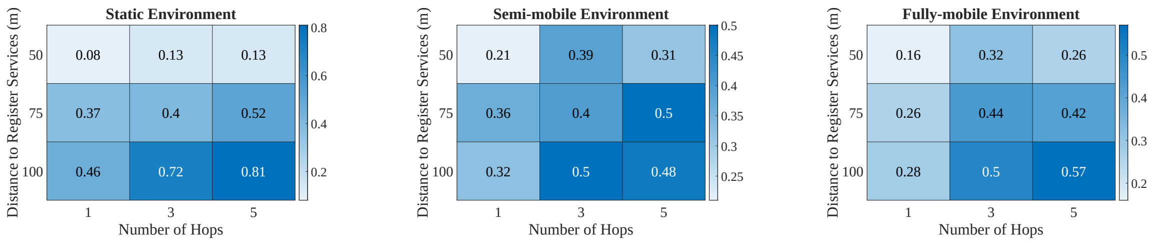


(d) 80000 Services

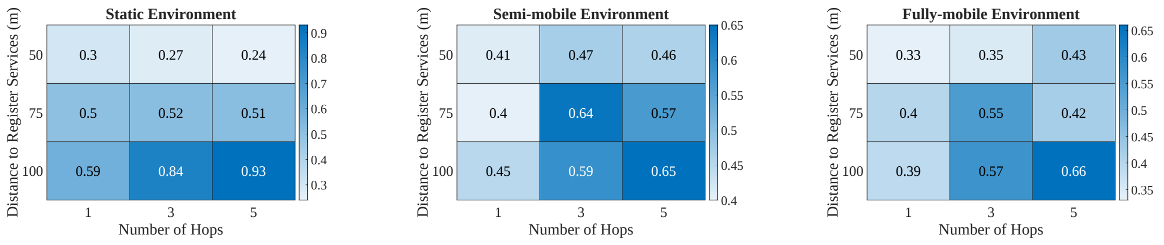


(e) 100000 Services

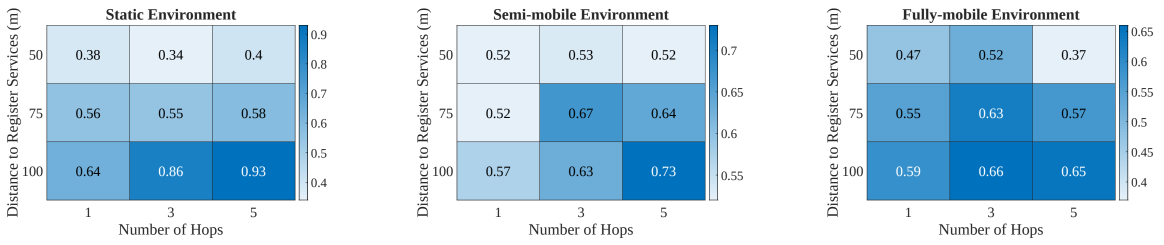
Figure A.4: Location-based Approach: Comparison of solved requests with 300 gateways.



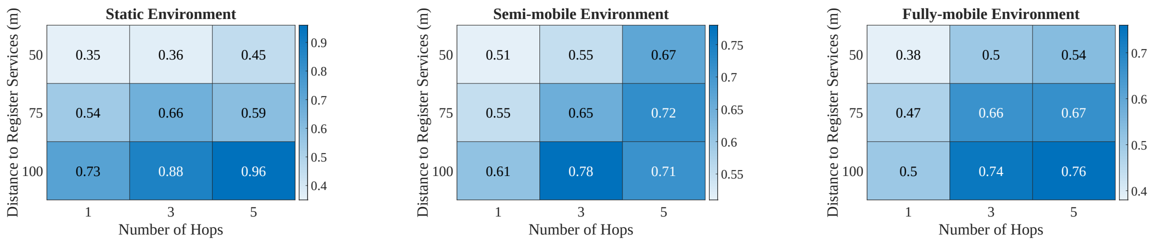
(a) 20000 Services



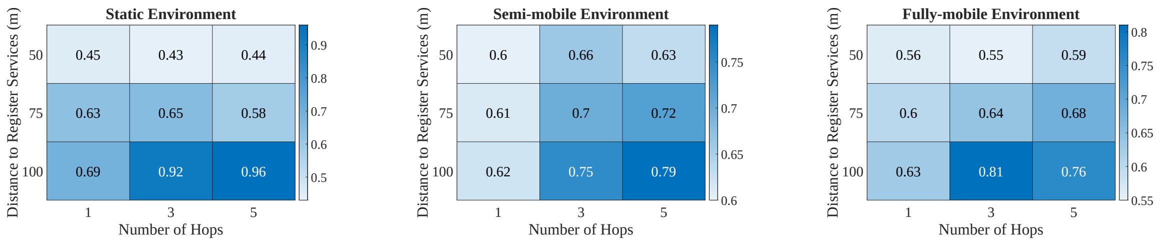
(b) 40000 Services



(c) 60000 Services

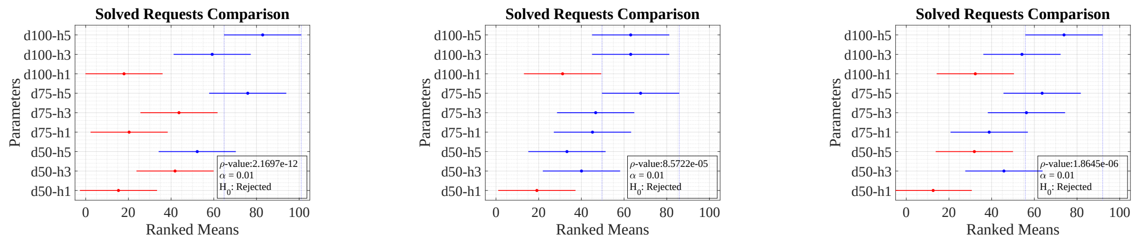


(d) 80000 Services

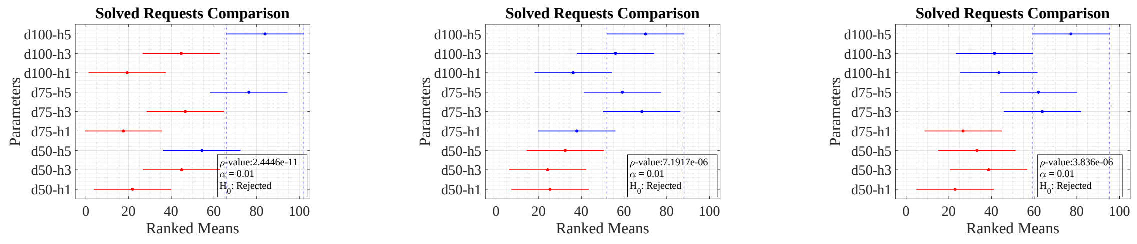


(e) 100000 Services

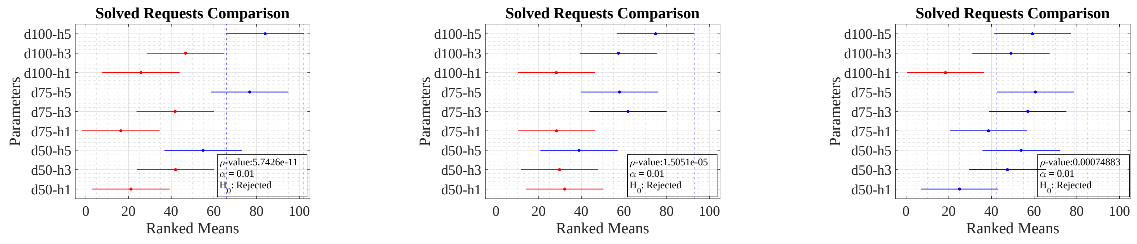
Figure A.5: Location-based Approach: Rate of solved requests heat maps with 500 gateways.



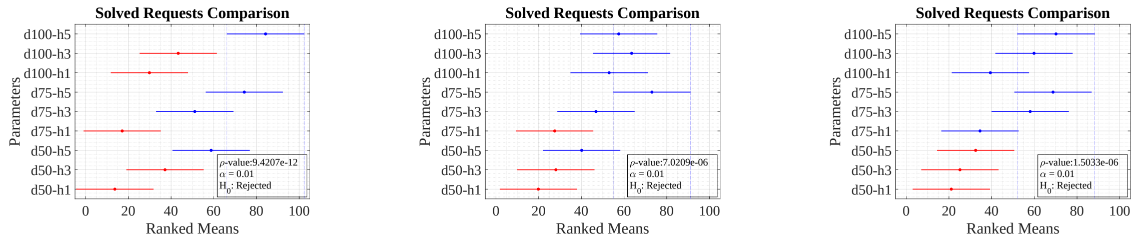
(a) 20000 Services



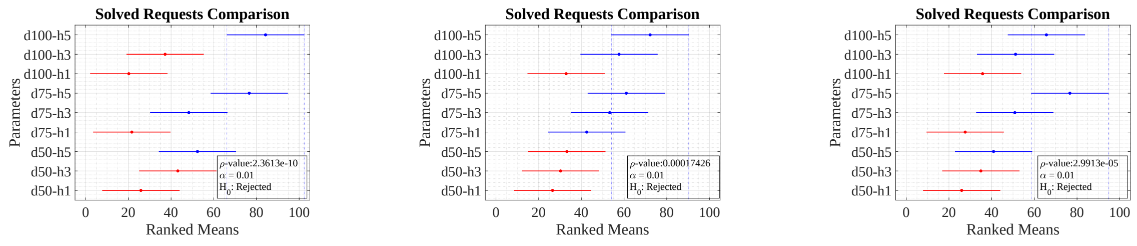
(b) 40000 Services



(c) 60000 Services



(d) 80000 Services



(e) 100000 Services

Figure A.6: Location-based Approach: Comparison of solved requests with 500 gateways.

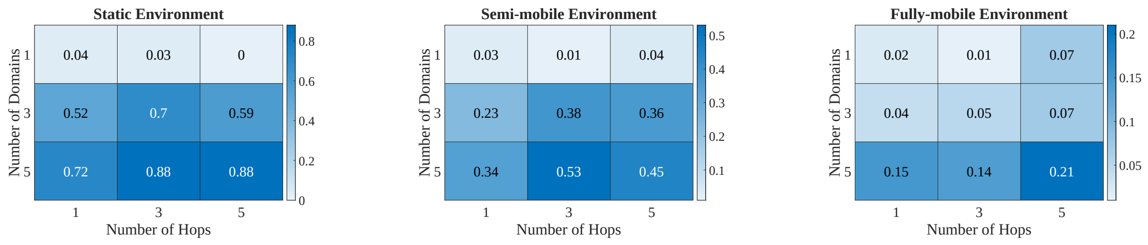
Table A.2: Experiments Parameters Domain-based Approach.

Length of time for service discovery	8 hours
# of consumer requests	100
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	100; 300; 500
Mobility scenarios	Static; semi-mobile; fully-mobile
Number of Domains	1; 3; 5
Hops limit	1; 3; 5
Scenarios	Services X gateways X mobility X domains X hops
Replication	10 rounds each experiment

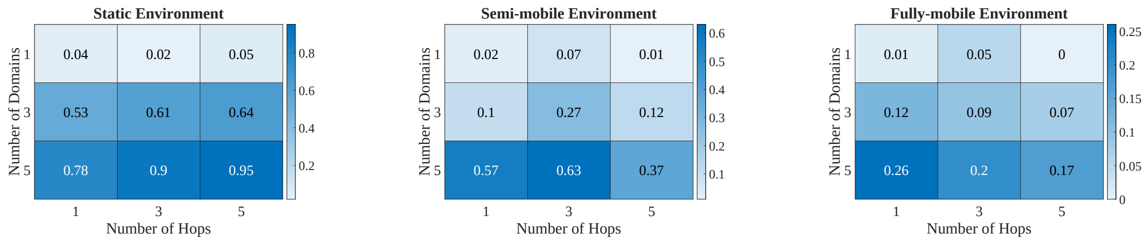
A.2 Domain-based Approach Parameters

Table A.1 describes the experiment design that is used to select the best set of parameters for the domain-based approach. Each experiment simulates 8 hours of services requests where a simulated consumer performs 100 requests. The experiments' parameters are the number of services, the number of gateways, the mobility scenarios, the number of domains, and the hops limit. They are combined in an scenario that evaluates the influence of each parameter. Each experiment is repeated 10 rounds.

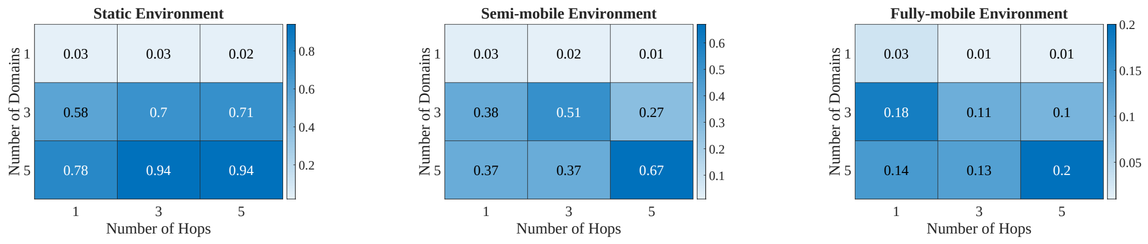
Figures A.7, A.9, and A.11 show the rate of solved requests of the domain-based approach for different combinations of parameters under different numbers of services, mobility scenarios, and network size. Figures show that the domain-based approach is negatively affected by mobile gateways in all cases. The rate of solved requests varies from 0 (i.e., fully-mobile environment) to 0.95 (i.e., static environment) with 100 gateways, from 0 to 0.86 with 300 gateways, and from 0 to 0.93 with 500 gateways. The domain-based approach has the highest rates of solved requests when the number of hops is 3 or 5 and the number of domains is 5. A Kruskal-Wallis test is performed on the rate of solved requests for each combination to confirm previous observations. Figures A.8, A.10, and A.12 present the multi comparison of the ranked means. Despite the network size, the combination of 5 domains and 3 hops, and the combination of 5 domains and 5 hops have a similar performance in most of the cases. However, the combination of 5 domains and 3 hops implies less network overhead. The best set of parameters for the domain-based approach are 5 domains, and 3 hops, and so these were used in the baseline simulations for the evaluation in Chapter 5.



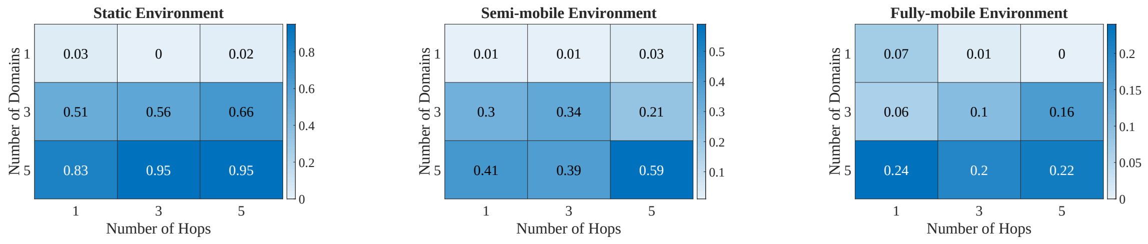
(a) 20000 Services



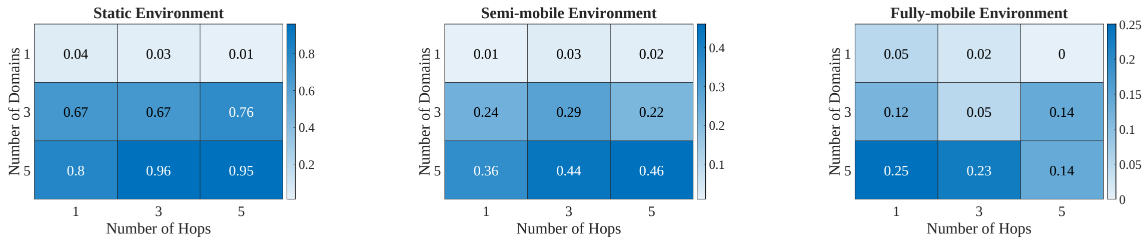
(b) 40000 Services



(c) 60000 Services

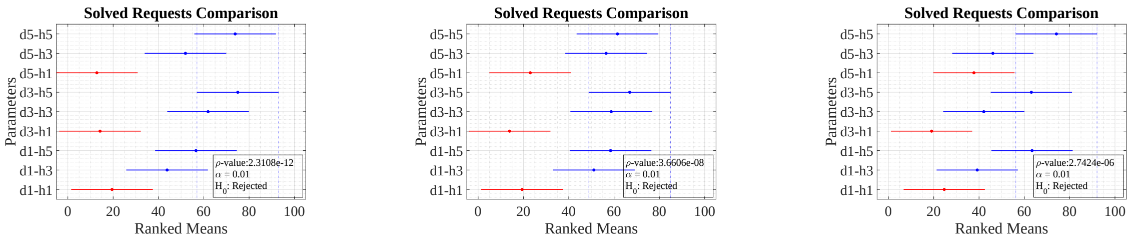


(d) 80000 Services

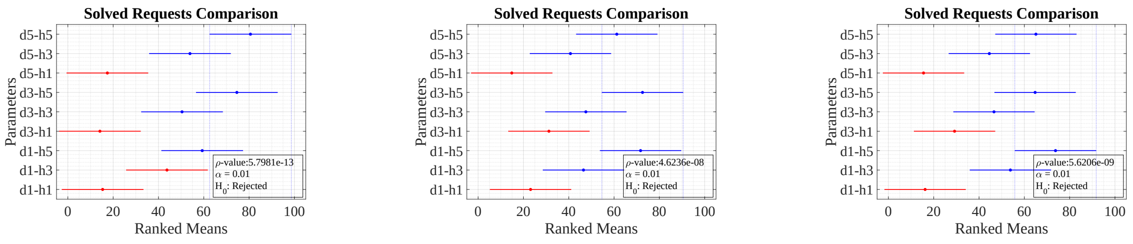


(e) 100000 Services

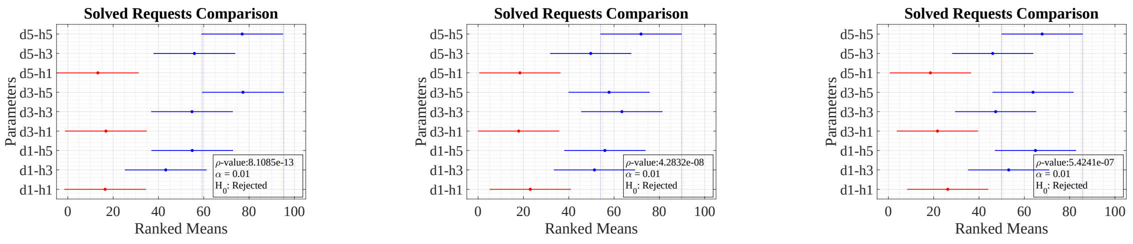
Figure A.7: Domain-based Approach: Rate of solved requests heat maps with 100 gateways.



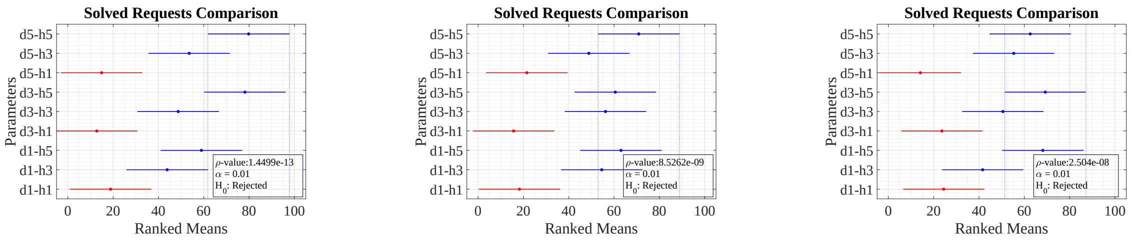
(a) 20000 Services



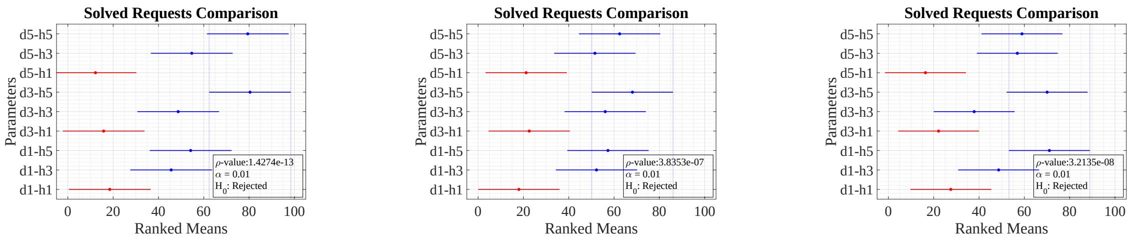
(b) 40000 Services



(c) 60000 Services

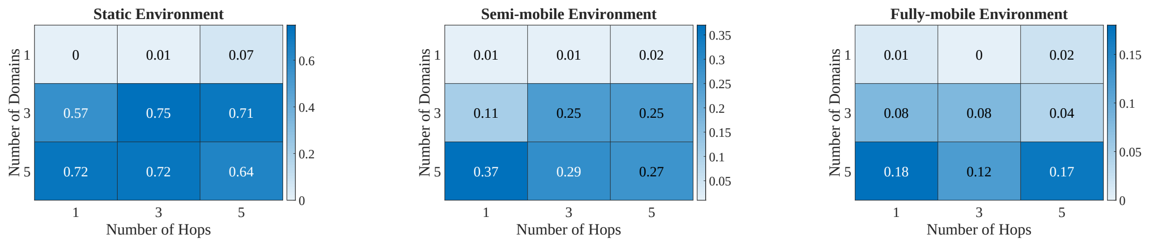


(d) 80000 Services

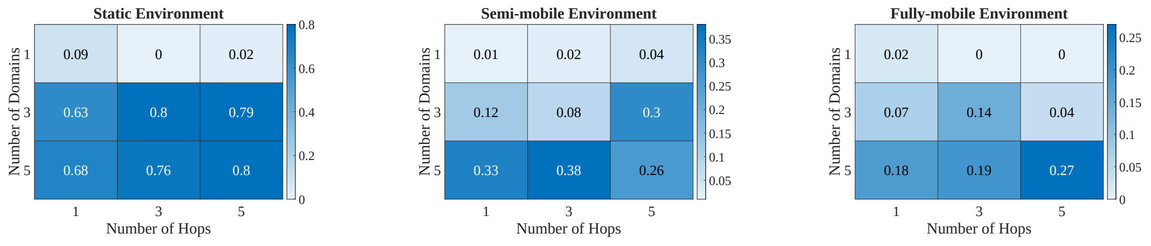


(e) 100000 Services

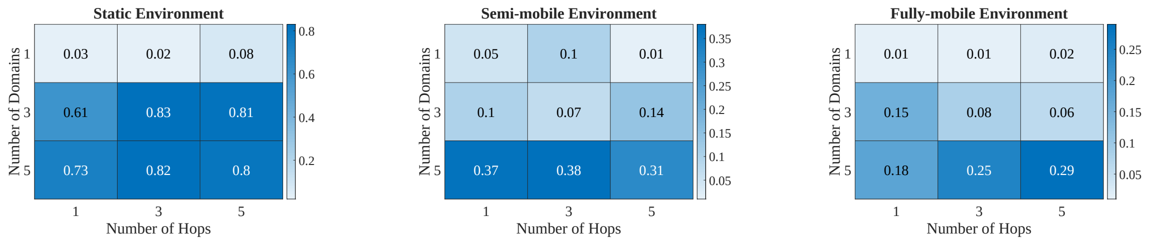
Figure A.8: Domain-based Approach: Comparison of solved requests with 100 gateways.



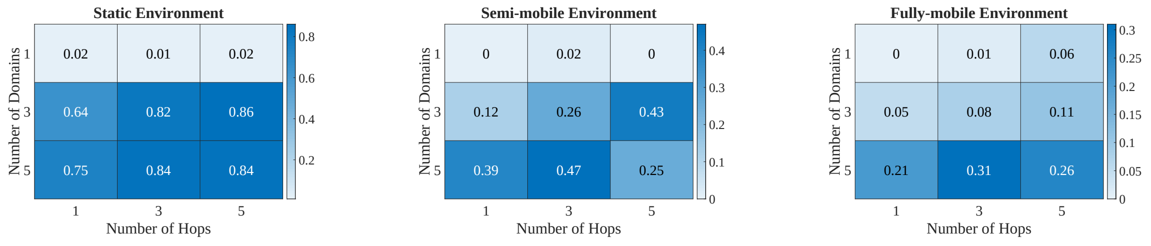
(a) 20000 Services



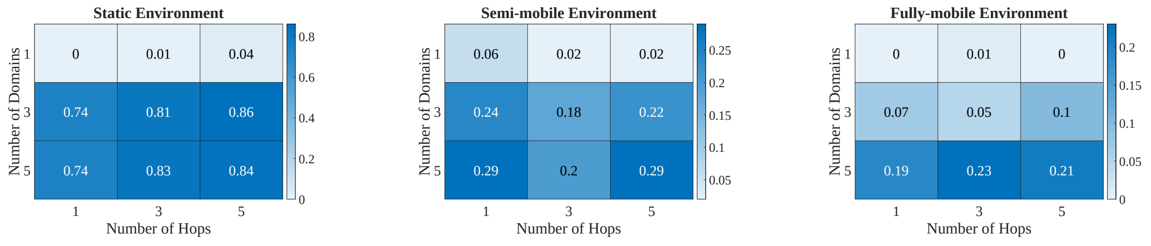
(b) 40000 Services



(c) 60000 Services

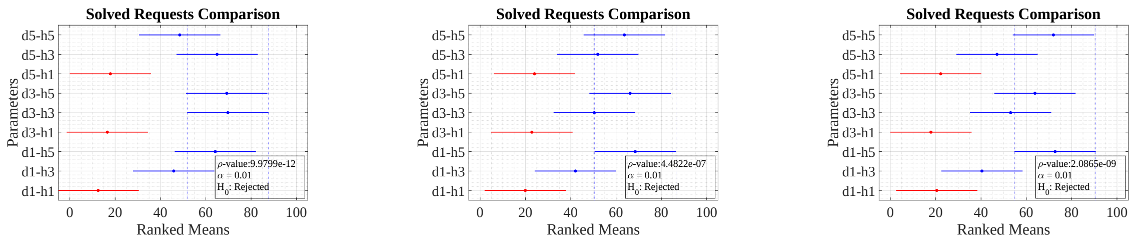


(d) 80000 Services

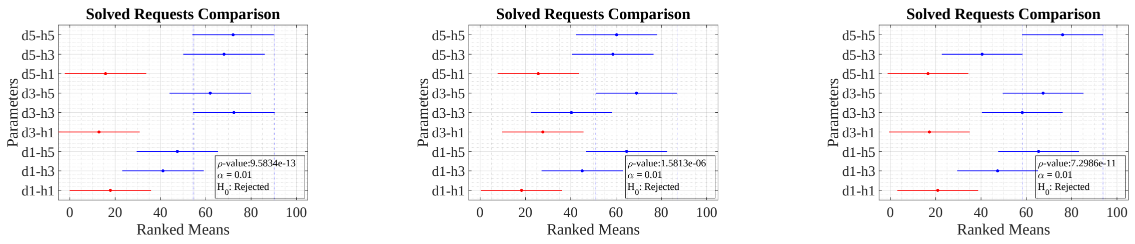


(e) 100000 Services

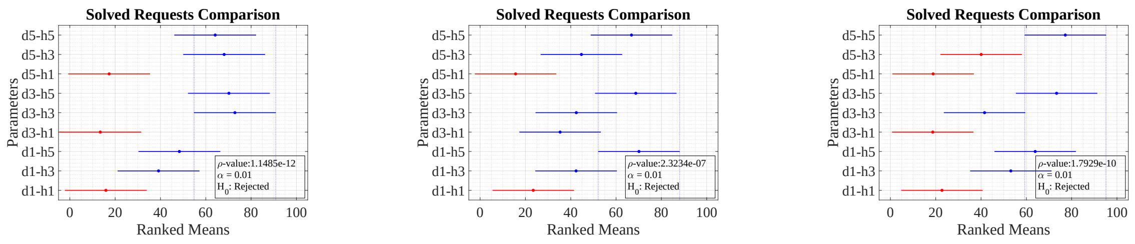
Figure A.9: Domain-based Approach: Rate of solved requests heat maps with 300 gateways.



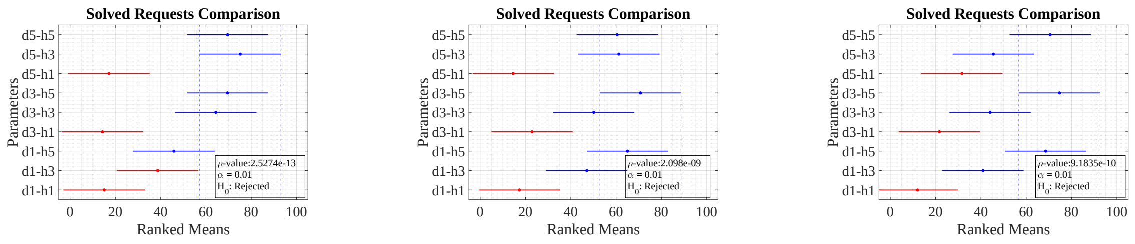
(a) 20000 Services



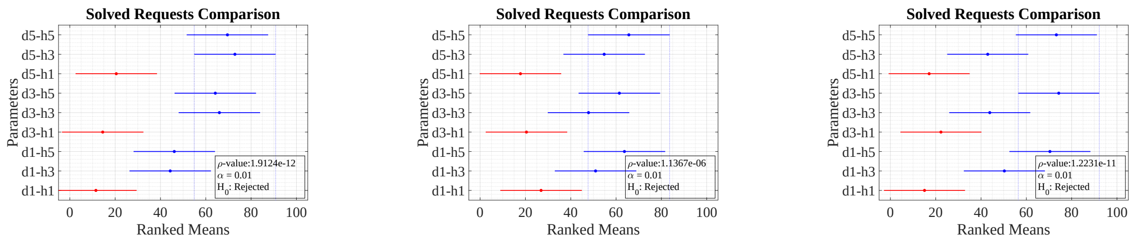
(b) 40000 Services



(c) 60000 Services

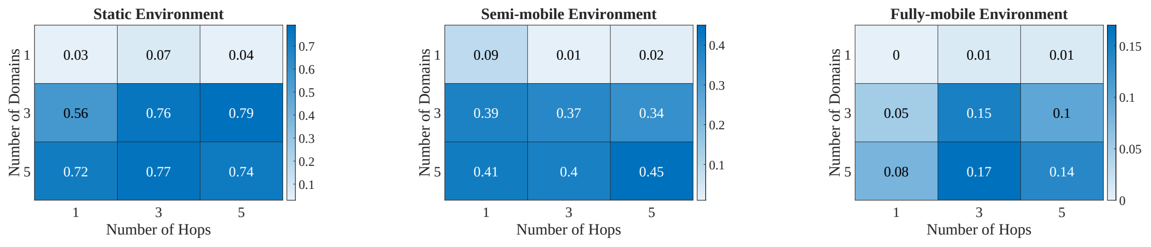


(d) 80000 Services

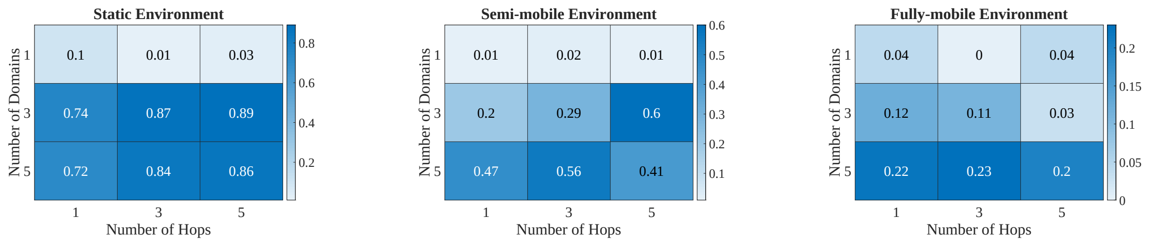


(e) 100000 Services

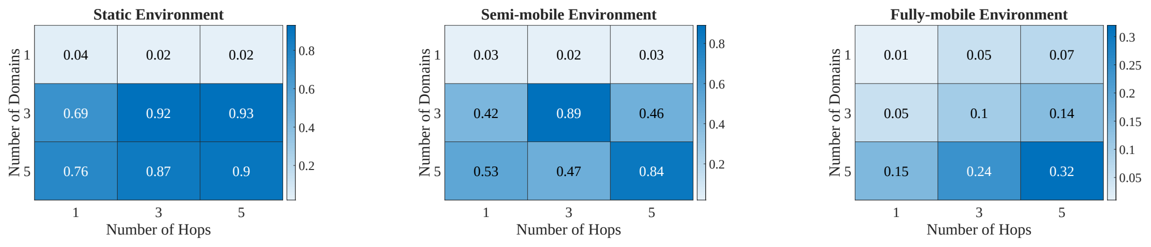
Figure A.10: Domain-based Approach: Comparison of solved requests with 300 gateways.



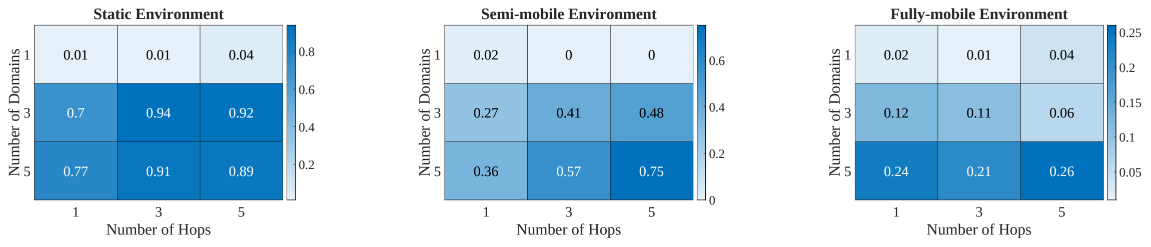
(a) 20000 Services



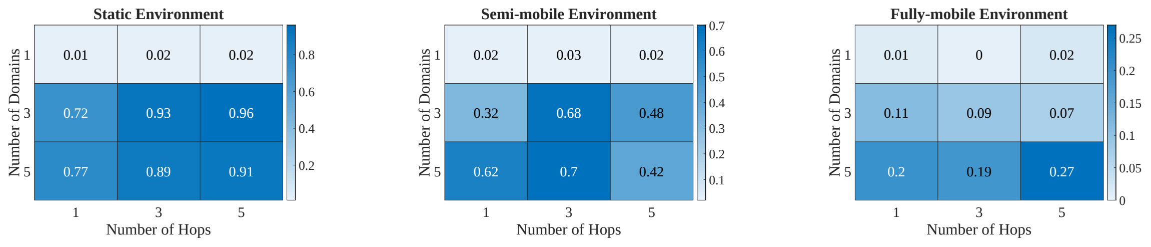
(b) 40000 Services



(c) 60000 Services

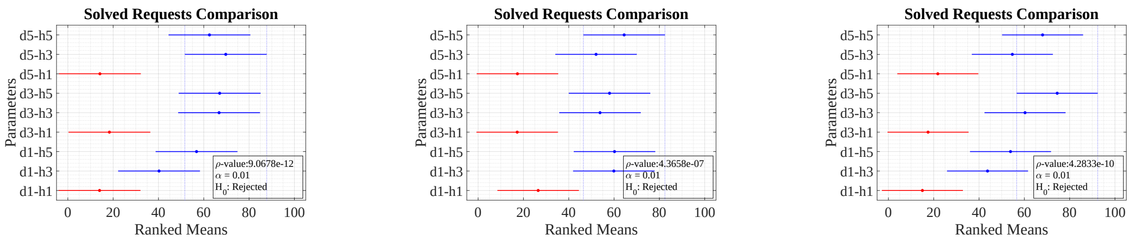


(d) 80000 Services

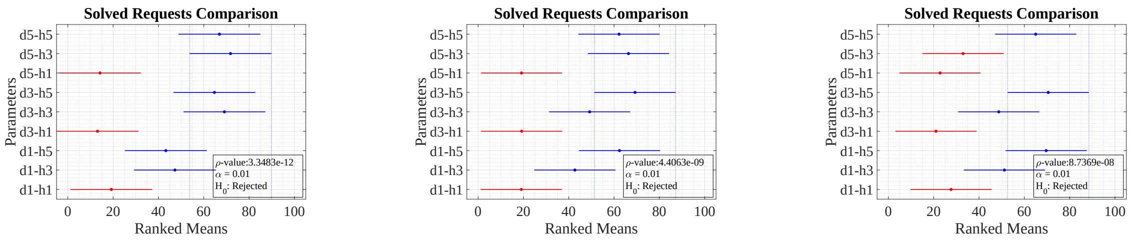


(e) 100000 Services

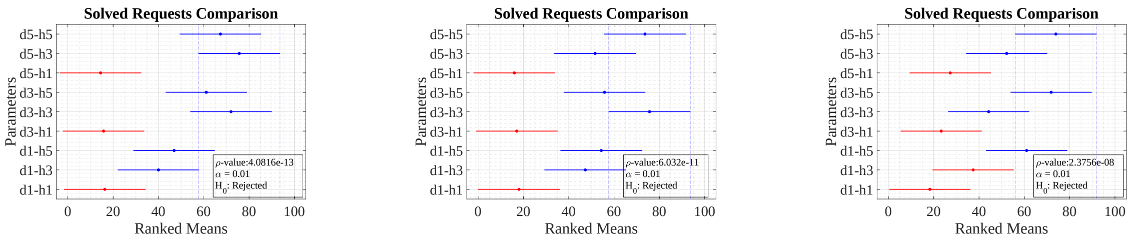
Figure A.11: Domain-based Approach: Rate of solved requests heat maps with 500 gateways.



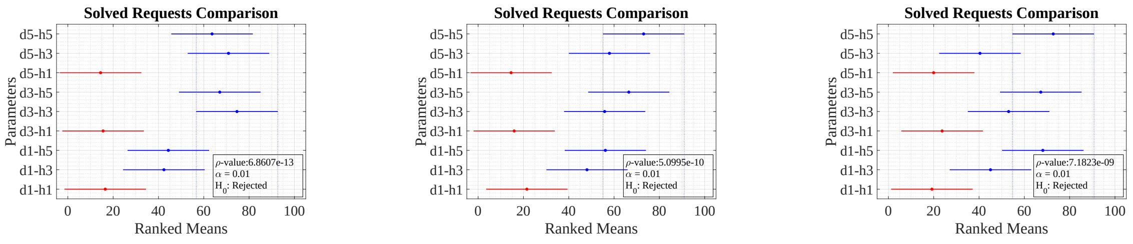
(a) 20000 Services



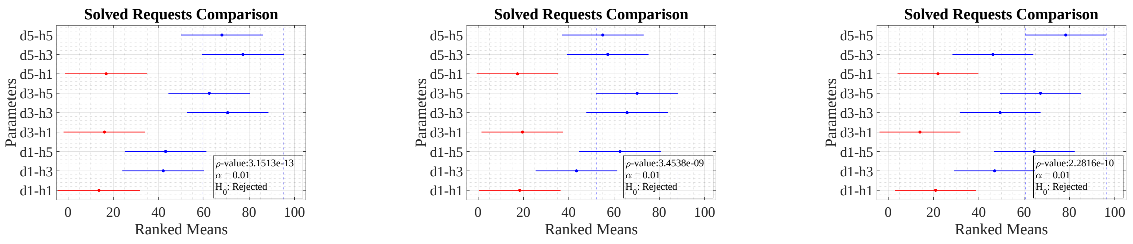
(b) 40000 Services



(c) 60000 Services



(d) 80000 Services



(e) 100000 Services

Figure A.12: Domain-based Approach: Comparison of solved requests with 500 gateways.

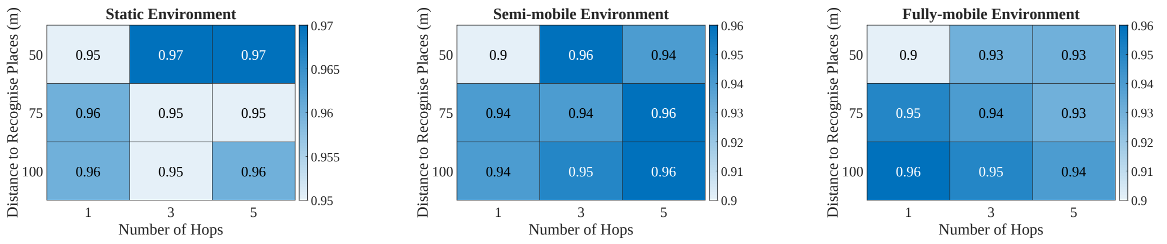
Table A.3: Experiments Parameters *uDiscovery* Approach.

Length of time for service discovery	8 hours
# of consumer requests	100
# of services	20,000; 40,000; 60,000; 80,000; 100,000
# of gateways	100; 300; 500
Mobility scenarios	Static; semi-mobile; fully-mobile
Distance to recognise places	50m; 75m; 100m
Hops limit	1; 3; 5
Scenarios	Services X gateways X mobility X distance X hops
Replication	10 rounds each experiment

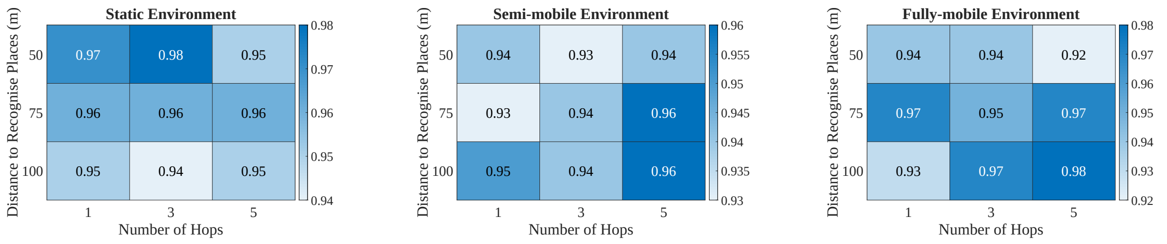
A.3 uDiscovery Parameters

Table A.1 describes the experiment design that is used to select the best set of parameters for the location-based approach. Each experiment simulates 8 hours of services requests where a simulated consumer performs 100 requests. The experiments' parameters are the number of services, the number of gateways, the mobility scenarios, the distance to recognise close places, and the hops limit. They are combined in an scenario that evaluates the influence of each parameter. Each experiment is repeated 10 rounds.

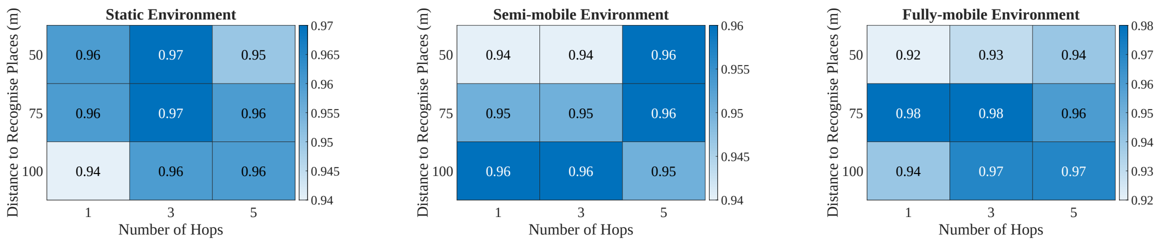
Figures A.13, A.15, and A.17 shows the rate of solved requests for *uDiscovery* under different numbers of services, mobility scenarios, and network size. Figure A.13 illustrates that *uDiscovery* has a similar and high rate of solved requests with 100 gateways, for most of the cases (i.e., it varies from 0.89 to 0.99). This behaviour is confirmed by the statistical tests in the Figure A.14, where most of the means are not statistically significantly different. A similar behaviour is observed when there are 300 gateways (Figure A.15). The rate of solved requests varies from 0.86 to 0.99. The best rates are achieved when the number of hops and the distance are high, according to the Figure A.16). The combination of 5 hops and 100m achieves the greatest ranked mean in 7 out of 15 cases, and the combination of 3 hops and 100m in 4 cases. Figure A.17 illustrates the rate of solved requests when there are 500 gateways. This rate varies from 0.83 to 0.99, and is greater in static scenarios. The figures show that *uDiscovery* has higher rates of solved requests with 3 or 5 hops. The best distance to recognise close places in static environments is 50, but it changes in semi-mobile and fully-mobile environments when 75 and 100m offer higher rate of solved requests. The Kruskal-Wallis (Figure A.18) shows that the combination of 5 hops and 100 m achieves the highest mean in most of the mobile environments. These are the selected values for *uDiscovery* because



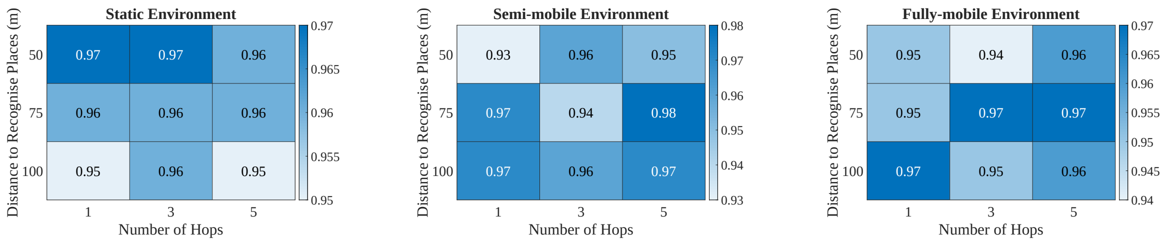
(a) 20000 Services



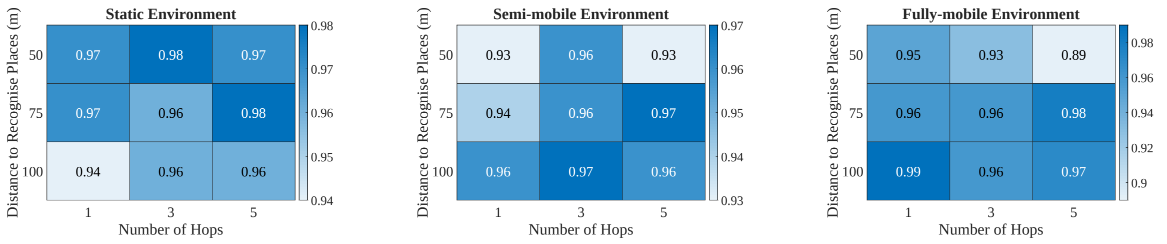
(b) 40000 Services



(c) 60000 Services

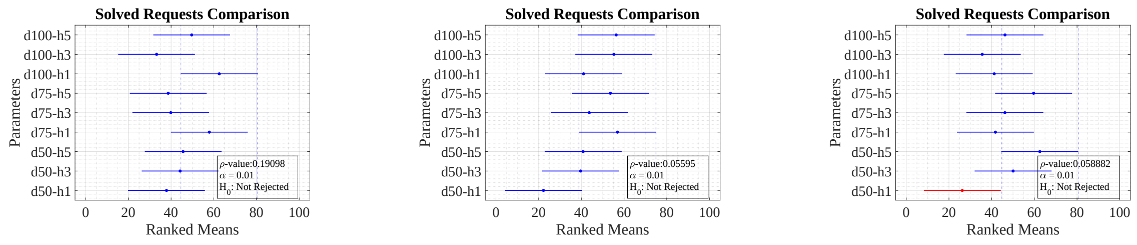


(d) 80000 Services

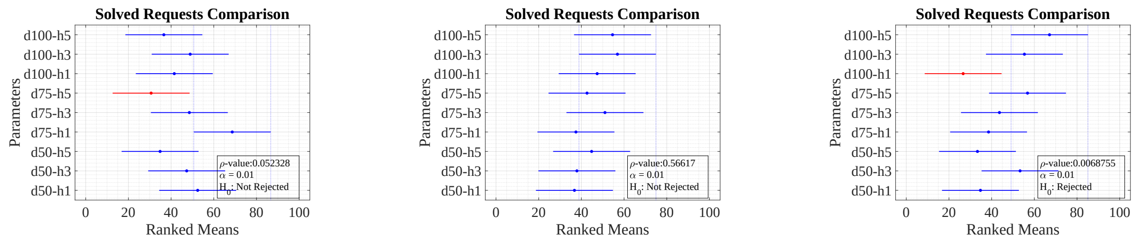


(e) 100000 Services

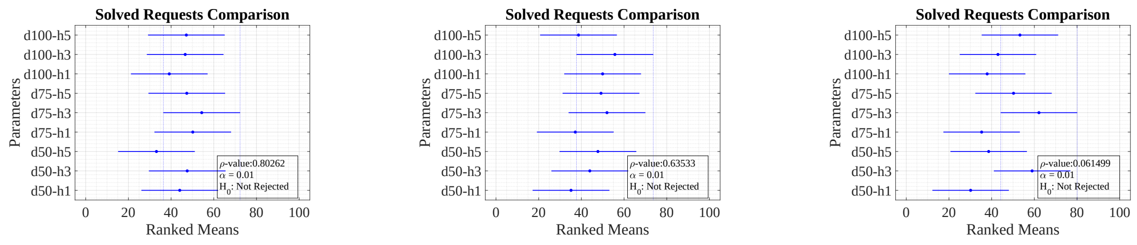
Figure A.13: uDiscovery Approach: Rate of solved requests heat maps with 100 gateways.



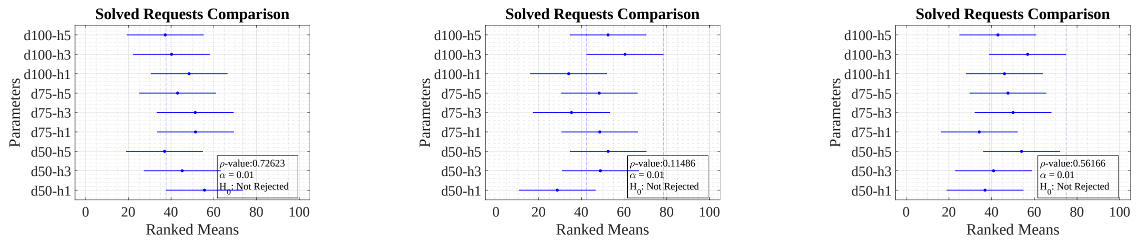
(a) 20000 Services



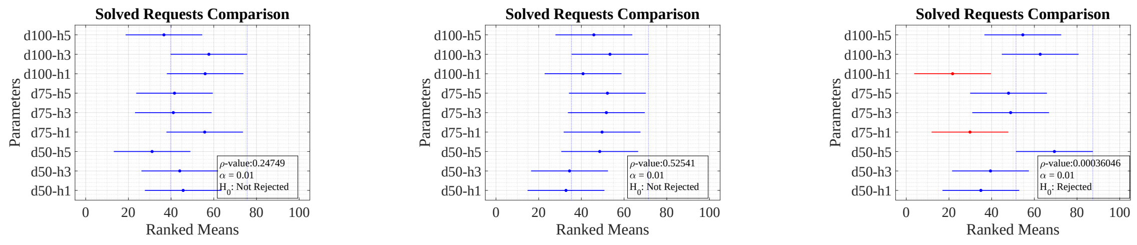
(b) 40000 Services



(c) 60000 Services

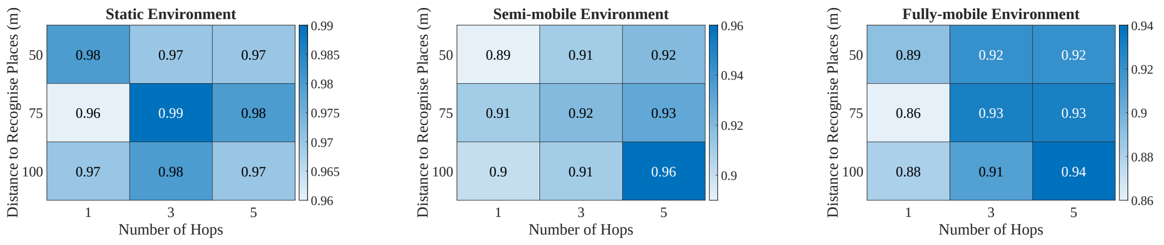


(d) 80000 Services

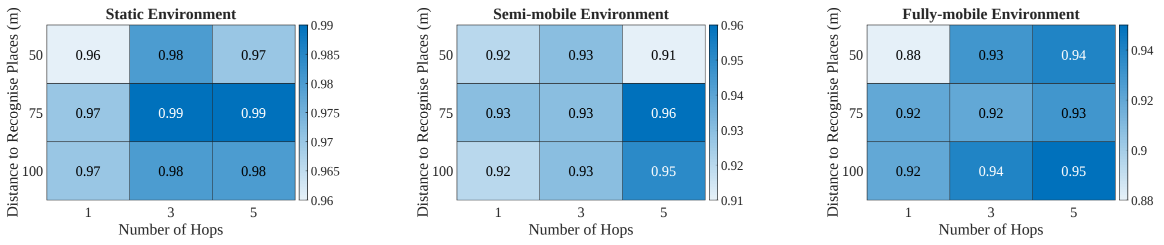


(e) 100000 Services

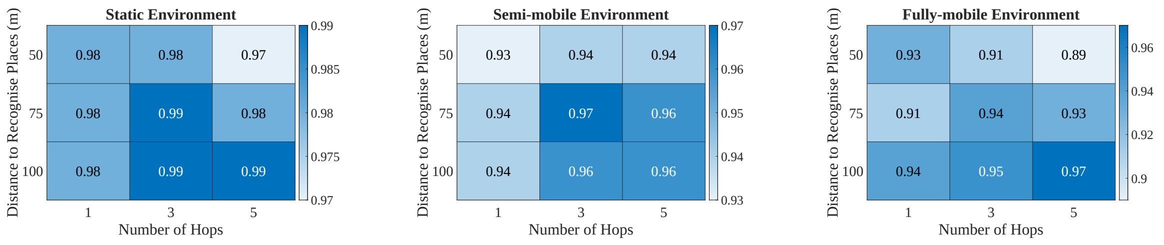
Figure A.14: uDiscovery Approach: Comparison of solved requests with 100 gateways.



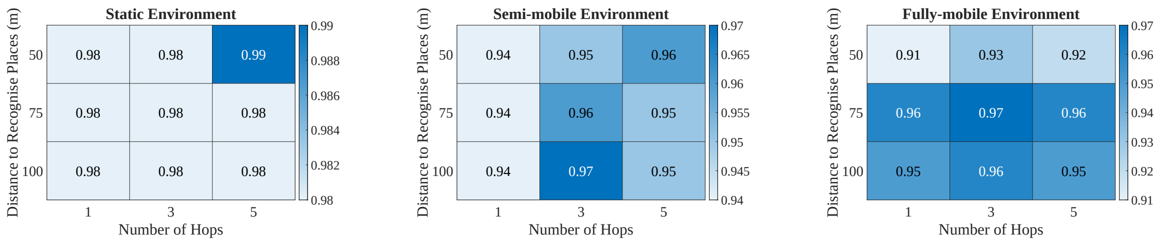
(a) 20000 Services



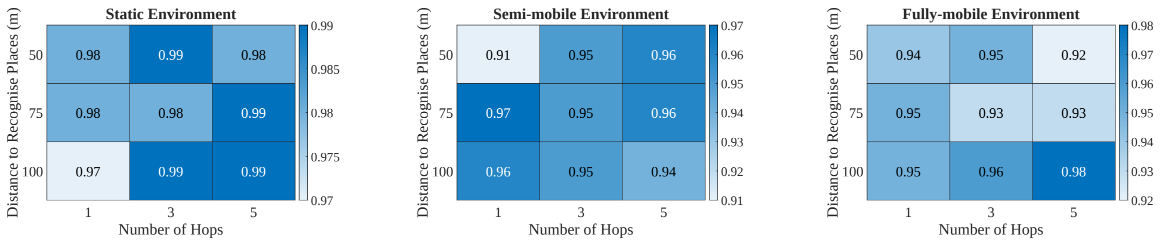
(b) 40000 Services



(c) 60000 Services

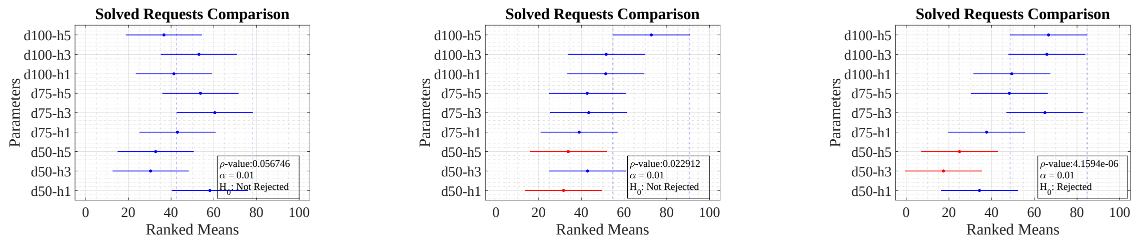


(d) 80000 Services

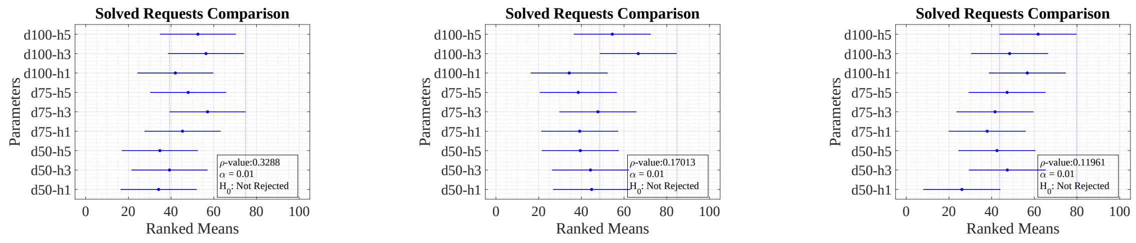


(e) 100000 Services

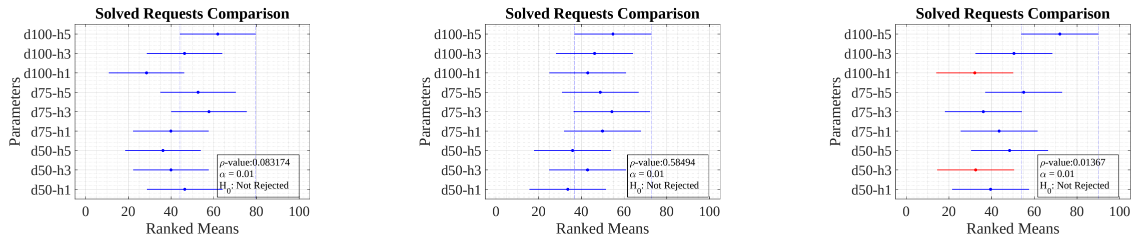
Figure A.15: uDiscovery Approach: Rate of solved requests heat maps with 300 gateways.



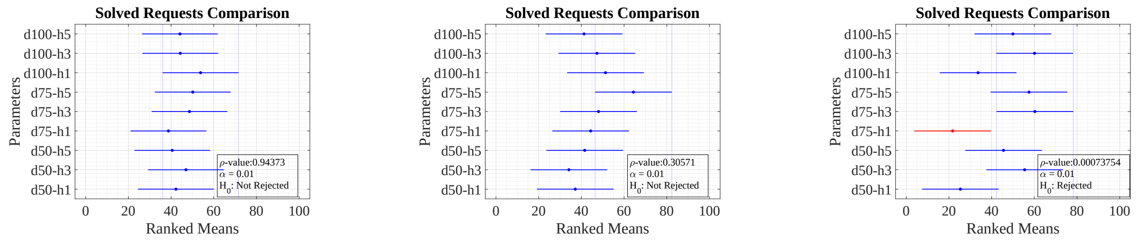
(a) 20000 Services



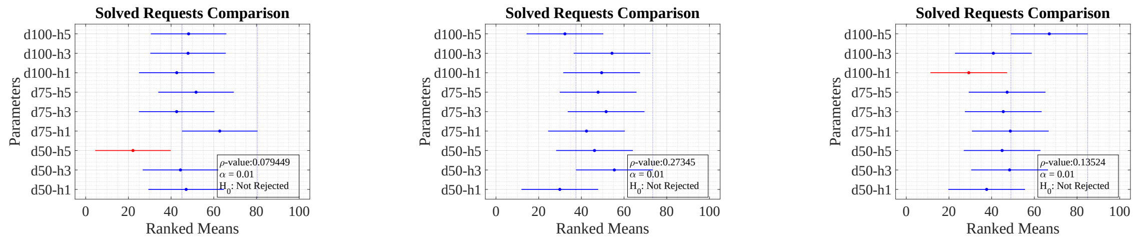
(b) 40000 Services



(c) 60000 Services

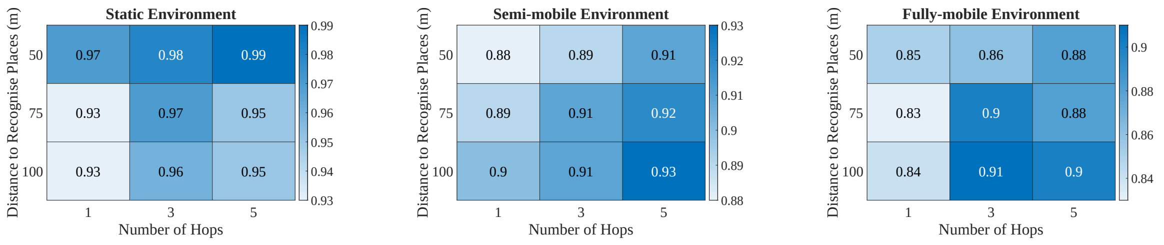


(d) 80000 Services

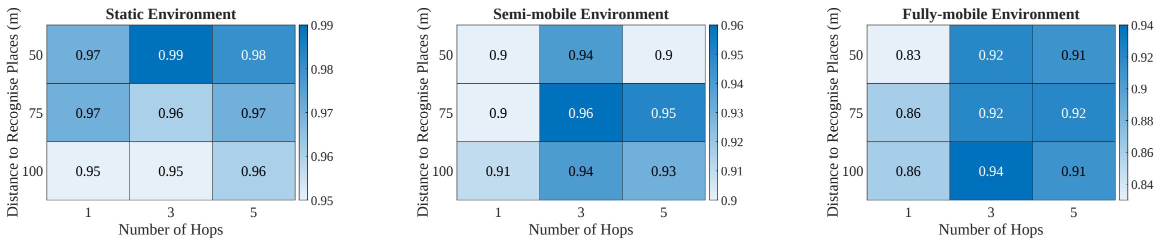


(e) 100000 Services

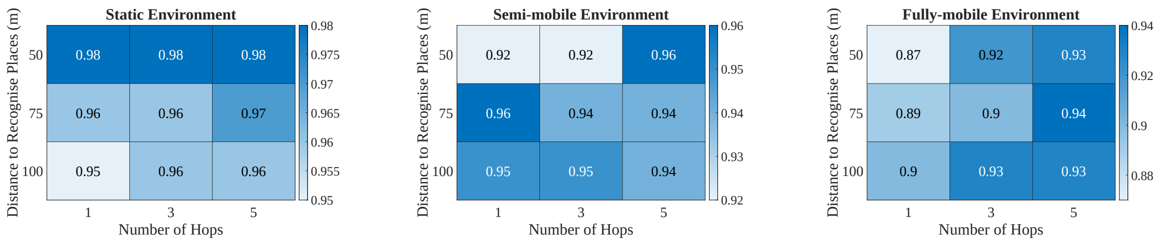
Figure A.16: uDiscovery Approach: Comparison of solved requests with 300 gateways.



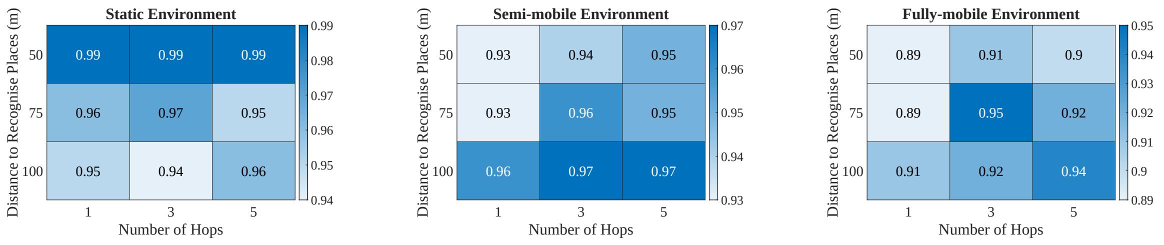
(a) 20000 Services



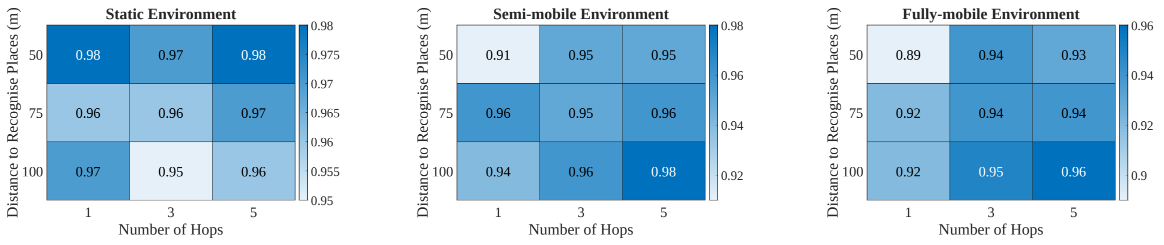
(b) 40000 Services



(c) 60000 Services

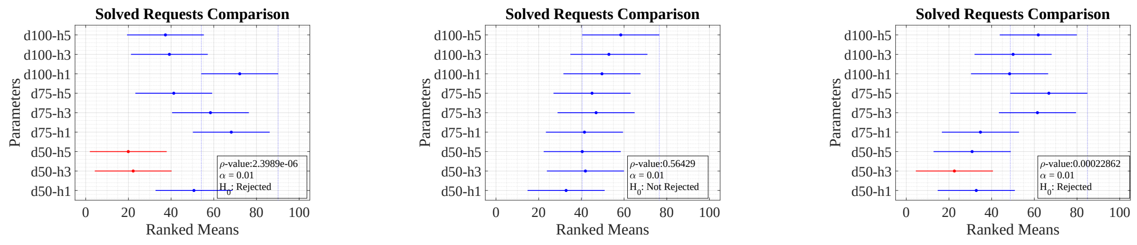


(d) 80000 Services

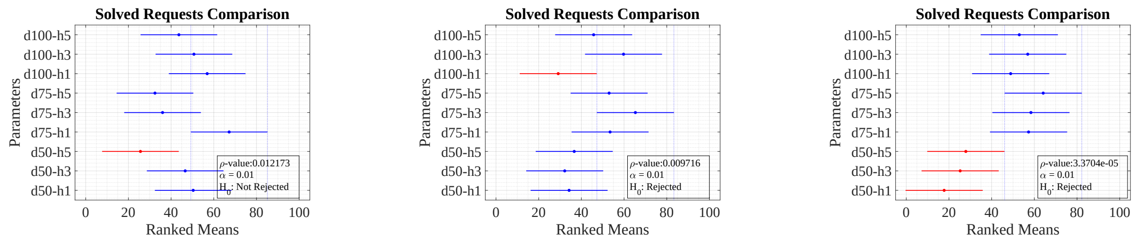


(e) 100000 Services

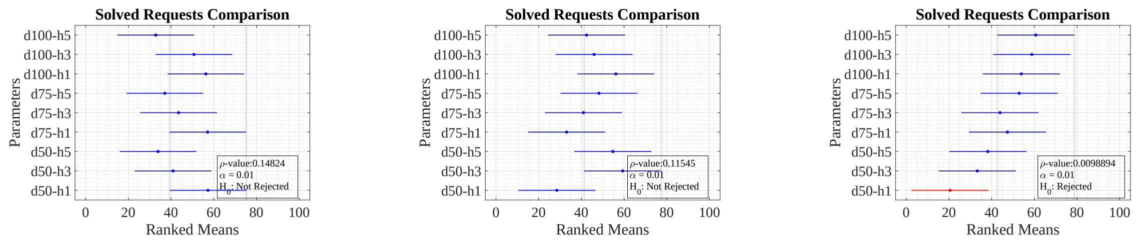
Figure A.17: uDiscovery Approach: Rate of solved requests heat maps with 500 gateways.



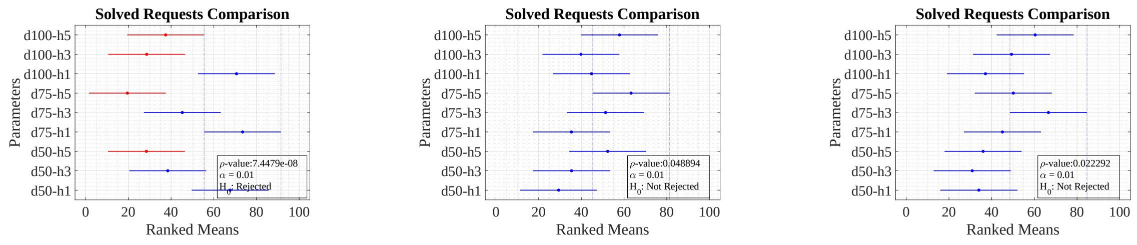
(a) 20000 Services



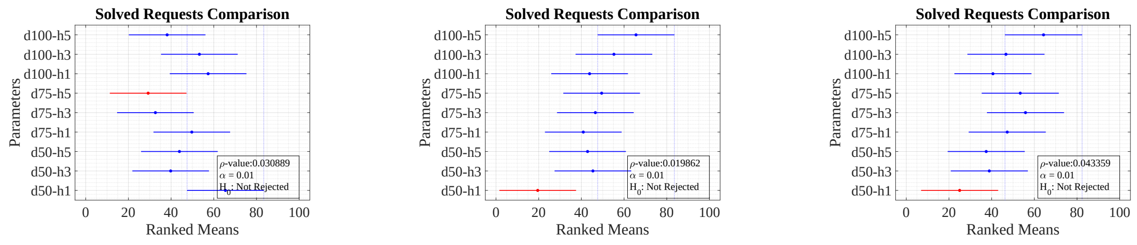
(b) 40000 Services



(c) 60000 Services



(d) 80000 Services



(e) 100000 Services

Figure A.18: uDiscovery Approach: Comparison of solved requests with 500 gateways.

Table A.4: Experiments Parameters *uDiscovery* Planner.

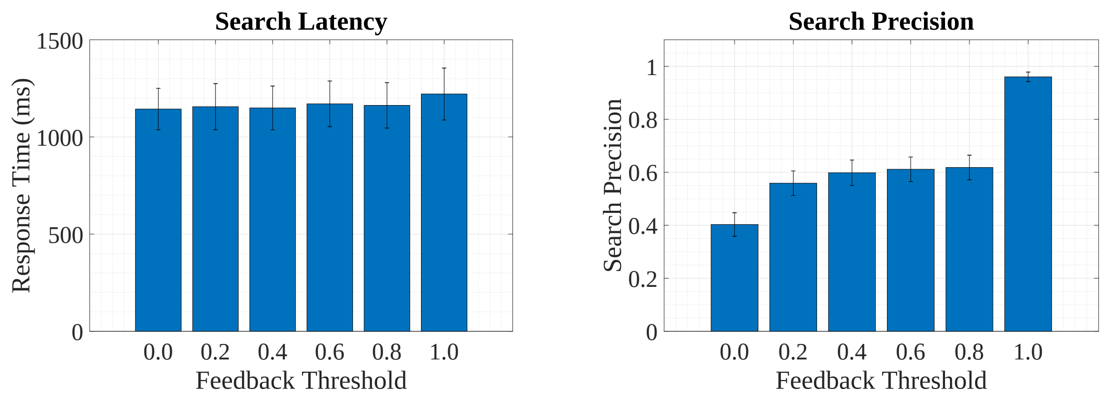
Experiment	The consumer requests one service
Feedback Threshold Values	0,0; 0,2; 0,4; 0,6; 0,8; 1,0
Functional Threshold Values	0,0; 0,2; 0,4; 0,6; 0,8; 1,0
K-Values	5; 15; 25
Scenarios	1. Variable feedback threshold 2. Variable functional threshold 3. Variable K-value
Replication	100 requests each experiment

mobile environments are more realistic in IoT and smart cities scenarios. These parameters were used in the simulations for the evaluation in Chapter 5.

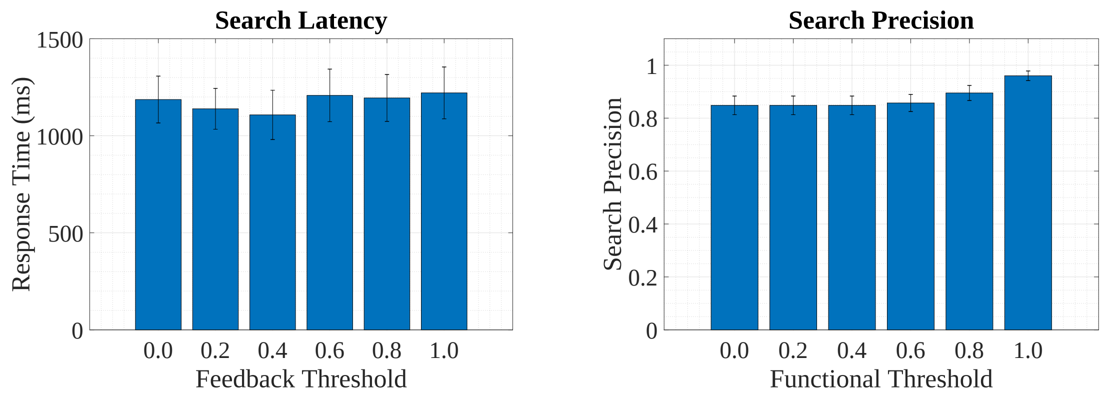
A.4 Heuristic Planner Parameters

uDiscovery has additional parameters for the heuristic planner introduced in the Section 3.4.3. These parameters are the $feedbackThreshold$ that is used to validate a discovered link between services against the consumers' feedback, the $functionalThreshold$ that is used to evaluate the relevance of a discovered plan with regard to the functional requirements, and the K value that is used to select the top-k plans to be explored. Table A.4 presents the design of the experiments performed on the IoT test bed (Section 5.3). The experiments' parameters are the feedback threshold, the functional threshold, and the K value. These parameters are combined in 3 scenarios. The first scenario has a variable feedback threshold, the second scenario has a variable functional threshold, and the third scenario has a variable K value. Each scenario is executed with 10000 services in the distributed repository, and request of plan length 1.

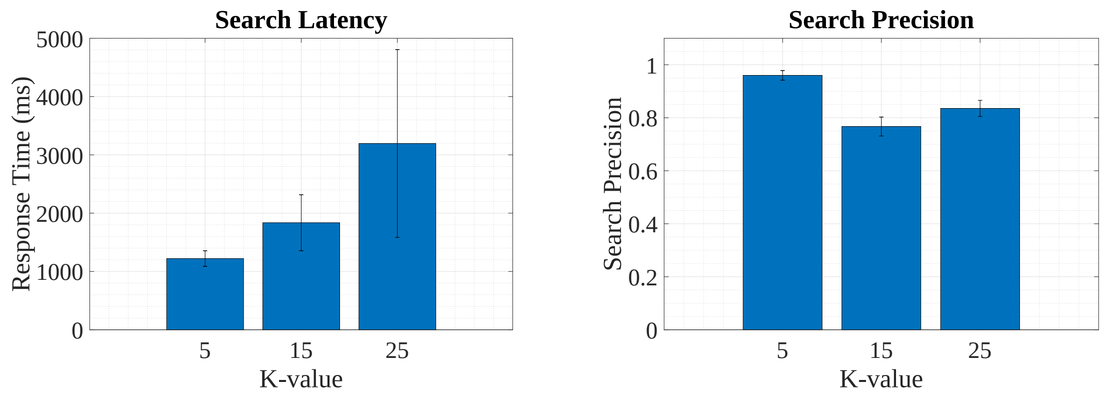
Figure A.19 shows the results of each scenario with regard to the search latency and precision. Each row corresponds to a scenario and each column corresponds to a metric. Figure A.19a shows the impact of the $feedbackThreshold$. The feedback threshold does not have an impact in the search latency as the *uDiscovery* keeps a constant response time regardless the threshold variability. But, this threshold has impact on the search precision. The planner reaches the highest search precision when the feedback threshold is equal to 1. Search precision is affected by lower threshold values and varies from 0.4, when the threshold is 0.0, to 0.6, when the threshold is 0.8. The feedback threshold affects the $uDiscovery's$ search precision because the planner validates services relations that have introduced non-relevant plans, when the threshold is lower than 1.0.



(a) Scenario 1 - Variable Feedback Threshold

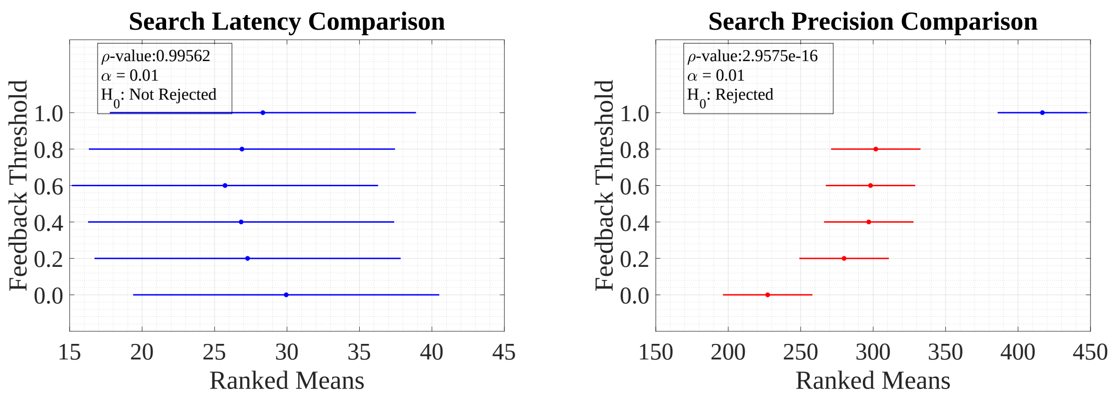


(b) Scenario 2 - Variable Functional Threshold

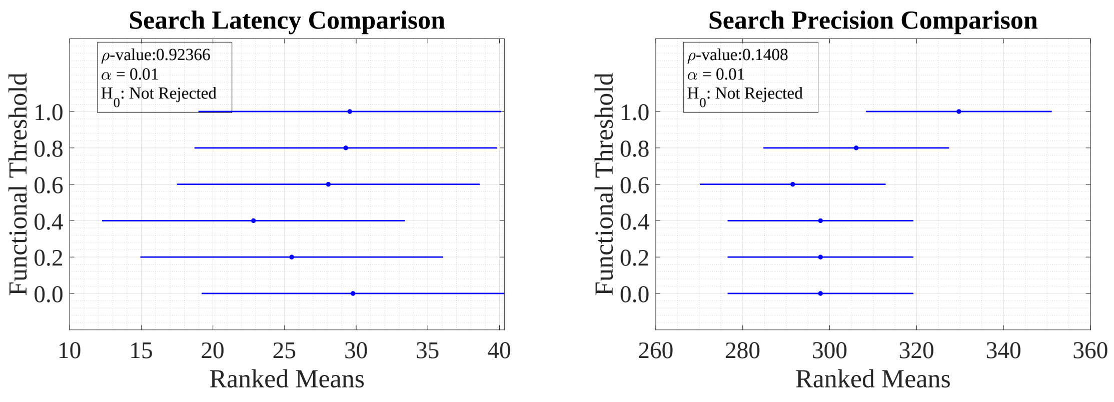


(c) Scenario 3 - K value Threshold

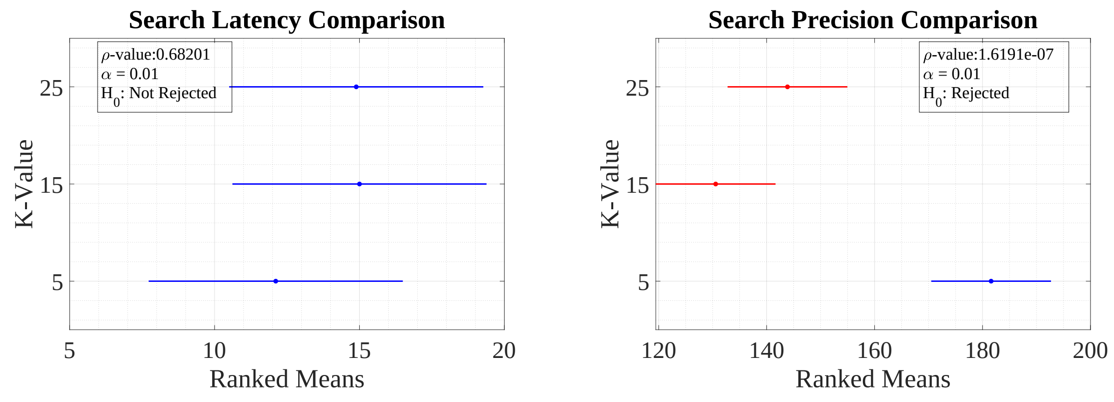
Figure A.19: *uDiscovery* Planner: Results Experiment Scenarios.



(a) Scenario 1 - Comparison of Ranked Means for Feedback Threshold



(b) Scenario 2 - Comparison of Ranked Means for Functional Threshold



(c) Scenario 3 - Comparison of Ranked Means for K value Threshold

Figure A.20: *uDiscovery* Planner: Comparison of Scenarios Ranked Means.

The functional threshold does not have a clear impact on the response time, but has a small impact in the search precision (Figure A.19b). *uDiscovery* keeps a constant response time regardless the threshold values as in the previous scenario. *uDiscovery* reaches the highest precision when the functional threshold is 1.0. The planner has a better accuracy when the threshold is 1.0 because it applies the more strict service matchmaking methods. Figure A.19c shows the impact of the K-value on the planner performance. The K-value impacts the response time because the planner explores less services when K is low. *uDiscovery* has the lowest latency when K is equal to 5. This K value does not have a clear impact on the search precision, *uDiscovery* is less likely to fail when K is low because it explores less alternatives.

Figure A.20 shows the results of the statistical analysis of the scenarios results to confirm previous observations. The $feedbackThreshold$ does not have a significant influence in the search latency, but it has significant and positive impact in the search precision when its value is 1.0. The $functionalThreshold$ does not have significant influence on any metric. However, it improves the search precision when its value is also 1.0. The K value does not have a significant impact in the search latency, although there were a clear difference in the previous study. But, it has a significant and positive impact in the search precision when K is equal to 5. The selected values are 1.0 for feedback and functional thresholds, and 5 for the K value.

Bibliography

[Al-Fuqaha et al., 2015] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols and Applications. *IEEE Communications Surveys & Tutorials*, 17(4):1–1.

[Albalas et al., 2017] Albalas, F., Mardini, W., and Al-Soud, M. (2017). Aft: Adaptive fibonacci-based tuning protocol for service and resource discovery in the internet of things. In *Fog and Mobile Edge Computing (FMEC), 2017 Second International Conference on*, pages 177–182. IEEE.

[Ammar et al., 2018] Ammar, M., Russello, G., and Crispo, B. (2018). Internet of things: A survey on the security of iot frameworks. *Journal of Information Security and Applications*, 38:8–27.

[Andreini et al., 2011] Andreini, F., Crisciani, F., Cicconetti, C., and Mambrini, R. (2011). A scalable architecture for geo-localized service access in smart cities. In *2011 Future Network & Mobile Summit*, pages 1–8. IEEE.

[Athanasopoulos, 2017] Athanasopoulos, D. (2017). Self-adaptive service organization for pragmatics-aware service discovery. In *Services Computing (SCC), 2017 IEEE International Conference on*, pages 164–171. IEEE.

[Atzori et al., 2011] Atzori, L., Iera, A., and Morabito, G. (2011). Siot: Giving a social structure to the internet of things. *IEEE communications letters*, 15(11):1193–1195.

[Baek and Ko, 2017] Baek, K.-D. and Ko, I.-Y. (2017). Spatially cohesive service discovery and dynamic service handover for distributed iot environments. In *International Conference on Web Engineering*, pages 60–78. Springer.

[Baker et al., 2017] Baker, T., Asim, M., Tawfik, H., Aldawsari, B., and Buyya, R. (2017). An energy-aware service composition algorithm for multiple cloud-based iot applications. *Journal of Network and Computer Applications*, pages 96–108.

[Bharti et al., 2018] Bharti, M., Kumar, R., and Saxena, S. (2018). Clustering-based resource discovery on internet-of-things. *International Journal of Communication Systems*, 31(5):e3501.

[Borgia, 2014] Borgia, E. (2014). The internet of things vision: Key features, applications and open issues. *Computer Communications*, 54:1–31.

[Boubiche et al., 2018] Boubiche, S., Boubiche, D. E., Bilami, A., and Toral-Cruz, H. (2018). Big data challenges and data aggregation strategies in wireless sensor networks. *IEEE Access*, 6:20558–20571.

[Bovet and Hennebert, 2014] Bovet, G. and Hennebert, J. (2014). Distributed semantic discovery for web-of-things enabled smart buildings. In *2014 6th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE.

[Bröring et al., 2016] Bröring, A., Datta, S. K., and Bonnet, C. (2016). A categorization of discovery technologies for the internet of things. In *Proceedings of the 6th International Conference on the Internet of Things*, pages 131–139. ACM.

[Butt et al., 2013] Butt, T. A., Phillips, I., Guan, L., and Oikonomou, G. (2013). Adaptive and context-aware service discovery for the internet of things. In *Internet of things, smart spaces, and next generation networking*, pages 36–47. Springer.

[Cabrera et al., 2017] Cabrera, C., Palade, A., and Clarke, S. (2017). An evaluation of service discovery protocols in the internet of things. In *Proceedings of the Symposium on Applied Computing*, pages 469–476. ACM.

[Cassar et al., 2014] Cassar, G., Barnaghi, P., and Moessner, K. (2014). Probabilistic matchmaking methods for automated service discovery. *IEEE Transactions on Services Computing*, 7(4):654–666.

[Chakraborty et al., 2006] Chakraborty, D., Joshi, A., Yesha, Y., and Finin, T. (2006). Toward distributed service discovery in pervasive computing environments. *IEEE Transactions on Mobile computing*, 5(2):97–112.

[Chen et al., 2016] Chen, N., Cardozo, N., and Clarke, S. (2016). Goal-driven service composition in mobile and pervasive computing. *IEEE Transactions on Services Computing*.

[Chen and Clarke, 2014] Chen, N. and Clarke, S. (2014). A dynamic service composition model for adaptive systems in mobile computing environments. In *International Conference on Service-Oriented Computing*, pages 93–107. Springer.

[Cheong et al., 2017] Cheong, P. Y., Aggarwal, D., Hanne, T., and Dornberger, R. (2017). Variation of ant colony optimization parameters for solving the travelling salesman problem. In *2017 IEEE 4th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, pages 60–65. IEEE.

[Ciortea et al., 2016] Ciortea, A., Boissier, O., Zimmermann, A., and Florea, A. M. (2016). Responsive decentralized composition of service mashups for the internet of things. In *Proceedings of the 6th International Conference on the Internet of Things*, pages 53–61. ACM.

[Cirani et al., 2014] Cirani, S., Davoli, L., Ferrari, G., Léone, R., Medagliani, P., Picone, M., and Veltri, L. (2014). A scalable and self-configuring architecture for service discovery in the internet of things. *IEEE Internet of Things Journal*, 1(5):508–521.

[Cirani et al., 2018] Cirani, S., Ferrari, G., Picone, M., and Veltri, L. (2018). *Internet of Things: Architectures, Protocols and Standards.* John Wiley & Sons.

[Corbellini et al., 2017] Corbellini, A., Godoy, D., Mateos, C., Zunino, A., and Lizarralde, I. (2017). Mining social web service repositories for social relationships to aid service discovery. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 75–79. IEEE.

[Dasgupta et al., 2014a] Dasgupta, S., Aroor, A., Shen, F., and Lee, Y. (2014a). Smartspace: Multiagent based distributed platform for semantic service discovery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(7):805–821.

[Dasgupta et al., 2014b] Dasgupta, S., Aroor, A., Shen, F., and Lee, Y. (2014b). Smartspace: Multiagent based distributed platform for semantic service discovery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(7):805–821.

[Datta et al., 2015] Datta, S. K., Da Costa, R. P. F., and Bonnet, C. (2015). Resource discovery in Internet of Things: Current trends and future standardization aspects. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 542–547. IEEE.

[del Val et al., 2014] del Val, E., Rebollo, M., and Botti, V. (2014). Combination of self-organization mechanisms to enhance service discovery in open systems. *Information Sciences*, 279:138–162.

[Deng et al., 2017] Deng, S., Huang, L., Taheri, J., Yin, J., Zhou, M., and Zomaya, A. Y. (2017). Mobility-aware service composition in mobile communities. *IEEE Transactions on*

Systems, Man, and Cybernetics: Systems, pages 555–568.

[D'Mello et al., 2011] D'Mello, D. A., Ananthanarayana, V., and Salian, S. (2011). A review of dynamic web service composition techniques. In *International Conference on Computer Science and Information Technology*, pages 85–97. Springer.

[Ebrahimi et al., 2015] Ebrahimi, M., Shafieibavani, E., Wong, R. K., and Chi, C.-H. (2015). A new meta-heuristic approach for efficient search in the internet of things. In *2015 IEEE International Conference on Services Computing*, pages 264–270. IEEE.

[Ebrahimi et al., 2017] Ebrahimi, M., ShafieiBavani, E., Wong, R. K., Fong, S., and Fiaidhi, J. (2017). An adaptive meta-heuristic search for the internet of things. *Future Generation Computer Systems*, 76:486–494.

[Fathy et al., 2017] Fathy, Y., Barnaghi, P., and Tafazolli, R. (2017). Distributed spatial indexing for the internet of things data management. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1246–1251. IEEE.

[Fathy et al., 2018] Fathy, Y., Barnaghi, P., and Tafazolli, R. (2018). Large-scale indexing, discovery, and ranking for the internet of things (iot). *ACM Computing Surveys (CSUR)*, 51(2):29.

[Fredj et al., 2013] Fredj, S. B., Boussard, M., Kofman, D., and Noirie, L. (2013). A scalable iot service search based on clustering and aggregation. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 403–410. IEEE.

[Fredj et al., 2014] Fredj, S. B., Boussard, M., Kofman, D., and Noirie, L. (2014). Efficient semantic-based iot service discovery mechanism for dynamic environments. In *Personal, Indoor, and Mobile Radio Communication (PIMRC), 2014 IEEE 25th Annual International Symposium on*, pages 2088–2092. IEEE.

[Georgievski and Aiello, 2017] Georgievski, I. and Aiello, M. (2017). Automated planning for ubiquitous computing. *ACM Computing Surveys (CSUR)*, 49(4):63.

[Girolami et al., 2015a] Girolami, M., Chessa, S., and Caruso, A. (2015a). On service discovery in mobile social networks: Survey and perspectives. *Computer Networks*, 88:51–71.

[Girolami et al., 2015b] Girolami, M., Chessa, S., and Ferro, E. (2015b). Discovery of services in smart cities of mobile social users. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 210–215. IEEE.

[Görgü et al., 2017] Görgü, L., Kroon, B., O'Grady, M. J., Yılmaz, Ö., and O'Hare, G. M. (2017). Sensor discovery in ambient iot ecosystems. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–12.

[Guerrero-Contreras et al., 2017] Guerrero-Contreras, G., Garrido, J. L., Balderas-Diaz, S., and Rodríguez-Domínguez, C. (2017). A context-aware architecture supporting service availability in mobile cloud computing. *IEEE Transactions on Services Computing*, 10(6):956–968.

[Hamidouche et al., 2018] Hamidouche, R., Aliouat, Z., and Gueroui, A. M. (2018). Bio-inspired vs classical solutions to overcome the iot challenges. In *2018 3rd Cloudification of the Internet of Things (CIoT)*, pages 1–7. IEEE.

[Han and Crespi, 2017] Han, S. N. and Crespi, N. (2017). Semantic service provisioning for smart objects: Integrating iot applications into the web. *Future Generation Computer Systems*.

[He et al., 2013] He, Q., Yan, J., Yang, Y., Kowalczyk, R., and Jin, H. (2013). A decentralized service discovery approach on peer-to-peer networks. *IEEE Transactions on Services Computing*, 6(1):64–75.

[Hoseinitabatabaei et al., 2018] Hoseinitabatabaei, S. A., Fathy, Y., Barnaghi, P., Wang, C., and Tafazolli, R. (2018). A novel indexing method for scalable iot source lookup. *IEEE Internet of Things Journal*, 5(3):2037–2054.

[Huber et al., 2016] Huber, S., Seiger, R., Kuehnert, A., and Schlegel, T. (2016). Using semantic queries to enable dynamic service invocation for processes in the internet of things. In *Semantic Computing (ICSC), 2016 IEEE Tenth International Conference on*, pages 214–221. IEEE.

[Hussein et al., 2017] Hussein, D., Han, S. N., Lee, G. M., Crespi, N., and Bertin, E. (2017). Towards a dynamic discovery of smart services in the social internet of things. *Computers & Electrical Engineering*, 58:429–443.

[Jara et al., 2014] Jara, A. J., Lopez, P., Fernandez, D., Castillo, J. F., Zamora, M. A., and Skarmeta, A. F. (2014). Mobile digcovery: discovering and interacting with the world through the internet of things. *Personal and ubiquitous computing*, 18(2):323–338.

[Jo et al., 2015] Jo, H.-J., Kwon, J.-H., and Ko, I.-Y. (2015). Distributed service discovery in mobile iot environments using hierarchical bloom filters. In *International Conference on Web Engineering*, pages 498–514. Springer.

[Khodadadi et al., 2015] Khodadadi, F., Dastjerdi, A. V., and Buyya, R. (2015). Simurgh: A framework for effective discovery, programming, and integration of services exposed in iot. In *2015 International Conference on Recent Advances in Internet of Things (RIoT)*, pages 1–6. IEEE.

[Klein and Bernstein, 2004] Klein, M. and Bernstein, A. (2004). Toward high-precision service retrieval. *IEEE Internet Computing*, pages 30–36.

[Klusch, 2014] Klusch, M. (2014). Service discovery. In *Encyclopedia of Social Network Analysis and Mining*, pages 1707–1717. Springer.

[Klusch et al., 2015] Klusch, M., Kapahnke, P., Schulte, S., Lecue, F., and Bernstein, A. (2015). Semantic web service search: a brief survey. *KI-Künstliche Intelligenz*, pages 1–9.

[Klusch et al., 2016] Klusch, M., Kapahnke, P., Schulte, S., Lecue, F., and Bernstein, A. (2016). Semantic web service search: a brief survey. *KI-Künstliche Intelligenz*, 30(2):139–147.

[Kovacevic et al., 2010] Kovacevic, A., Ansari, J., and Mahonen, P. (2010). Nanosd: A flexible service discovery protocol for dynamic and heterogeneous wireless sensor networks. In *Mobile Ad-hoc and Sensor Networks (MSN), 2010 Sixth International Conference on*, pages 14–19. IEEE.

[Kozlov et al., 2012] Kozlov, D., Veijalainen, J., and Ali, Y. (2012). Security and privacy threats in iot architectures. In *Proceedings of the 7th International Conference on Body Area Networks*, pages 256–262. ICST (Institute for Computer Sciences, Social-Informatics and ....

[Kumar and Satyanarayana, 2016] Kumar, V. V. and Satyanarayana, N. (2016). Self-adaptive semantic classification using domain knowledge and web usage log for web service discovery. *International Journal of Applied Engineering Research*, 11(6):4618–4622.

[Lalanda et al., 2013] Lalanda, P., McCann, J. A., and Diaconescu, A. (2013). Autonomic computing principles design and implementation.

[Lee et al., 2007] Lee, C., Ko, S., Lee, S., Lee, W., and Helal, S. (2007). Context-aware service composition for mobile network environments. *Ubiquitous Intelligence and Computing*, pages 941–952.

[Lee and Lee, 2018] Lee, D. and Lee, H. (2018). Iot service classification and clustering for integration of iot service platforms. *The Journal of Supercomputing*, 74(12):6859–6875.

[Lemos et al., 2016] Lemos, A. L., Daniel, F., and Benatallah, B. (2016). Web service composition: a survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33.

[Li et al., 2017] Li, J., Bai, Y., Zaman, N., and Leung, V. C. (2017). A decentralized trustworthy context and qos-aware service discovery framework for the internet of things. *IEEE Access*, 5:19154–19166.

[Li et al., 2017] Li, Z., Song, Y., and Bi, J. (2017). Tssd: Exploiting temporal-spatial correlation for service discovery in mobile social networking. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–7.

[Liu et al., 2016] Liu, M., Xu, Y., Hu, H., and Mohammed, A.-W. (2016). Semantic agent-based service middleware and simulation for smart cities. *Sensors*, 16(12):2200.

[Liu et al., 2019] Liu, Y., Zhu, T., Jiang, Y., and Liu, X. (2019). Service matchmaking for internet of things based on probabilistic topic model. *Future Generation Computer Systems*, 94:272–281.

[Loser et al., 2007] Loser, A., Staab, S., and Tempich, C. (2007). Semantic social overlay networks. *IEEE Journal on selected areas in communications*, 25(1):5–14.

[Lunardi et al., 2015] Lunardi, W. T., de Matos, E., Tiburski, R., Amaral, L. A., Marczak, S., and Hessel, F. (2015). Context-based search engine for industrial iot: Discovery, search, selection, and usage of devices. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–8. IEEE.

[Manning et al., 2010] Manning, C., Raghavan, P., and Schütze, H. (2010). Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103.

[Maymounkov and Mazieres, 2002] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer.

[McKight and Najab, 2010] McKight, P. E. and Najab, J. (2010). Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[Mokhtar et al., 2007] Mokhtar, S. B., Georgantas, N., and Issarny, V. (2007). Cocoa: Conversation-based service composition in pervasive computing environments with qos support. *Journal of Systems and Software*, 80(12):1941–1955.

[Mokhtar et al., 2010] Mokhtar, S. B., Raverdy, P.-G., Urbieta, A., and Cardoso, R. S. (2010). Interoperable semantic and syntactic service discovery for ambient computing environments. *International Journal of Ambient Computing and Intelligence (IJACI)*, 2(4):13–32.

[Nedos et al., 2009] Nedos, A., Singh, K., Cunningham, R., and Clarke, S. (2009). Probabilistic discovery of semantically diverse content in manets. *IEEE Transactions on Mobile Computing*, 8(4):544–557.

[Nesi et al., 2016] Nesi, P., Badii, C., Bellini, P., Cenni, D., Martelli, G., and Paolucc, M. (2016). Km4city smart city api: an integrated support for mobility services. In *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*, pages 1–8. IEEE.

[Nguyen et al., 2017] Nguyen, B. M., Hoang, H.-N. Q., Hluchỳ, L., Vu, T. T., and Le, H. (2017). Multiple peer chord rings approach for device discovery in iot environment. *Procedia Computer Science*, 110:125–134.

[O'Toole et al., 2017] O'Toole, E., Nallur, V., and Clarke, S. (2017). Decentralised detection of emergence in complex adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 12(1):4:1–4:31.

[Paganelli and Parlanti, 2012] Paganelli, F. and Parlanti, D. (2012). A dht-based discovery service for the internet of things. *Journal of Computer Networks and Communications*, 2012.

[Palade et al., 2018] Palade, A., Cabrera, C., White, G., and Clarke, S. (2018). Stigmergic service composition and adaptation in mobile environments. In *International Conference on Service-Oriented Computing*, pages 618–633. Springer.

[Papazoglou and Georgakopoulos, 2003] Papazoglou, M. P. and Georgakopoulos, D. (2003). Service-oriented computing. *Communications of the ACM*, 46(10):25–28.

[Pattar et al., 2018] Pattar, S., Buyya, R., Venugopal, K., Iyengar, S., and Patnaik, L. (2018). Searching for the iot resources: Fundamentals, requirements, comprehensive review, and future directions. *IEEE Communications Surveys & Tutorials*, 20(3):2101–2132.

[Perera et al., 2014a] Perera, C., Jayaraman, P. P., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014a). Context-aware dynamic discovery and configuration of 'things' in smart environments. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 215–241. Springer.

[Perera and Vasilakos, 2016] Perera, C. and Vasilakos, A. V. (2016). A knowledge-based resource discovery for internet of things. *Knowledge-Based Systems*, 109:122–136.

[Perera et al., 2014b] Perera, C., Zaslavsky, A., Liu, C. H., Compton, M., Christen, P., and Georgakopoulos, D. (2014b). Sensor search techniques for sensing as a service architecture for the internet of things. *IEEE Sensors Journal*, 14(2):406–420.

[Petrolo et al., 2016a] Petrolo, R., Bonifacio, S. G., Loscri, V., and Mitton, N. (2016a). The discovery of relevant data-sources in a smart city environment. In *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*, pages 1–5. IEEE.

[Petrolo et al., 2016b] Petrolo, R., Loscri, V., and Mitton, N. (2016b). Cyber-physical objects as key elements for a smart cyber-city. In *Management of Cyber Physical Objects in the Future Internet of Things*, pages 31–49. Springer.

[Piro et al., 2014] Piro, G., Cianci, I., Grieco, L., Boggia, G., and Camarda, P. (2014). Information centric services in Smart Cities. *Journal of Systems and Software*, 88:169–188.

[Quevedo et al., 2016] Quevedo, J., Antunes, M., Corujo, D., Gomes, D., and Aguiar, R. L. (2016). On the application of contextual iot service discovery in information centric networks. *Computer Communications*, 89:117–127.

[Rapti et al., 2016] Rapti, E., Karageorgos, A., Houstis, C., and Houstis, E. (2016). Decentralized service discovery and selection in internet of things applications based on artificial potential fields. *Service Oriented Computing and Applications*, pages 1–12.

[Richerzhagen et al., 2015] Richerzhagen, B., Stingl, D., Rückert, J., and Steinmetz, R. (2015). Simonstrator: Simulation and prototyping platform for distributed mobile applications. In *Proceedings of the 8th International Conference on Simulation Tools and Techniques*, pages 99–108, ICST, Brussels, Belgium, Belgium.

[Rodriguez-Mier et al., 2016] Rodriguez-Mier, P., Pedrinaci, C., Lama, M., and Mucientes, M. (2016). An integrated semantic web service discovery and composition framework. *IEEE transactions on services computing*, pages 537–550.

[Rubio et al., 2016] Rubio, G., Martínez, J. F., Gómez, D., and Li, X. (2016). Semantic registration and discovery system of subsystems and services within an interoperable coordination platform in smart cities. *Sensors*, 16(7):955.

[Sikri, 2019] Sikri, M. (2019). An adaptive and scalable framework for automated service discovery. *Service Oriented Computing and Applications*, pages 1–13.

[Sivrikaya et al., 2019] Sivrikaya, F., Ben-Sassi, N., Dang, X.-T., Görür, O. C., and Kuster, C. (2019). Internet of smart city objects: A distributed framework for service discovery and composition. *IEEE Access*.

[Stoica et al., 2003] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32.

[Stolikj et al., 2016] Stolikj, M., Lukkien, J. J., Cuijpers, P. J., and Buchina, N. (2016). Nomadic service discovery in smart cities. In *Smart Cities and Homes*, pages 59–90. Elsevier.

[Tanganelli et al., 2017] Tanganelli, G., Vallati, C., and Mingozzi, E. (2017). A fog-based distributed look-up service for intelligent transportation systems. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2017 IEEE 18th International Symposium on*, pages 1–6. IEEE.

[Teixeira et al., 2011] Teixeira, T., Hachem, S., Issarny, V., and Georgantas, N. (2011). Service oriented middleware for the internet of things: A perspective. In Abramowicz, W., Llorente, I. M., Surridge, M., Zisman, A., and Vayssière, J., editors, *Towards a Service-Based Internet*, pages 220–229, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Thiagarajan et al., 2002] Thiagarajan, R. K., Srivastava, A. K., Pujari, A. K., and Bulusu, V. K. (2002). Bpml: A process modeling language for dynamic business models. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2002.(WECWIS 2002). Proceedings. Fourth IEEE International Workshop on*, pages 222–224. IEEE.

[Tran et al., 2017] Tran, N. K., Sheng, Q. Z., Babar, M. A., and Yao, L. (2017). Searching the web of things: State of the art, challenges, and solutions. *ACM Computing Surveys*

(CSUR), 50(4):55.

[Tzortzis and Spyrou, 2016] Tzortzis, G. and Spyrou, E. (2016). A semi-automatic approach for semantic iot service composition. In *Workshop on Artificial Intelligence and Internet of Things in conjunction with SETN*.

[Urbieta et al., 2017] Urbieta, A., González-Beltrán, A., Mokhtar, S. B., Hossain, M. A., and Capra, L. (2017). Adaptive and context-aware service composition for iot-based smart cities. *Future Generation Computer Systems*.

[Urbieta et al., 2015] Urbieta, A., González-Beltrán, A., Mokhtar, S. B., Parra, J., Capra, L., Hossain, M. A., Alelaiwi, A., and Vázquez, J. I. (2015). Hybrid service matchmaking in ambient assisted living environments based on context-aware service modeling. *Cluster Computing*, 18(3):1171–1188.

[Walsh, 2002] Walsh, A. E. (2002). *Uddi, Soap, and WSDL: the web services specification reference book*. Prentice Hall Professional Technical Reference.

[Wang and Chow, 2016] Wang, E. and Chow, R. (2016). What can i do here? IoT service discovery in smart cities. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE.

[Wang et al., 2015] Wang, W., De, S., Cassar, G., and Moessner, K. (2015). An experimental study on geospatial indexing for sensor service discovery. *Expert Systems with Applications*, 42(7):3528–3538.

[Wanigasekara et al., 2016] Wanigasekara, N., Schmalfuss, J., Carlson, D., and Rosenblum, D. S. (2016). A bandit approach for intelligent iot service composition across heterogeneous smart spaces. In *Proceedings of the 6th International Conference on the Internet of Things*, pages 121–129. ACM.

[White et al., 2017] White, G., Nallur, V., and Clarke, S. (2017). Quality of service approaches in iot: A systematic mapping. *Journal of Systems and Software*, 132:186–203.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

[Wu et al., 2015] Wu, Y., Yan, C., Liu, L., Ding, Z., and Jiang, C. (2015). An adaptive multilevel indexing method for disaster service discovery. *IEEE Transactions on Computers*, 64(9):2447–2459.

[Xia et al., 2019] Xia, H., Hu, C.-q., Xiao, F., Cheng, X.-g., and Pan, Z.-k. (2019). An efficient social-like semantic-aware service discovery mechanism for large-scale internet of things. *Computer Networks*.

[Yuan et al., 2018] Yuan, B., Liu, L., and Antonopoulos, N. (2018). Efficient service discovery in decentralized online social networks. *Future Generation Computer Systems*, 86:775–791.

[Zhao et al., 2017a] Zhao, D., Zhou, Z., Ning, K., Duan, Y., and Zhang, L.-J. (2017a). An energy-aware service composition mechanism in service-oriented wireless sensor networks. In *Internet of Things (ICIOT), 2017 IEEE International Congress on*, pages 89–96. IEEE.

[Zhao et al., 2015] Zhao, S., Zhang, Y., Yu, L., Cheng, B., Ji, Y., and Chen, J. (2015). A multidimensional resource model for dynamic resource matching in internet of things. *Concurrency and Computation: Practice and Experience*, 27(8):1819–1843.

[Zhao et al., 2017b] Zhao, Y., Wang, S., Zou, Y., Ng, J., and Ng, T. (2017b). Automatically learning user preferences for personalized service composition. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 776–783. IEEE.

[Zhou et al., 2016] Zhou, Y., De, S., Wang, W., and Moessner, K. (2016). Search techniques for the web of things: A taxonomy and survey. *Sensors*, 16(5):600.

[Zikria et al., 2018] Zikria, Y. B., Afzal, M. K., Ishmanov, F., Kim, S. W., and Yu, H. (2018). A survey on routing protocols supported by the contiki internet of things operating system. *Future Generation Computer Systems*, 82:200–219.

[Zisman et al., 2013] Zisman, A., Spanoudakis, G., Dooley, J., and Siveroni, I. (2013). Proactive and reactive runtime service discovery: A framework and its evaluation. *IEEE Transactions on Software Engineering*, 39(7):954–974.