Context-dependent Exogenous Coordination for Building Large Scale, Dynamic Fog Computing Applications

by

Nam Ky Giang

B. Engineering, Hanoi University of Science and Technology, 2010
M. Computer Science, Korea Advanced Institute of Science and Technology, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2019

© Nam Ky Giang, 2019
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Context-dependent Exogenous Coordination for Building Large Scale, Dynamic Fog Computing Applications

Submitted by: Nam Ky Giang in partial fulfilment of the requirements for the degree of: Doctor of Philosophy in: Computer Engineering

Examining Committee:

Victor C.M. Leung, Electrical and Computer Engineering

Co-supervisor

Rodger Lea, School of Computing and Communication, Lancaster University

Co-supervisor

Sathish Gopalakrishnan, Electrical and Computer Engineering

Supervisory Committee Member

Jose Marti, Electrical and Computer Engineering

University Examiner

Alan Wagner, Computer Science

University Examiner

Kui Wu, Computer Science

External Examiner
Abstract

Recently, fog computing has emerged as a new system model for applications that are data-intensive or delay-sensitive. Thanks to the computing infrastructure that is closer to the network edge, communication cost and response time are significantly reduced. While promising, fog computing has its unique challenges, particularly due to its large number of computing elements, their geographic distribution and the dynamic nature of the edge network.

First, the large scale complexity of fog computing poses new challenges in the application development process. By analysing various application models, we provide a new taxonomy with important trade-offs that help fog computing developers to navigate their design space in building fog applications. From here, we found that exogenous coordination, where there is a clear separation of concerns between computation and communication activities, is a suitable approach in supporting the complexity of many fog computing scenarios.

Second, the geo-distribution of fog devices introduces new applications that depend on such devices’ physical context. While we found exogenous coordination to be a suitable approach, it is a software-focused concept that does not cater specifically to those hardware contexts. We propose to extend exogenous coordination with coordination primitives that help to express context-dependent fog application logic. Our proposal includes a clear separation of concerns between contextual and application data, context-dependent constraints for fog application components, and context-bounded communication cardinality among them.

Third, the dynamic nature of the edge network requires certain degrees of system moni-
toring and adaptation, which are resource consuming. To support our proposed coordination model in such a dynamic environment, we develop an incremental coordination technique that based on historical coordination activities to reduce resource consumption. In addition, we show that the coordination activity is not generally affected by the system dynamics with regard to the application’s overall performance. This is due to the orthogonality-by-design between coordination and computation aspects of the application.

Our platform has been implemented and made publicly available through an open source project, which has been evaluated by both industrial and academic researchers. A large scale, lab-based simulation has also been developed to testify the feasibility of the proposed model.
Lay Summary

The thesis contributes a taxonomy of application models for designing fog applications, a context-dependent coordination model and necessary system supports to realise many applications that are large scale and dynamic in nature. Fog computing is a computing infrastructure that involves devices across the edge network (e.g. smartphones), the access network (e.g. Wi-Fi routers, cell towers), and the cloud. The contributed taxonomy helps fog computing developers to navigate the design space for fog applications. It also shows the appropriateness of exogenous coordination concept in building many fog applications. The contributed coordination model takes into account the physical context of the fog devices; it provides context-dependent primitives to express a range of fog application requirements. The contributed coordination platform supports those primitives in the deployment of applications in large scale, dynamic fog computing infrastructure. By employing an incremental coordination technique that remembers the last coordinated result, the coordination overhead is significantly reduced.
Preface

Most of the work in this thesis has been done by the author during his PhD program at the University of British Columbia, under the supervision of his committee members. The application field trial in Fujisawa city has been carried out by Rodger Lea’s partner group at the Keio University, Japan, under the EU-Japan BigClout project.

Original contributions in Chapter 3 have been published in the proceeding of the ACM DIVANET 2019 conference. Original contributions in Chapter 4 have been published in the IEEE Access journal. Original contributions in Chapter 5 have been published in the Software: Practice and Experience journal.

The application field trial in Fujisawa, which is reported in Chapter 4, Section 4.5.2 has also been jointly published with Rodger Lea’s partner group at the Keio University, in the proceeding of Urb-IoT 2018 conference.

The requirements and challenges analysis in Chapter 2 and most of the literature survey in Chapter 6 have been published in several conferences and workshops, including the 2015 IEEE International Conference on the Internet of Things, the 12th Middleware Doctoral Symposium 2015, the Smart Cities 2016 workshop, and the DIVANet 2016 conference.

All the publications where the author is the first author are his original work with criticisms and help from the supervisory committee (Dr Rodger Lea and Prof. Victor Leung). Some of these publications also gather contributions from Michael Blackstock, one of the author’s colleagues. His contributions include suggestions, criticisms, as well as editing efforts.
The list of journal publications is as follows:


The list of other publications is as follows:


# Table of Contents

Abstract ................................................................. iii

Lay Summary ............................................................... V

Preface ................................................................. vi

Table of Contents ....................................................... ix

List of Tables ........................................................... xiv

List of Figures ............................................................ xv

Glossary ............................................................... xvii

Acknowledgements ......................................................... xviii

Dedication ............................................................... xx

1 Introduction .......................................................... 1

2 Background and Contributions ........................................ 7
   2.1 Motivation ....................................................... 7
   2.2 Application Scenarios ............................................ 10
   2.3 Application Characteristics ........................................ 11
2.3.1 System Scale .................................................. 12
2.3.2 Awareness of Physical Context ............................ 13
2.3.3 Dynamic Nature .............................................. 14

2.4 Research Challenges ........................................... 16
2.4.1 Large Scale, Geo-Distribution of Devices ............... 16
2.4.2 Dynamic Nature of the fog computing infrastructure .... 17

2.5 Literature Overview ............................................ 18

2.6 Thesis Objectives ............................................... 20
2.6.1 An Analysis of Various Application Models ............. 20
2.6.2 Context-dependent Exogenous Coordination in Building Fog Applications ............................................ 21
2.6.3 System Supports for Fog Applications ................... 22
2.6.4 Thesis Contributions ........................................... 23
2.6.5 Case Study-driven Research Methodology ............... 24

2.7 Summary .......................................................... 25

3 An analysis of Various Application Models: Toward Developing Large Scale, Dynamic Fog Applications with Exogenous Coordination ................. 27

3.1 Scope .............................................................. 28
3.1.1 Component scope ............................................. 30
3.1.2 Inter-Component scope ..................................... 30
3.1.3 Inter-Device scope ........................................... 31
3.1.4 Inter-Context scope ......................................... 31

3.2 Abstraction Level ................................................ 32
3.2.1 Goal-based .................................................... 34
3.2.2 Rule-based .................................................... 35
3.2.3 Component Cooperation .................................... 36
4 Context-dependent Exogenous Coordination for Developing Fog Applications 55

4.1 Contextual Data versus Application Data .......................... 57

4.2 Proposed Coordination Model for Fog Applications ............... 59

4.2.1 Participatory Coordination ..................................... 59

4.2.2 Workflow ...................................................... 60

4.3 Context-dependent Coordination Primitives ........................ 61

4.4 Communication Cardinality .......................................... 65

4.5 Evaluations ......................................................... 67

4.5.1 Distributed Node-RED Project as a Coordination Platform .... 67
5 Coordinating Large Scale, Dynamic Fog Computing Applications

5.1 Design Considerations

5.1.1 Large Scale Coordination

5.1.2 Coordination Overhead

5.2 Large Scale Coordination Problem Formalisation and Issues

5.3 Incremental Coordination for Dynamic Fog Applications

5.4 Implementation

5.4.1 Participating Devices

5.4.2 Coordination Medium

5.4.3 Coordination Platform

5.5 Evaluations

5.5.1 Omnet++ Simulation

5.5.2 Evaluation Metrics

5.5.3 Results and Discussions

5.5.4 Reflect on RQ3

5.6 Summary

6 Literature Survey

6.1 Related Application Models

6.1.1 Scope

6.1.2 Application Structure

6.1.3 Abstraction Level

6.1.4 Programming Models
List of Tables

Table 5.1 Omnet++ Simulation Configuration .......................... 100

Table 6.1 Comparison with Related Application Models .................. 123
## List of Figures

| Figure 2.1 | Data, Communication and Computation | 8 |
| Figure 2.2 | Typical Applications Leveraging Fog Computing | 9 |
| Figure 2.3 | Example of a Smart City Application | 10 |
| Figure 2.4 | Our class of fog applications | 11 |
| Figure 3.1 | An Analysis of Various Application Models | 29 |
| Figure 3.2 | Application scopes | 30 |
| Figure 3.3 | Abstraction Levels of Common Applications | 34 |
| Figure 3.4 | Application structures | 38 |
| Figure 3.5 | Communication models | 41 |
| Figure 3.6 | Programming models | 42 |
| Figure 3.7 | Analysed important design requirements for building our class of fog applications | 44 |
| Figure 3.8 | Exogenous Coordination in Programming | 50 |
| Figure 4.1 | Separation of concerns: contextual vs application data | 58 |
| Figure 4.2 | Fog application modeled as a dataflow graph | 60 |
| Figure 4.3 | Participatory fog applications design and deployment | 60 |
| Figure 4.4 | Context-bounded Communication Cardinality | 66 |
| Figure 4.5 | Building fog applications using the DNR platform | 69 |
Figure 4.6  Lab setup for our example fog application. ..........................  71
Figure 4.7  Fog applications design and development in DNR. ...............  74
Figure 4.8  Industry Example: wizr.com ...........................................  76

Figure 5.1  Coordination Problem Formalisation .................................  84
Figure 5.2  System Architecture .....................................................  88
Figure 5.3  Local Coordinators .....................................................  90
Figure 5.4  Local Coordinator: Deriving Coordination State .................  91
Figure 5.5  Omnet++ Simulation Model for Large Scale Fog Applications ....  99
Figure 5.6  Performance of a distributed coordination platform ............... 101
Figure 5.7  Effect of system dynamic to large scale coordination .......... 103
Figure 5.8  Stateless vs incremental coordination in fast moving scenario ... 103

Figure 6.1  Sehic et al.: Origin Toolkit [80] ....................................... 108
Figure 6.2  Hong et al.: Mobile Fog ............................................... 110
Figure 6.3  Cherrier et al.: D-Lite .................................................. 118
Figure 6.4  Bischoff et al.: A Compiler for the Smart Space .................. 119
Figure 6.5  Context-dependent Exogenous Coordination in Fog Computing . 128

Figure A.1  Influence on accuracy due to a spatial-temporal covariate shift. For models taught from data around commercial facilities, prediction accuracy is high in similar commercial facilities. The prediction accuracy decreases as the spatial features differ. .................................................  152

Figure A.2  Road damage detection application developed by CityFlow. ....  152
# Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligent</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CSP</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>DNR</td>
<td>Distributed Node-RED</td>
</tr>
</tbody>
</table>

**EXOGENOUS COORDINATION** coordination from the outside

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
</tr>
<tr>
<td>IOT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>QOS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SDN</td>
<td>Software-Defined Network</td>
</tr>
<tr>
<td>SUMO</td>
<td>Simulation of Urban MObility</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Networks</td>
</tr>
</tbody>
</table>
Acknowledgements

This work would not have been complete without the critical supervision of Dr. Rodger Lea, and the generous financial support of Prof. Victor Leung. Dr Lea taught me many things, from "The Craft of Research", an old book but highly recommended to researchers in all fields, to croquet, a beautiful yet cruelly competitive game. It is not overstating to say that I owe my critical thinking skills to his hands-on supervision. Prof. Leung on the other hand, provided me not only financial support that sustained me throughout the years, but also motivations and encouragements to solve hard problems.

I was lucky to have constant support from my dearest wife, Trang, during the difficult days and years, that keeps me on the road. I was also lucky to have constant distractions from my lovely little Felix, that help me to cool down and not become overclocked. It was an adventurous journey that we will never forget, thank you all for being a part of it.

While there are so many other people I need to thank, I need to single out several of them, who not only support me financially, but also a huge part of my journey as friends and mentors. A big thank to Donald and Sherry, who were supporting me from the earliest days of my program. It was a memorable experience working with you at Moovee Innovations Inc to try out a lot of new technologies and innovative solutions, and to make impacts to the world. Another big thank you to Mike, who helped me to brainstorm (and also shoot down) ideas from the very early days. I was really lucky to have Mike as one of my mentors in computer systems design, and also in preventing Starbuck cup drips\(^1\). The time spent

\(^1\)https://blog.thinkreliability.com/why-does-my-starbucks-cup-drip
at Sensetecnic Systems Inc was invaluable to my completion and also my career. Thank you very much Babak, my colleague and friend, who gave me tips and tricks of working out a PhD. Thank you very much Steve, my friend, who gave me lots of legal advice and entrepreneurial spirit.

Lastly, me and my wife would not have existed in the first place without being born and raised. Thank you very much, to our parents, Mr. Dong Giang, Mrs. Hat Le, Mr. Ngu Dinh, Mrs. Truc Tran, for bringing us into this life and raising us to this day and age. Your support (and of course, complaints) are the very crucial part of my completion.
for Trang
Chapter 1

Introduction

With the cost of hardware going down, embedded devices are being equipped with more and more computing resources. This trend results in the rapid rise of smart devices that have wireless connectivity and computation capability. At the same time, the core network infrastructure has been evolving toward a software-defined architecture with "softwarised" network functions. These advancements push more computing resources to the network edge instead of having them to stay in the cloud. This movement opens up a new computing paradigm called fog computing.

Unlike cloud computing, where computing resources are centralised in large data centres, fog computing leverages new computing capabilities at the core and the edge of the network for software deployment. Due to these new capabilities, it is possible to, in a cost-effective way, develop and deploy new classes of application that are data-intensive or delay-sensitive. The primary motivation is to process the ever-increasing amount of data, especially in many emerging Internet of Things (IoT) and Artificial Intelligent (AI) applications, near their source of origin instead of transporting them to a distant facility. It is worth noting that the fog computing infrastructure that we refer to in this thesis encompasses all the computing elements between the edge and the cloud, with a focus on the edge side.

Consider a smart city application that uses video streams from CCTV cameras and dash-
cams mounted on vehicles, to analyse the traffic condition. From here, certain tasks can be achieved, such as finding a suspect car, or control traffic signals. Because video data are bulky, it is expensive to transmit those data to a distant cloud or a centralised location (e.g. the transportation department) to be analysed. While we can manufacture highly capable cameras so that all the analysis tasks can be embedded on board, this is also not a desirable solution due to manufacturing cost, and the reduced reusability of such specialised systems. In this type of application, it is best to leverage the fog computing infrastructure for the deployment of the software so that "bulky" data streams can be processed near their origins.

In a broader perspective, this extends the traditional client-server application model to a more general edge-everything in between-cloud model where it involves more intermediary computation units between a data generating component (at the edge) and a final data processing component (at the cloud). This view seems similar to general distributed systems applications, where multiple computing elements work together to solve complex problems. However, there are notable differences, namely the larger scale of the deployment (by this we mean across city versus in data centres) and the tight coupling with physical context of the computing elements (e.g. computing elements deployed in different locations). This does not imply that general distributed system approaches cannot be exploited in building fog applications. In fact, message brokering, a fundamental technique in designing distributed systems, is also widely used to glue together fog computing elements.

Our interest here is, however, to think about a high-level application model that can be used to express a range of fog applications, for which, message brokering might play an essential role in the realisation of such model. This curiosity leads us to the central research questions of this thesis; namely:

- RQ1: How do fog applications differ from general distributed applications?

- RQ2: How can fog applications be developed with minimum requirements on the
developers to cope with the complexity of fog computing infrastructure (large scale and dynamic)?

- RQ3: How an application platform for fog computing can keep up with its large scale and dynamic nature in an efficient way while still maintaining the correctness of the application logic?

To compare with the present context, we have seen desktop applications, Internet applications, and mobile applications as examples of how computing applications have been evolving. With the advent of fog computing, we are left with an open question of how do such fog applications differ from the rest, and if we are going to adapt and exploit existing models and techniques, what are the challenges and solutions with regard to application models and system supports.

Our observation is that, due to its large scale, geo-distribution, many fog applications do express a coupling with the physical context in which the computing elements run (e.g. exploit sensing data only in a particular location). In contrast, common cloud-based applications running in data centres only focus on the computation activities only (e.g. functions, procedures, algorithms). That is, the underlying computing infrastructure in data centres remains highly transparent unless we are working with regional scaling and fail-overs, which are unrelated aspects to the applications’ logic.

Thus, our hypothesis in answering RQ1 (How do fog applications differ from general distributed applications?) is that fog applications distinguish themselves by the need to depend on the contextual data of the underlying fog devices. Furthermore, the complexity of fog applications (i.e. large scale and dynamic) requires a high level of decomposition where applications are constructed from off-the-shelf software components in a reusable and flexible way. This goes beyond the concept of modularity in software construction in that the software components that make up the applications can be independently developed (e.g. by different developers, using different languages).
To verify these hypotheses and to answer RQ1, we conduct an analysis over various application models in fog computing as well as related literature to develop a taxonomy for application models, where we draw the distinction between our class of fog application and the others.

Based on this analysis, we found the concept of exogenous coordination, where there is a clear separation of concerns between computation and coordination activities, appears to be the closest approach to support the decompositionality of fog applications. However, we also found that due to the focus on coordinating software components, exogenous coordination lacks support for context-dependent logic of many fog applications.

This leads us to our hypothesis in answering RQ2 (How can fog applications be developed with minimum requirements on the developers to cope with the complexity of fog computing infrastructure (large scale and dynamic)?), which is, exogenous coordination appears to be a suitable approach in realising a high-level application model for fog computing. Meanwhile, it also needs to be extended to abstract away the arduous characteristics of fog applications, namely, the dynamic nature of the system and its coupling with the hardware context.

To verify this hypothesis and to answer RQ2, we develop an exogenous coordination platform and several context-dependent primitives for expressing a range of fog application requirements. We use the platform to build two sample fog applications, one of which is a lab-based experiment while the other is a real-world field trial in Fujisawa, Japan. The experiments allow us to concretely verify that exogenous coordination is a very suitable approach in designing and developing our fog applications, and that context-dependent primitives further extend this approach to fully support the development of fog applications.

Based on our answers to RQ1 and RQ2, we proceed to answer our last research question RQ3 (How an application platform for fog can keep up with its large scale and dynamic nature in an efficient way while still maintaining the correctness of the application logic?).
Our hypothesis is that in coordinating a large number of fog devices in such a dynamic environment, we need a scalable and dynamic coordination layer. While this is straightforward when compared to distributed applications, we expect that the distinctive features of fog applications would need special treatments.

To answer this, we develop a simulation of a large scale fog scenario that encompasses both static and dynamic (mobile) fog devices in a large area. To coordinate the simulated fog devices given the context-dependent application logic, we use a large number of coordinators to collectively solve our coordination problem. The simulation allows us to study how the large scale and dynamic nature of fog devices affect the design of a coordination layer.

Interestingly, these research questions are coherent and are not independently developed, nor answered in arbitrary order. That is, by answering RQ1, we have a better understanding of how fog applications distinguish themselves. Based on this understanding, we have a narrower design space to choose from in order to sketch out the answer to RQ2. Lastly, only after answering RQ2, we know what needs to be done to realise such application model in large scale and dynamic fog computing infrastructure, which is the answer to RQ3.

This thesis is structured into 7 chapters. In Chapter 2 we give a more detailed background of the thesis, including the motivations, research challenges and our claimed contributions.

In Chapter 3, we develop an analysis of the notion of "application model" and draw the distinct features of our class of fog applications. From here, we show that coordination from the outside (EXOGENOUS COORDINATION) is a natural approach to develop various fog applications, and how it can be extended to fully support the development of our fog applications. This is the answer to RQ1.

In Chapter 4, we propose our context dependent coordination primitives to extend the concept of exogenous coordination to be applicable in developing our fog applications. We show that the primitives help to capture a range of high-level fog computing requirements into the application logic. Our open source project, the Distributed Node-RED has been
developed to support the evaluation of the primitives. Two sample applications scenarios are developed to show the feasibility of the approach. This is to answer RQ2.

In Chapter 5, we develop a coordination platform and use it to coordinate a simulated network of fog devices in large scale settings. We show that the optimal number of coordinators in a distributed coordination layer is not affected by the dynamic nature of the system due to the orthogonality-by-design between computation and coordination activities. We also show that by doing incremental coordination, this optimal number of coordinators is reduced by nearly 60% without significantly compromising the system performance. This is to answer RQ3.

Chapter 6 reports our literature survey and finally, Chapter 7 summarises the thesis and provides some discussions with regards to limitations and future research directions.

In summary, this thesis contributes an understanding of application models toward developing a class of fog applications that are dynamic and large scale. Exogenous coordination concept appears to be a suitable approach in realising such a model. The thesis builds upon this concept and extends it by catering specifically to the large scale and dynamic characteristics of our fog applications accordingly, context-dependent primitives and appropriate system supports are proposed. An application platform for building and running fog applications was developed to evaluate the hypotheses. It includes necessary programming abstracts to develop applications and system supports to deploy and maintain the applications in large scale and dynamic fog computing systems.
Chapter 2

Background and Contributions

2.1 Motivation

As the number of smart embedded devices increases, computing infrastructure is more intimately tied to the physical world, often capturing high volumes of real-world sensor data. We refer to this trend as the IoT paradigm, where more and more physical things are connected to the Internet and equipped with computation and sensing capabilities. It is estimated that by 2025, the number of deployed IoT devices could reach 1 trillion devices, contributing to as much as 11 per cent of the world economy [62]. This number of smart devices is also expected to produce more than 45 zettabyte real-time data by then [77].

Traditionally, cloud computing infrastructure has been used as the de facto backend for the IoT where computation is offloaded from constrained devices to the cloud [19]. This computation offloading involves transporting IoT data to the cloud where data transformation, analysis and storage take place. Due to the projected growth of the IoT sector and the intensity of the real-time data generated, cloud-based architecture is showing inefficiencies and shortcomings in many application scenarios that are data intensive or delay sensitive [97].

To better accommodate these emerging application scenarios, fog computing architec-
Figure 2.1: Data, Communication and Computation

A conceptual model has been proposed, which acts as a bridging architecture between the data-generating edge network and the resourceful cloud infrastructure. Fog computing, where computing resources are distributed closer to the edge network, greatly compliments the cloud computing by having part of the data-intensive computation done "in the fog". This significantly improves the consumption of network bandwidth as well as service latency [57] [17] [16].

To better understand the role of fog computing, Figure 2.1 illustrates the interplay between edge, fog and cloud computing in a range of computation activities, which are characterised by the relationship between the size of data and the cost of communication/computation.

Traditionally, the size of the data is directly proportional to the cost of computation. That is, "bigger" data usually needs more computation resources and vice versa. These computing activities are placed either in the cloud infrastructure or within the edge network depending on the size of their input data. This scenario is found in many traditional cloud or edge application architectures that are prevalent in today's computing infrastructure. For some applications, where the size of data generated by devices at the edge network is high, the cost to transport data to the cloud is sometimes prohibitive. Our observation is that, for many applications, there are simple data preprocessing steps that do not require substantial
computation resources but can significantly reduce the size of the data to be processed by multiple analytic modules. For example, in applications that process high-frequency sensing data, preprocessing steps could include lowering the frequency by applying moving average over the data stream. In other applications that do scene classification over video streams, preprocessing steps could include recolouring, resizing the images before sending out for more computation demanding tasks (e.g. classification).

Fog computing, where a combination of computing resources across the edge network and the cloud, is argued to be a natural architectural choice for these application scenarios [17]. This is shown in Figure 2.2. While the preprocessing steps work with a large volume of data, they do not require substantial computation resources. These computation activities can be carried out in less-capable devices near the edge network. The results of these preprocessing steps in turn, which might need more computation resources, are small in size, so do not consume a lot of network bandwidth. Thus, these computation activities can then be placed in the cloud without incurring expensive communication costs. This approach also means the application is now decomposed into several subcomponents so that they can be distributed to appropriate computing infrastructure.

In the following sections, we describe some application scenarios that illustrate these observations and help us to understand more about the motivation of the fog computing

Figure 2.2: Typical Applications Leveraging Fog Computing
2.2 Application Scenarios

Consider an example smart city application as follow. The application takes advantages of dashcams mounted on cars and computing infrastructure across the city to do vision processing and AI tasks. For example, identifying the locations of street markers that need to be repainted [47] or looking for a suspect car to help the police. The dashcams could be mounted on city buses and the video streams might go through several processing steps such as sizing, recolouring, or background subtraction. The processed video streams are then fed into AI components to identify excellent or lousy lane markers or to classify, recognise cars.

Due to the bulky nature of video data, it is best to leverage fog/edge computing infrastructure to have data processing tasks closer to the edge network where video streams are generated. We assume that the city has some of these computing resources available at the edge network, such as in road-side units, in traffic control elements or some cell towers. From the software development perspective, there are several software components involved
Figure 2.4: Our class of fog applications.

Due to geographic constraints, some locations do not have a traffic component or have overlapping communication coverage. Thus, the context information of the participating computing resources plays an essential role in the application architecture. At the same time, where there is no infrastructure available, a particular car might be selected to host the data processing component if it can fulfil the application’s requirements. This scenario illustrates the dynamic nature of the system and the challenge in coordinating communication among devices in such scenarios.

2.3 Application Characteristics

In this section, we conceptualise and discuss the characteristics of the described application scenario. We also identify the requirements from the developers’ perspective in building
such applications.

We identify this class of fog applications as large scale and dynamic fog applications. The reason is, it takes into account the dynamic nature of edge devices (e.g. mobile phones, smart cars) and involves a large number (e.g. hundreds or thousands) of devices that are distributed over a large geographic area. Due to these characteristics, our class of fog applications has a close bonding with the physical context of the computing infrastructure. That is, unlike traditional distributed applications running on the cloud with minimal knowledge of the underlying infrastructure, our fog applications are intimately connected to the device on which they are deployed. Figure 2.4 illustrates the positioning of our application class concerning three attributes: the deployment scale of the system, the level of bonding with physical context and the dynamic nature of the system.

### 2.3.1 System Scale

The deployment scale of the system can range from smart home, spanning smart offices, factories to smart cities. Each deployment scale has its own particular challenges that require its unique solutions. For example, at small scale such as smart home, device naming and identification become the topics of interest to the application developers. However, when it comes to the larger deployment scale, such as smart cities, these topics are irrelevant to fog computing developers.

At such scale, fog computing developers often work with groups of entities (e.g. software components, devices) rather than with any single one. Thus, classification and grouping methods become an important topic. It could be based on the class of the software components, such as all the background subtraction instances in the network, or on the context of the devices, such as all the devices in one area.

When the developers work with such distinctive groups of entities, communication among those entity groups becomes another topic of interest. That is, how do we express and enforce the communication semantic between a group of dashcam components with another
group of background subtraction components? Should the data streams be sent freely among them, or should there be any other communication semantic?

In this thesis, we focus on applications that are deployed at the city scale whose computing elements are geographically distributed. As a consequence of a geo-distributed computing infrastructure, we specifically consider the influence of physical context to the application design and development process. Cohesively, the next distinctive attribute of our target fog applications is the clear awareness of physical context of the underlying computing devices.

### 2.3.2 Awareness of Physical Context

We define three levels of awareness of physical context (of the computing elements) in applications, namely, not being aware of, hardware capability-aware, and physical context-aware.

Most applications we see everyday, such as common desktop applications, cloud-based applications are not aware of the physical context of any computing elements that are involved in such applications. Many mobile applications are aware of the hardware capabilities of the involved computing elements, such as the need to have a certain sensor onboard, or to have Bluetooth connectivity. Lastly, some mobile applications depend on the physical context of the underlying computing elements; for example, navigation applications depend on the location of the computing devices to operate.

Since we are focusing on the city scale characteristic for our fog applications, inherently, they also need to be aware of the physical context of the underlying computing elements where they run.

Again, different level of physical context awareness has its own unique requirements and specific solutions.

For example, without the need to be aware of the hardware context, the focus of the applications is on the computation activities. That is, which algorithms to use, what are the right design pattern, etc. are the common topics of interest to the developers.
When the applications have to be aware of the capability of the underlying computing hardware they run on, the developers are then interested in how to access such capabilities and make use of them. This is not to say that algorithms or programming primitives become irrelevant, however, it is safe to assume that these issues are not in the central part of the decision making process in such class of applications.

Similar phenomenon is observed when we consider applications that are aware of the physical context of the computing devices. In our example application scenario, the dash-cam component deployed in cars can be enabled or disabled based on the location of the car. With such application logic, the developers are required to tap into the context of the underlying computing elements, either to acquire the context data, or to monitor them. While the context acquisition process can be partly learnt from the previous class of applications (acquiring context data versus acquiring hardware capability of the underlying devices), there is a need to express and maintain the context-dependent requirements across devices. For example, how do we program and enforce the application requirements such that, the dashcam components should only communicate with the nearby background subtraction components?

Cohesively, the awareness of physical context in our fog applications is also aligned with the high level of dynamic nature of fog computing elements. That is, since physical context of devices change frequently, our fog applications are also said to be highly dynamic.

2.3.3 Dynamic Nature

We define four levels of dynamic nature of any system with regard to the changing in the physical context of the computing elements, namely, static, join/leave, predictable, and fully dynamic.

Many applications do not depend on the dynamic of the computing devices. As described in the previous section, applications that are not aware of the hardware context fall into this category.

Some applications depend on the availability of the participants. For example, many
"smart space" applications (e.g. smart homes, building automation) depend on a service discovery process to recognise which devices coming in or leaving the network. The dynamic nature of such systems is said to be at join/leave level following our definition. Under this category, developers often benefit from event-driven application models that react to the changes in the system’s dynamics. Thus, coordinating the streams of events (e.g. utilising Complex Event Processing engines), or applying service discovery mechanisms become example topics of interest in this category.

At the predictable dynamic level, the applications exhibit a certain level of dynamic nature beyond the join/leave level, however, such dynamic nature can be predicted or modeled so that the applications can depend on a fixed schedule or heuristic for their execution. For example, some "smart space" applications can support this level of dynamic by observing and studying the dynamic of the environment to carry out predicted execution in the future (e.g. predicting when occupancy is the highest to control the HVAC systems). Thus, developers in this category are more likely interested in how to apply a prediction or modeling method in their applications.

Lastly, for fully dynamic systems - our target class of fog applications, the applications have to periodically monitor the system status to make correct execution. Large scale systems such as smart city or fog computing are usually fully dynamic. This is because it is very difficult to predict or model such a complex system that involves a wide range of participants (e.g. citizens, vehicles). In this category, the developers are required to keep their applications up to date with the changing in the context of the systems. Therefore, accessing and make use of context data are the important requirements.

Again, this characteristic is also highly relevant to our previous focuses on the city scale and context-dependent characteristics. That is, applications at the city scale are more dependent on the physical context of the computing elements; they are also more dynamic.
2.4 Research Challenges

The application scenarios described in the previous sections show that, while fog computing enables a more efficient and time-sensitive model for many data-intensive and delay sensitive applications, it comes with its challenges in application development. Particularly, these challenges stem from those described intrinsic characteristics of fog computing systems, notably the geo-distribution of computing resources and the dynamic nature of participating devices [96] [88] [82] [66].

Our topic of interest in this section is to look at these inherent challenges of fog computing systems from the application development perspective. Because the deployment scale is large, the participating devices are very heterogeneous and are contributed by multiple entities (e.g. users, companies, municipalities). In this perspective, one of the most important requirements is the ability to leverage off-the-shelf, independently developed software components in a flexible and reusable way so that they can be easily programmed, deployed and run across such a fog computing environment. As Nath et al. pointed out in their survey, a generic, high-level application model for fog computing systems that supports its large scale and dynamic nature is still an open research question [69].

2.4.1 Large Scale, Geo-Distribution of Devices

Since the computing resources in fog systems are geographically distributed, it requires a different application model and software engineering process as compared to traditional distributed applications. That is, although these applications also consist of distributed components, the distribution of these components on the computing infrastructure is transparent to the developer. The reason is, in a centralised data centre setting, developers do not worry about which machine hosts which computation. In contrast, since computing and communication infrastructures in fog computing are not centralised, or homogeneous, developers need to specifically cater to these complexities.
Most importantly, the application model should be able to leverage off-the-shelf software components and coordinate them together to fulfil the application logic. That is, since these software components are written in different languages and by different people, there has to be a mechanism to put them together and coordinate them when they are deployed in such a fog computing infrastructure.

Furthermore, we also need to provide necessary APIs or primitives that allow the developers to interact with the underlying fog devices due to the close bonding between the application and the hardware context. That is, we expect that the application model would need to take into consideration the underlying computing infrastructure and its associated contextual information as an integral part of the design and development process. This requirement leads to the question of how to expose the context information to programming primitives or constructs that developers can use to build the application.

Finally, due to the heterogeneous context of fog devices (e.g. same software component is deployed in devices running at different locations), the deployment of software components results in a large number of component instances that are distributed in a large area. Thus, we particularly need support for the coordination of such component instances to dictate how they should interact with one another under which context.

2.4.2 Dynamic Nature of the fog computing infrastructure

Due to the close bonding with the physical world, fog computing systems exhibit a highly dynamic nature, in terms of both load fluctuation and context changes (e.g. location changes when fog nodes are mobile). While load balancing and dynamic scaling are commonly used in cloud computing to cope with the fluctuation in application load, it is more challenging to do the same in the fog computing environment. This is partly because fog computing resources are not as centralised and readily available as cloud computing resources so that it is harder to scale horizontally, and partly because the heterogeneous networking environment makes it difficult to locate resources for the load balancing task. In addition to this, changes
in physical context also play an essential role in fog applications, which requires a certain level of context monitoring and situational re-evaluation.

Therefore, we argue that there needs to be a dynamic coordination mechanism in fog computing that supports its dynamic nature. Since we are focusing on a class of fog application that exhibits a high level of dynamic nature, a periodic coordination model is necessary. This requirement leads to the question of how to effectively coordinate the whole infrastructure at large without incurring expensive coordination overhead.

2.5 Literature Overview

While a comprehensive literature survey is available in Chapter 6, this section briefly looks at the research communities surrounding the problem of developing fog applications.

Since Bonomi et al. coined the term in 2012 [17], research communities have been embracing the idea of fog computing for the broad class of data-intensive and delay sensitive applications. Generally, there are domain-specific and domain-agnostic applications that benefit from the massively distributed computing infrastructure [66]. In this work, we aim at a domain-agnostic, general application development approach where programming primitives and system supports are designed to take advantage of fog computing resources. Within this branch, there were a few programming models that are specifically tailored to the fog computing system, notably Mobile-Fog by Hong et al. [39], Crystal by Jeong et al. [42], Yangui et al. [95], Olena et al. [86] and FogFlow by Cheng et al. [22].

The main limitations of these research bodies are that they often see fog computing infrastructure as a static, hierarchical computing system that is geographically distributed. The provided programming models support the distribution of application components across the distributed computing resources statically rather than coping with their highly dynamic nature. Thus, their dynamic nature becomes a second class characteristic that pertains to the underlying infrastructure and not related to the high-level application itself.
We argue that this view is a simple extension of cloud computing toward distributed, hierarchical fog computing infrastructure. Therefore, a minimum effort has been spent to distinguish the characteristics of a fog computing system. Here our primary focus is to support a highly dynamic computing system across a large area of resource distribution. Hence, the underlying physical context surrounding each computing elements and their large quantity become top priorities for our work.

Some works do cater to the dynamic nature of the systems, such as in [39], however, their programming models follow the traditional approach of mixing computation and communication activities. As we shall see in this thesis, an important design choice we make is the adoption of exogenous coordination, which explicitly separates these concerns. We shall see why exogenous coordination is suitable for developing large scale and complex fog applications.

On the other hand, it is reasonable to say that fog computing is yet another distributed system architecture where there is already a wealth of research effort in supporting the development of distributed applications. As we shall see in later chapters in this thesis, although we can learn extensively from experience in distributed systems, the main difficulty arises from the coordination of software components on top of distributed devices under different physical context.

Our observation is that although we have such a broad knowledge on distributed systems in general, many of its applications rely on confined distributed computing infrastructure, such as data centres, where the context of the computing elements is not crucial to the application layer.

Over the last decade, with the research into Context-aware applications, Pervasive applications, the WSN and the IOT, we now start to care about the context of the computing elements. Thus, a newer class of distributed systems, often called Mobile Ad-hoc Network was drawing attention from the research community. While we also learn from experience
building context-aware applications, they are found to surround the presence of a user. Thus, the process of building these apps surrounds the requirements of the end users instead of a larger entity, such as a smart city. This phenomenon has an important implication as we only care about the context of the user rather than that of the computing elements. Obviously, smart city applications do exist, but traditionally, they rely on cloud computing infrastructure, which is again a confined distributed computing environment.

In summary, our proposition here is that we focus on the development of large scale fog applications that involve computing resources that are large in quantity and dynamic in context. Our application logic, therefore, is also dependent on such a dynamic context of the computing infrastructure on which it runs.

2.6 Thesis Objectives

From these backgrounds, we found that there are three overall objectives in supporting the development of our fog applications. First, we need to concretely understand how fog applications differ from general distributed applications. Second, from understanding such unique features of our fog applications, we are able to design an appropriate application model to represent and to help developing them. Lastly, we need to look at the deployment of our fog applications in a large scale and highly dynamic computing infrastructure and identify the unique system supports needed.

2.6.1 An Analysis of Various Application Models

To fulfil the first objective (i.e. answering RQ1 - *How do fog applications differ from general distributed applications?*), it is necessary to conduct an analysis over various application models found in literature, particularly the ones that are designed for fog applications. From here, a taxonomy of application models helps to distinguish our fog applications from others.

As a result of the analysis, we found that our fog applications distinguish themselves from the others by having the close bonding with the physical context of the underlying
computing devices. Our second finding is that "decompositionality" becomes an essential feature of the application model to support the complexity of the large scale, geo-distributed fog systems. It is worth recalling that by “decompositionality”, we mean the ability to incorporate, in a flexible and reusable way, off-the-shelf software components that are independently developed.

These observations lead us to the second objective of the thesis, which is an exploration of coordination-oriented approach for our fog applications, especially exogenous coordination where there is a clear separation of concerns between computation and coordination activities. The reason is, such a clear separation of concerns naturally breakdowns the complex applications into smaller components, eventually it becomes the ultimate choice for solving the “decompositionality” of fog applications.

2.6.2 Context-dependent Exogenous Coordination in Building Fog Applications

There are two types of coordination models, endogenous and exogenous [5]. Primarily, this classification is based on how the coordination activities are carried out, whether it within each participant or from an external entity that oversees the whole system. In endogenous coordination, each participating software component by itself coordinates their participant into the common application scenario. Thus the coordination activities take place within each participant. Tuplespace [33] is a typical endogenous coordination model where each software component proactively decides with which tuple it should interact. On the other hand, exogenous coordination differentiates itself by having an external entity that oversees the whole system and coordinates all the participating components. Thus, a dedicated coordinator is an essential part of an exogenous coordination platform.

In large scale cases such as in fog applications, it is difficult for each participating entity to coordinate their participation to the common application scenario solely by itself. Therefore, we argue that exogenous coordination is a more appropriate approach to develop our
fog applications. Furthermore, since coordination primitives are pushed out of the computation activities, the participating software components do not have to make any assumptions about the environment they are running on. This characteristic makes it easier for application developers to incorporate off-the-shelf components in a flexible and reusable way.

While exogenous coordination itself advocates for the clear separation between computation and communication, an important characteristic toward solving the complexity of fog computing environment, it lacks basic primitives that allow the developer to describe a broad range of large scale fog applications as we focus on this thesis (the reason can be found later in Chapter 6). In essence, we need to build upon exogenous coordination and extend it with context-dependent primitives so that it can be leveraged for our application development process. This is to answer RQ2 *(How can fog applications be developed with minimum requirements on the developers to cope with the complexity of fog computing infrastructure (large scale and dynamic)?)*.  

### 2.6.3 System Supports for Fog Applications

In addition to extending exogenous coordination with necessary primitives, a proper coordination platform is required. Such a coordination platform serves as the external coordinator to the participating entities. The main requirement for such coordination platform is to cope with the dynamic nature of the fog computing systems.

As discussed previously, fog applications exhibit a high level of dynamic nature by having their computation activities to be deployed in volatile computing infrastructure, such as in moving vehicles. A natural consequence is that the physical context of the computing elements changes frequently. Since fog applications have a close bonding with the physical context, their logic tends to depend on the physical context of the underlying computing elements.

While many exogenous coordination models and languages have been designed for different requirements, most of the work only focus on the software component level, leaving
Therefore, our third objective is to study the problem of coordinating large scale and dynamic fog applications. This is to answer RQ3 (*How an application platform for fog can keep up with its large scale and dynamic nature in an efficient way while still maintaining the correctness of the application logic?*).

### 2.6.4 Thesis Contributions

Based on the analysed requirements and challenges, this thesis focuses on the study of applying exogenous coordination approach in supporting the development of our class of fog applications. We claim the following research contributions:

- **Through an analysis of various application models, we show that:**
  - *Fog applications distinguish themselves from the others by exhibiting a close coupling with the physical context of the computing elements, and having a crucial requirement on decompositional property.*
  - *Exogenous coordination is a suitable approach in developing an application model for fog applications.*

- **By developing a context-dependent exogenous coordination platform (Distributed Node-RED) and field trial real-world applications, we showed that:**
  - *Separation of concerns between application data and contextual data (in addition to communication and computation) is an important concept in designing an exogenous coordination model for fog applications.*
– Context-dependent replications of fog application components need special coordination primitives to be developed (i.e. context-dependent constraints and bounded communication cardinality).

• By developing a large scale simulation of fog computing environment, we showed that:

  – The coordination of large scale, dynamic fog applications benefits from an incremental coordination approach that remembers the past coordination execution to reduce the coordination overhead.

  – System dynamics do not affect the coordination activities with regard to the application performance (i.e. orthogonality between computation and coordination activities).

Distributed Node-RED (DNR), the coordination platform for fog applications developed in this thesis is fully open-sourced. It has been widely evaluated in both academic and industry environment, thus further proves the necessity of such an application model and the feasibility of the approach.

2.6.5 Case Study-driven Research Methodology

The importance of case study-driven research has been identified by many researchers, particularly in the social sciences field of study [31] [92]. An important observation is that when there are "concerns about limitations of quantitative methods in providing holistic and in-depth explanations of the social and behavioural problems in question" the researchers are able to "go beyond the quantitative statistical results and understand the behavioural conditions through the actor’s perspective".

In the research into fog computing and many new computing paradigms where concrete architectures and systems are not yet available, a holistic view of the ecosystem becomes a suitable approach.
For example, one of the objectives in this thesis is to find and evaluate the appropriateness of an application model in fog computing. Ultimately, an evaluation of such research question is to work directly with fog application developers to identify the need and possible solutions/supports. Since we do not have yet such a fog computing environment, the definition of fog computing developers is still unknown. Thus, we choose to apply a holistic view of the fog computing paradigm and depend on case study analysis to answer our research questions.

To embark on such a case study-driven research, an important question is that, is the chosen case study representative enough to generalise an answer? [84]. As the core of our research, the specific application scenario (a case study) that we developed here is well recognised in the "fog" community. That is, due to the bulky nature of video data, many researchers have used a similar use case to show the necessity and to generalise a range of fog computing systems [21] [39] [70]. Thus, we believe that the chosen case study can be seen as a representative one to study many problems in fog computing area.

2.7 Summary

This chapter introduced the background and motivation behind this thesis. Through the developed application scenario, we show that fog computing is a new computing model that promises to complement existing cloud computing in supporting a large class of data-intensive applications. Developing applications in fog computing, however, is not an easy task due to the large scale and dynamic nature of the fog computing infrastructure.

We tackle these challenges in developing fog applications by answering the three research questions. Our first objective is to conduct an analysis of various application models to identify the unique features of fog applications. Due to the complexity of large scale and dynamic fog applications, we propose that exogenous coordination is a suitable approach. This leads us to our second objective, which is to build upon exogenous coordination and
extend it to support context-dependent fog application logic. Our final objective is to develop system supports that are unique to realising our fog applications in large scale, dynamic infrastructure.

The thesis follows a research methodology based on case study, where we based on our developed application scenario to generalise the requirements and solutions in addressing the development of fog applications. Additional application scenarios will be discussed throughout the thesis to weight the applicability of the proposed solutions and to generalise them.
Chapter 3

An analysis of Various Application Models: Toward Developing Large Scale, Dynamic Fog Applications with Exogenous Coordination

With the advent of more advanced computing models beyond personal computing, such as cloud computing, jungle computing or fog computing [37], it is essential that we understand the application development process of each model. A better understanding of how applications are built on different computing models and in different settings allow us to design better development tooling and system supports.

In this thesis, we use the term application model deliberately instead of programming model when addressing the development of fog computing application. Our focus is, indeed, to design an application model for fog computing. From our survey, we found that while there were many programming models designed for different types of systems ranging from Personal Computers to WSNs and the IoT, the notion of application model has not yet been
clearly defined. We believe the construction of complex applications does not solely depend on the language aspect (i.e. programming model), but also other criteria, such as application structure, abstraction levels or communication models. Certainly, some of these criteria were already discussed and surveyed among many existing systems. We found that it is valuable to put together a set of aspects that collectively define the notion of "application model", which we can use as a guideline for constructing different types of applications in different system settings. In addition, we present different views for certain aspects that have not been recognised before, mainly due to the presence of large scale and complex fog computing systems that span over large physical areas.

Thus, before going into solving our challenges in developing fog applications, we take a step back and adequately examine the notion of "application model". Such a definition serves as our tool to analyse existing works, judging their applicability and pushing forward to develop an appropriate application model for fog applications.

From surveying the literature, we consider five aspects that we found are important for developing applications, namely, scope, abstraction level, application structure, programming model and communication model. An overview of these aspects is shown in Figure 3.1.

### 3.1 Scope

The scope of an application defines the boundary of the application and its external ecosystem. For example, a typical distributed application is that of a traditional Internet application, which consists of a client terminal and a server component that communicate with one another. At the broader scope, we can perceive this scenario to be one application that has two interconnecting components, a client and a server. If we zoom in each of these components, either the client or the server is seen as another utterly independent application. These smaller applications can also be further decomposed. The server component, for instance, could include a web server and a database; the client terminal could include a user interface
Our observation is that different levels of application scope come with different concerns from the developer; consequently, they need different toolings and supports. For example, at the atomic component level (we use the term atomic here to denote that such component "should not" be decomposed further), concerns are how to derive efficient computing algorithms and how to choose the suitable programming primitives and constructs. At a higher level of scope, we start to employ different design pattern to facilitate the inter-component interaction. One of the primary considerations for such level is whether to use object-oriented or functional programming. Moving higher up the scope levels, we start to think about the communication patterns among components that are distributed across machines.

We define four levels of application scope, namely component scope, inter-component scope, inter-device scope and inter-context scope. This is illustrated in Figure 3.2.

**Figure 3.1:** An Analysis of Various Application Models
3.1.1 Component scope

Component scope is the most basic application scope where the developers focus on the implementation of algorithms and basic programming concerns such as the languages, or the programming models (e.g. imperative, declarative or hybrid) to solve the functionality problem of one component.

With this application scope, the main task the developers are doing is purely implementing computation activities.

Examples of component scope applications are sorting, mapping or reading sensor data.

3.1.2 Inter-Component scope

In this application scope, the components themselves are well encapsulated and have well-defined interfaces. Often, the interaction among those is object method calls or function invocations. The task of the developers is to combine the components to fulfil the applications’ requirements.

Primary concerns in this application scope are choosing the right design patterns to max-
imise the efficiency of inter-component interactions, choosing the right component organisation (e.g. flat versus hierarchical, see later sections), or to learn the components’ interfaces.

Examples of applications in this scope are general computer applications, such as desktop apps, mobile apps. That is, these applications are developed from many subcomponents, libraries.

### 3.1.3 Inter-Device scope

In this application scope, the components themselves can be distributed on top of several physical devices. Thus we have applications that run across different devices. Many applications within this scope can also be deployed in a single machine; however, desired features such as load balanced ability, scalability, or availability motivate the distributed deployment of the application’s components.

The main concern in this application scope is then to coordinate the interaction among the components across devices, for example, to ensure the communication follows some strict orders or rules (e.g. message delivery semantic that is exactly one or at most one).

Examples of inter-device application scope include a traditional client-server application, a load-balanced web server, or a high availability file storage system.

### 3.1.4 Inter-Context scope

In this application scope, not only the components are deployed in a distributed fashion on top of many devices, the context of each device is different from one another and can interfere with the application’s logic. The running devices in this scope are heterogeneous rather than homogeneous as in inter-device scope applications. They come from all shapes and sizes and operate in different conditions.

The main concerns of the developers when developing applications at this scope include to coordinate the inter-component interactions and cater specifically to the physical context of the underlying devices.
Examples of applications in this scope are WSN, IoT, or Smart City applications, where the location of a computing device also participates to the application’s logic.

Note that our definition of application scope is also related to the *programming-in-the-large vs programming-in-the-small* phenomenon in the construction of large software [28] or recently the *macro-programming vs node-level programming* differentiation in Wireless Sensor Networks [65]. Here we aim at a more general notion of application scope that involves different levels of application decomposition and deployment. We believe this definition allows developers to position themselves better and choose the appropriate tool for their needs.

### 3.2 Abstraction Level

We define the abstraction aspect of an application model as the level of flexibility in expressing the application’s logic. This definition is not to be confused with the abstraction level of programming models, which is used to denote the flexibility or transparency when the developers want to tap into low-level system concerns (e.g. working with memory, controlling networking stack). For example, as we shall see in [Chapter 6](#), TinyDB, a query processor for WSN, provides a very high level of abstraction in terms of programming model [59] (it abstracts away the technical details of a WSN). However, with regard to our definition of application’s abstraction level, it belongs to the data querying applications group, which has the lowest level of abstraction (it provides the most flexible way to express an application logic, but the users are also required to master its query language).

In our definition, applications with a higher level of abstraction are easier to operate, but functionality is limited. Meanwhile, the ones with a lower level of abstraction are more flexible (i.e. they allow users to do many different tasks, even ones that are not foreseen by the developers) but are more difficult to operate.

To better understand this concept, it is worthwhile to take a look back at the relative
difference between an application developer and its end user since this is a highly related concept.

Traditionally, there was a clear difference between these two classes of entities. Application developers build the applications so that to make it very easy for the end users without technical knowledge to fulfil their requirements. On the other hand, what the end user can do with the application depends on what the developers dictate in their code. In this scenario, we say that the abstraction level of the application is very high as all the detail functionality is well abstracted by the developers.

One example in this scenario is a temperature control application. All the user needs to make the application to operate is to specify the desired temperature, and the application fulfils this requirement. In this case, the distinction between the end users and the developers is undoubtedly sharp. However, while being very easy to use without any technical knowledge, the users cannot achieve certain requirements such as when it is at night, drop the temperature for 2 degrees C and raise it again during the day. Thus we say that applications with higher levels of abstraction are more comfortable to use but less flexible in terms of which requirements it can achieve.

Moving down the abstraction level scale, when the application is more flexible, it requires the users to do specific tasks rather than pushing a button. For example, with the above application scenario, the users should know how to express a simple rule such as if the time is night time, decrease the temperature and vice versa during the day. With this requirement, the developers have to expose the application internals to the users slightly. This essentially means the developers are sharing a portion of their responsibility with the users, consequently making the line between end users and developers slightly less clear as before.

In its lowest level of abstraction, the application’s users tend to blend with the developers as both of them should know the ins and outs of the application themselves. Thus, sometimes it is unclear whether an individual user is an end user or a developer.
Figure 3.3: Abstraction Levels of Common Applications

Data query languages such as SQL are typical examples of this level of abstraction. To take advantage of a query engine, the users have to master its query language. In turn, the query engine allows the users to work directly with the application’s data, either accessing or manipulating them. Thus, query engines are the examples of applications that provide the most flexibility to their users. On the other hand, they also demand the highest level of competency from their users. Looking at query engines from this perspective, it is unclear whether or not a user is a developer or an end user.

Understanding this relative relationship between the developers and end users allow us to classify the abstraction levels of applications.

We define five abstraction levels, from the highest level of abstraction to the lowest as follows: Goal-based, Rule-based, Component Cooperation, Data Processing and Data Querying applications. We illustrate these categories in Figure 3.3 and describe them as follows.

3.2.1 Goal-based

Goal-based applications offer the easiest way to interact with the application state and its data, normally through just a push of a button. Applications we see today, such as mobile or desktop apps belong to this level of abstraction. To carry out business logic, the users only
need to navigate through a range of predefined inputs elements such as buttons or text fields. Thus, zero programming skill or competency level is required by the users. In contrast, the developers have to put in their best effort to bring a seamless an easiest user experience to the apps.

Another example application is the thermostat, which allows the user to key in their desired room temperature. The application itself coordinates its components (e.g. by turning on and off ventilation fans) so that the desired temperature is met. A complete solution to this example is presented in [63] where the application depends on a semantic reasoning process to coordinate the components to achieve the users’ predefined goal. Since the application is very easy to use, its functionality is limited to the features that are hard coded by the developers (e.g. we cannot express the requirement such that when the temperature drops below 20 degree Celsius for 10 minutes, start the ventilation fan).

3.2.2 Rule-based

Applications at this level of abstraction are slightly more flexible than the previous one and require a certain level of involvement from the users’ perspective.

In this level of abstraction, the users are allowed to specify the requirements through a set of rules rather than a specific goal. A set of logic skills is required to make use of the applications. *If This Then That (IFTTT)*[^1] is a typical example of this level of abstraction. When the user installs the IFTTT application, they can express their application requirements through a set of simple rules in the form of *if this then that*. For example, if the time is night time, dim the light and reduce the temperature.

Since the users can specify the rules directly, the developers’ involvement lessens. We say that as the abstraction level goes down, the developers’ tasks lessen and the users’ tasks increase.

[^1]: http://ifttt.com
3.2.3 Component Cooperation

At this level of abstraction, the components of an application are exposed to the users so that the users can control their cooperation within the application. The applications at this level are much more flexible; consequently, the users can leverage them to achieve many different tasks, sometimes out of the imagination of the developers.

This level of abstraction strikes a balance between the users and the developers’ involvement. The developers do have a set of criteria to define the usage of their applications; however, they also give the users the power to change the applications’ behaviours the way they want.

Examples of applications in this case are visual programming platforms, such as Microsoft Flow\(^2\), Max\(^3\) or Node-RED\(^4\).

Since the purpose of these programs are for expert users to construct their applications (applications that make applications), the notion of application developers and users, in this case, becomes system developers and domain experts. That is, as the abstraction level of applications goes further down, we start to see less the role of an end user. Instead, we are seeing domain experts, more of application developers and eventually, the notion of application users vanishes.

3.2.4 Data Processing

Data processing is the next level down the application abstraction scale, where the main purpose of an application is to modify or transform its data.

At this level, the users are given with some data processing primitives (e.g. min/max, stdev) and should know how to use these primitives for their needs. The detail implementation of these primitives is the responsibility of the developers.

\(^2\)https://flow.microsoft.com/
\(^3\)https://cycling74.com/products/max/
\(^4\)https://nodered.org
Some examples in this scenario are the spreadsheet application or the awk command in Linux operating system. The spreadsheet application allows users to do many processing tasks on their data. It naturally requires the users to be proficient at the tasks they are completing (e.g. using the max function to find a maximum). Awk command in most Linux distributions is a powerful tool for processing text files. It is one example of an end user applications that require specific skills to master the operation of such applications to carry out the user’s needs.

With this level of abstraction, we say that the role of application users becomes more dominant than application developers. Thus, *domain experts* becomes a more appropriate term.

### 3.2.5 Data Querying

Lastly, data query-based applications provide the most flexibility to address the application’s requirements. With this level of abstraction, the users have full control over what they want with the application by issuing queries to access and manipulate the application data and state. Examples of this are traditional database systems and their query languages, such as SQL. To illustrate the flexibility, web-based or enterprise applications can be constructed from a single database layer where users can issue queries and manipulate the business data directly from the database. Another example is TinyDB [60], a query processing system for Wireless Sensor Networks that allows access to sensing data and a range of data aggregation operations. TinyDB itself could be seen as an application for WSNs, which allows users to interact with the networks via issuing queries.

Data query-based applications provide the most flexibility for users to express their requirements. However, the users also become the developers in this scenario as they have to master the query language for their needs.
3.3 Application Structure

Unlike the scope, or abstraction level of an application model, the application structure aspect taps into more details about how the developers designed an application. Generally, subcomponents of an application can be organised into either a flat or hierarchical structure. While a hybrid one does exist, as we shall explain, it has inherent drawbacks.

Note that, this aspect is different from the distribution of components concerning the hosting hardware. Generally, the subcomponents of an application can be either distributed or non-distributed concerning their hosting hardware. Most of the applications we see every day, such as desktop apps, mobile apps, are non-distributed as they operate in a single machine. Some of these applications follow client-server or peer-to-peer architecture; they are distributed applications. Regardless of this physical distribution, the structure of components within one application can still be either flat or hierarchical.

The application structure aspect is summarised in Figure 3.4.
3.3.1 Flat Structure

Flat structures usually need a dedicated coordination mechanism to facilitate the interaction among subcomponents, such as a service registry that holds information about the components, unless mass broadcast is used. Flat structures can be further categorised into peer-to-peer and role-based structure. In peer-to-peer structure, application’s components are generally homogeneous, the only difference among them is the data they hold. On the other hand, applications with the role-based structure are more heterogeneous in terms of their sub-component composition. Accordingly, these subcomponents are different from one another and are bound together in one program by dedicated coordination mechanism.

Examples of a flat structure include peer-to-peer applications, such as torrent file sharing, or blockchain-based smart contracts. In these types of applications, special component discovery services are required for inter-component interaction. That is, in torrent file sharing applications, it is the torrent trackers that facilitate this interaction, in blockchain-based applications, it is the DNS Seed nodes.

Example applications with role-based structure include micro-service (components play the role of different services), client-server (client components or server components), or map-reduce (mappers and reducers) applications.

3.3.2 Hierarchical Structure

Hierarchical structures can be self-coordinated, meaning the components themselves can discover and interact with one another without a dedicated service.

Examples of hierarchical structures are domain name systems, applications that rely on MQTT message broker topic naming scheme.

These component structures have their pros and cons and are chosen based on the application’s characteristics. An essential aspect of evaluating these two organisational models is communication among components. In flat architecture, the components communicate
with one another either directly or via a communication broker. In hierarchical architecture, inter-component communication is narrowed to parent-child communication. Thus, two components A and B can only communicate if they share a common ancestor component. However, there is no need for a dedicated coordination entity in the hierarchical structure as the components can interact via their direct parent or children.

Hybrid structures are possible and do exist \[^{39}\] to take advantages of both worlds. However, when the components in a hierarchical structure communicate directly with one another, their parents might have difficulty keeping track of the state of their children.

Applications based on Akka actor system \[^{5}\] or recently React programming model \[^{6}\] are examples of the hybrid structure.

### 3.4 Communication Model

Two popular communication models for applications are message passing and shared memory models \[^{4}\]. While these models are from communication patterns in distributed systems, it also applies to non-distributed applications. For example, method calls among components within a non-distributed application represent the message passing communication model. Meanwhile, global-variable scope represents the shared memory model. These communication models are summarised in Figure 3.5.

#### 3.4.1 Message Passing Communication

While message passing is a well-known communication model in many distributed applications, large scale applications at inter-context scope exhibit a different communication cardinality that we do not see before. For example, we have point-to-point (1-1) or point-to-multipoint (1-*, *-1, as in broadcast or aggregate) communication cardinality among the components. Due to the context scope, sometimes it is irrelevant for a component to broad-

[^5]: https://akka.io
[^6]: https://reactjs.org
cast itself to the whole ecosystem. In such cases, bounded group communication among a small set of components that share the same context might be necessary. Thus, we have another category of communication cardinality beyond the unbounded one, which is bounded group communication.

### 3.4.2 Shared Medium Communication

In shared medium communication models, application’s subcomponents communicate via a shared medium where one component writes data to the medium, and the others read from the same location. This is sometimes referred to as implicit rather than explicit communication as communication is a side effect of the actual data sharing process [4]. Shared medium communication is also regarded as the control-flow hidden interaction model [8]. This is because the control flow is not visible, nor easily to be grasped as the components do not interact directly with one another.

Examples of shared medium communication models include applications that are based on tuple space or publish/subscribe model.

It is worth noting that the shared medium communication models do not require a ded-
icated coordination entity to facilitate the interaction among components. This is because the shared medium itself becomes the coordination entity that glue together the applications’ subcomponents. In contrast, message passing communication models do require a dedicated coordination entity to dictate to where each component should direct the message.

### 3.5 Programming Model

The last aspect of constructing an application model according to our definition, is its programming model. This is the most involved aspect of the application development process. Generally, researchers categorised programming models into declarative or imperative languages.

Sometimes, a hybrid category is introduced \[65\] in which, the language to develop application components is imperative while the language to develop the interaction among components is declarative. Surprisingly, if we apply the definition of application scope in this thesis, this view is no longer suitable. That is if we see the scope of the application to be inter-component scope, the components’ internal implementations become irrelevant, the application is, therefore, developed with declarative languages only. A similar argument could be applied to the component scope, where the declarative interaction among components is
irrelevant.

Another related concept is the notion of node-level and system-level programming, which is borrowed from the WSN research. In programming WSN, node-level programming refers to the practice of developing applications for individual nodes while system-level programming refers to developing the collective sensing system as a whole. Imperative language tends to do well in node-level programming while declarative is more suitable for system-level programming [7].

It is perceived that imperative language is suitable for expressing the functional aspect of an application while declarative is used to express the non-functional logic [65] [5]. For example, to execute a sensor reading from one device, imperative language is used to call system procedures and deliver the data packages. However, to specify logic such as execute this sensor reading only from 9 AM to 5 PM every Sunday, it is more concise for the developer to specify this using an annotation approach than an imperative one.

Thus, we define another notion of hybrid programming model to denote a mix of annotations within the imperative or declarative code. Aspect Oriented Programming [30] is an example of hybrid, annotation-based programming model. A summary of programming models is shown in Figure 3.6.

3.6 Toward Building Large Scale Fog Application

This section analyses the defined aspects of application models and gives the recommended requirements for designing an application model for our fog applications. A summary of our recommendations is highlighted in Figure 3.7.

3.6.1 Application Scope

Due to the large-scale distribution of computing resources, their physical location, or more generally, their physical context, becomes an important factor in a fog computing application model. Thus, we say that our class of fog applications falls into the inter-context application
Figure 3.7: Analysed important design requirements for building our class of fog applications.

For example, in [47], the authors deployed a smart city application where cameras are mounted on city’s garbage trucks, and the captured video streams are used to identify the road markers that need to be repainted. In this application, the city might only want to do the road marker classification in specific regions within the city. Since the garbage trucks are mobile entities, the developers have to program the application so that the software components are enabled or disabled appropriately based on the cars’ current location. Furthermore, when leveraging video streams from vehicles’ onboard cameras to do the image classification, process those video streams is a computation and communication intensive task. It can require the application to exploit a wide range of "nearby" computing resources located at road-side units or mobile base stations.

In these scenarios, we see the critical role of the physical context of the computing elements to the correctness of the application logic. We also see how the application developers might use it in their applications.
Therefore, to develop an application in complex fog computing infrastructure, we argue that the application model should provide developers with the ability to express their application requirements based on the physical context of the underlying computing elements.

At the same time, we also found that there is another separation of concerns when building inter-context scope applications, which are not present when building typical applications. The first concern is how to implement the application logic (component-scope), and the second is how the system can be deployed across different physical contexts (inter-context scope).

The application does not only run on a single device or server but across cloud servers, gateways and edge devices under various contexts. This involves writing components that encapsulate functionality, can be easily distributed and can communicate with other required components in various ways. The second concern is the need for the inter-context application developer to specify how the system as a whole decides where groups of application components should be split, run, and how do they communicate with each other.

In essence, the developer needs to adopt an inter-context programming mindset, one that involves specifying context-dependent constraints such as location, computing and network environment, replication and cardinality requirements. The system can then use dynamic information from the physical environment such as the quality of network communications, current location, current power levels and other factors to decide what application components are deployed where. The combination of these two concerns, expressed in two levels of application scope that work together, is key to the development of our fog applications.

3.6.2 Abstraction Level

Due to the large scale of the system, our class of fog applications exhibits a substantial level of complexity. Thus, the application model should find a suitable balance between flexibility and expressiveness.

Clearly, we can design an application model at its highest level of abstraction - goal-
based - for a smart city. For example, define a goal (e.g. city air quality index to be lower than 25) and all the participating devices in the city cooperate to achieve such a goal. The scale of our applications sometimes makes this approach unfeasible.

On the other hand, if we choose to design an application model with the lowest level of abstraction (i.e. data-querying), the cost of implementing any smart city solution might be prohibitively high.

Meanwhile, the distributed nature of our fog application class requires an application model that promotes application decomposition capability. This makes some abstraction levels, such as rule-based or data-processing inadequate.

To this end, we found that component cooperation seems to be the appropriate abstraction level for our class of fog applications. As per our analysis, it provides a reasonable balance between flexibility and expressiveness in building large scale, complex fog applications. Naturally, it also fosters the development of large scale distributed applications.

On top of this, an essential requirement to develop an application model based on the component-cooperation level of abstraction is that the communication and computation aspects of one application should be well defined and separated [5].

That is, although fog applications, which are distributed system applications in general, can be programmed using many existing models and techniques in distributed systems such as Mobile Agents [85], TupleSpaces [34], RPC-based Middleware such as Jini, CORBA, or RESTful web services. It has been shown that these approaches, which rely on the extensions of the sequential programming paradigm, are ill-suited to meet the challenges of large scale and massively distributed computing systems [71]. The problem of these models and techniques when being applied into such complex applications is that the communication primitives which bind sub-systems together are mixed within the computation parts.

A typical scenario is when the developers work on the computation code, they have to decide how to carry out the communication at the same time. For example, in Mobile
Agent or Tuple Space-based implementations such as LIME [67], it is to decide the target agent or tuple space to send the data to after finishing a computation activity. In another implementation such as RESTful web service, it is to decide which API endpoints to invoke on the remote services.

This mixture of communication and computation aspects hinders large scale and complex distributed applications to be built due to the lack of modularity and reusability, as well as the need to cope with complex nature of communication (asynchronous, heterogeneous, volatile). In such a setting, the developers should pay attention to the high-level application logic, which is how software components are glued together to solve a problem, rather than to work on each one.

Therefore, to design for the component-cooperation level of application abstraction, communication and computation should be well defined and separated. As we shall see in the following sections, exogenous coordination is a closely related concept and an appropriate approach for the component cooperation level of abstraction.

### 3.6.3 Application Structure

While hierarchical structure remains the popular design choice for many fog applications [38], for applications that operate at large scale and more dynamic scenario, this structure poses some limitations in terms of scalability and flexibility.

This is because, at large scale scenarios, the components of one application are arbitrarily distributed, their operations are dynamic and more likely unpredictable. Thus, they need a more flexible communication pattern that can be quickly adapt to their dynamic nature. A hierarchical structure depends on the parent-child relationship among components. This means such relationships have to be established first before any interaction can be made. Furthermore, interaction among components always needs to go through their common ancestor, which could be very high up the hierarchy. At large scale deployment with thousands of devices and tens of hierarchical layers, this can quickly become unmanageable.
For such class of fog applications, we advocate for a flat structure where inter-component interaction is more flexible and fast to adapt to changes. As discussed in earlier sections, an important drawback of flat structures is that a dedicated coordination layer is required to facilitate this inter-component interaction.

This requirement has several distinguishing challenges. First, the replication of components in a fog computing environment is context-dependent. That is, each application component in fog computing can be deployed in many participating devices, yielding many instances or replications of such component. While these component instances are identical in terms of logic or even data, their physical context makes them distinct entities. Second, since each instance of an application component is a distinct entity, the component composition out of these instances is also non interchangeable.

To illustrate this, let us take an example from the popular class of Internet applications. In traditional Internet applications, while there are multiple instances of each component, they are load-balanced and seen as a single entity from other components’ point of view (e.g. a web client sees a cluster of load-balanced web servers as a single entry point; a web server sees a load-balanced database cluster as a single database entry point).

In our class of fog applications, due to the large scale deployment of the computing elements, there are many possible compositions of the software components’ instances. Each composition might serve at a particular location of the city, and each component instance sees one another as an independent peer. Therefore, these compositions are non interchangeable. In essence, they should be non-overlapped (no shared instances between combinations), and the supporting system has to derive as many of these non-overlapped compositions as possible to extend the coverage of the application.

Thus, even though fog applications use a flat structure, the actual application model has to take into account the context-dependent replication of application components and their compositions.
3.6.4 Communication Model

For large scale fog applications that span a wide area and consists of independent software components and various devices, shared medium communication model has its inherent limitations.

Firstly, each of the application’s components individually has to conform to the shared medium protocol and the shared medium location. This means the implementation of each component is influenced by its deployment environment, which reduces its reusability. Second, the shared medium quickly becomes a bottleneck for the inter-component interaction at large scale deployment. Third, the application’s control flow is not visible in shared medium model [8], at large scale deployment, this inherently affects the design capability and maintainability of the application model.

Thus, toward our class of fog application, message passing is recommended as the ultimate communication model for large scale applications [8].

Going further into the message passing communication model, we also found that, fog computing exhibit a different style of communication cardinality. In particular, due to the context-dependent replication of application components, communication cardinality in the fog tends to be bounded (i.e. 1-N, N-1, N-M) instead of unbound ones (i.e. 1-*, *-1). This is because of the large scale deployment of fog computing elements that makes the system-wide broadcast or aggregation communication irrelevant to many participating devices (e.g. components only care about data sent from other ones that are close to them).

This means there has to be support for application-level group communication. Coordinating these group communication is then the central requirement of a fog computing platform.
3.6.5 Programming Model

As we discussed earlier, imperative programming models are suitable for expressing the functional aspect of the application’s components, i.e., for developing component-scope applications. Meanwhile, declarative programming is suitable for expressing the overall components’ behaviour and their interaction within a larger application. This is due to the concise of declarative language that is suitable for expressing complex inter-component interactions, which otherwise would be difficult to implement and hard to understand using an imperative programming model.

Toward developing large scale fog applications, we found that the close bonding with the physical world of the computing devices makes their physical context a particular concern that affects the application’s logic. By referencing to the aspect-oriented programming model, we can say that in our class of fog applications, the physical context of the underlying devices acts as a cross-cutting concern with regard to the component’s functionality.

3.7 Exogenous Coordination for Building Fog Applications

Through our literature survey, we came across the concept of coordination models and languages that we found plays a critical role in building component-based applications.
In [35], the authors highlighted the significance of coordination languages and the need to have a dedicated body of coordination languages rather than seeing coordination aspect as just a task of the computation process. A survey by [71], which draws on significant research into coordination languages and models, identified two classes of coordination languages, data-driven and control-driven coordination. In data-driven coordination, the computations are coordinated based on the data involved in the coordination activities while in control-driven coordination, the coordination is based on the cooperation pattern that computation processes adhere to.

In [5], those two classes of coordination models were also called "endogenous" and "exogenous". The reason for this is the observation that data-driven coordination languages (e.g. TupleSpace) provide primitives that are mixed within the computation part, hence the term "endogenous"; meanwhile the control-driven coordination languages (e.g. Dataflow) show the opposite, hence the term "exogenous".

Figure 3.8 briefly summarises the connection between exogenous coordination and other branches of programming models and languages.

Endogenous coordination is said to perform better in fine granularity coordination [23], where a single shared variable can be coordinated (as in TupleSpace), thus it is used extensively in programming Distributed and Parallel Systems. On the other hand, exogenous coordination models bring the coordination part outside of the computation code, creating a new entity, a coordinator that coordinates the participating computation activities. Exogenous coordination is said to be more suitable in expressing system behaviours, sometimes the configurations of Distributed Systems and fostering the reusability of components.

Typical endogenous coordination models include platforms that are based on the tuple space model [33]. For example, TS-Mid [56], LIME [67] and TeenyLIME [26] are efforts to extend the tuple space model to support distributed tuple space, tuples observance and mobility of sensor network. Sensor nodes in TeenyLIME only share their tuple space with
their one-hop neighbourhoods instead of with the whole network, which can be seen in LIME. This design choice is due to the spatial locality of WSNs in which related sensors and actuators are closely located (fire extinguishers and smoke detectors). TS-Mid opted for a hierarchical system of tuple spaces that splits the network into different clusters with cluster head being responsible for computation tasks.

In these data sharing-based coordination models, all systems rely on a common data sharing medium to cooperate. While the data sharing medium can be used to allow distributed systems to communicate, the communication primitives are always blended in the computation parts. This makes it hard to coordinate the cooperation from the outside. For example, when a system output some data into the common data space, the other system needs to query or read from them. These data output and input procedures require that each system knows where to send the data and from where to read the data back. This knowledge has to be embedded in each system, preventing an external coordinator from intervening with the cooperation among processes.

The situation is different in many fog applications where participating devices are very loosely-coupled and operate independently. The software components are also independently developed and leveraged off-the-shelf. This means these components should not make any assumptions about their operating environment. They might also use different programming languages. Thus, we found that exogenous coordination appears to be a natural approach in developing fog applications as an external coordinator is provided to govern the inter-process cooperation. Interestingly, when discussing design challenges for Cyber Physical Systems (CPS), Lee also found that coordination models and languages promise to solve many large scale and real-time problems of the CPS [54]. This further justifies our design choice for using exogenous coordination as the starting point for developing our fog applications.

It is worth noting that the exogenous coordination approach is suitable for our class of
fog applications where we put a focus on the edge devices. An important characteristic of these fog applications is that they need to be aware of the physical context of the fog devices on which they are deployed. This requirement is due to the dynamic nature of edge devices. For many other fog systems, fog devices are considered stationary and are deployed in a fixed infrastructure [39] [42] [86]. In such systems, physical context of fog devices becomes irrelevant to the applications they are hosting. Since such a deployment environment is not dynamic, an external coordinator is not needed. For example, while a component-based abstraction level is still favourable to decompose applications into subcomponents, they can be structured hierarchically and be self-coordinated without needing an external coordinator (as many of these research bodies have chosen).

3.8 Summary

This chapter reported an analysis of various aspects of an application model, namely, the application scope, abstraction level, application structure, communication and programming model. By understanding these aspects, we have a better understanding of how every application is built, from the simplest to the most complex ones. Ultimately, by understanding how applications are made, we can design better development tools for each of the classes of applications, and their corresponding scopes.

Based on this analysis, we identified the unique features of our interested fog applications. These features stem from the close bonding with the physical context of computing devices on which the applications run, and the large scale, geo-distribution of fog computing resources. We then described the requirements and suggestions in developing our fog applications. Accordingly, our desired application model should follow a combination of the inter-context scope, flat structure, component cooperation abstraction level, bounded communication cardinality and annotation-based hybrid programming model.

While these design choices are suitable for our particular use case, our application mod-
els taxonomy provides a new set of trade-offs that are highly relevant to general fog applications. Thus, it helps developers to make an educated decision on how to proceed with each particular use case.

Finally, we showed that the concept of exogenous coordination appears to be a natural approach in realising such an application model for fog applications. The explicit separation of concerns between computation and coordination activities allows us to leverage independently developed, off-the-shelf software components of possibly different programming languages in constructing complex fog applications.
Chapter 4

Context-dependent Exogenous Coordination for Developing Fog Applications

Fog computing systems are complex distributed systems whose computing resources are geographically distributed in large areas. These systems include computing systems that are heterogeneous and connected to Wide Area Networks via different technologies, such as wired, Wi-Fi, LTE or 5G. In this environment, communication among devices is difficult to manage. The developers might have to know about the deployed systems beforehand in order to code the necessary communication routines. Further, they might also have to cater to individual devices or class of devices to make the most out of their capabilities.

These process if done manually, would significantly raise the complexity of the application being developed. Meanwhile, the application logic itself might be elementary. Let us return to our example application scenario developed in [Chapter 1]. As we can see, such application only has four software components with very simple application logic. However, to be executed in large scale and dynamic fog computing infrastructure, it needs special pro-
programming primitives to express the running constraints. This is different from cloud-based or smaller scale applications where running condition hardly affects the development process. Meanwhile on some applications, the check for running condition is mixed within the computation code [39]. This mixture poses a great challenges in extending the applications and in reusing the computation code.

Therefore, we find the need to explicitly take this running condition logic out of the computation logic and develop necessary programming primitives and constructs to let the developers to express their applications in a flexible and reusable way. This is similar to the work in Aspect Oriented Programming (AOP) [48], where the cross cutting concerns that are orthogonal to the computation activities are represented as annotations outside of the computation code.

While exogenous coordination sets the basis to our application model, existing exogenous coordination models and languages do not have the focus on the coordination in large scale systems, ones that span a large geographical area. That is, in large scale fog computing systems, applications often depend on the physical context of the computing elements, unless they do not involve the interaction with edge devices (which is not in our focus). Second, they involve a large number of instances of the application’s subcomponents.

To support these emerging concerns, we first propose the separation of concerns between application data versus contextual data and show that they significantly improve the flexibility and reusability of the application model. We then describe our application model and the context-dependent primitives that expose the physical context of the underlying computing elements to the application developers. We then discuss the notion of bounded communication cardinality as an important aspect of coordinating fog applications.
4.1 Contextual Data versus Application Data

In large scale fog computing, the physical context of computing elements plays an important role in the correctness of the application logic. From here, we found that there are two orthogonal aspects concerning the application model for fog computing, namely, programming for the application logic and expressing physical context requirements. That is, a suitable application model should be able to express the application logic and at the same time able to weave in the physical context requirements.

This requirement leads us to an important design consideration where we argue that there is a need to distinguish between the contextual data and the application data within an application.

Returning to our example application in Chapter 2, application data are the video streams that are sent between the software components. They are the first class citizen of the application logic so that all the application’s sub-components are designed to work with them (to process the video streams and their downstream results). Meanwhile, the contextual data are the contextual information of the underlying computing elements that run the software components, such as the location of a car.

In an extreme case, where all the cars are static, contextual data do not affect the application logic in any way and can be dismissed. In fog computing, the correctness of application logic is highly dependent on the physical context of the underlying computing elements. One can see contextual data as the data of the computing elements (the hardware) while the application data as the data within the software components themselves (the software).

Traditionally, there is no clear distinction between contextual data and application data in this type of applications. Consequently, the developers have to incorporate the reasoning about contextual data within the application logic. For example, the developers have to conditionally branch the application logic based on the acquired context of the underlying computing elements. This programming model is less reusable and flexible as the routines
Figure 4.1: Separation of concerns: contextual vs application data

have to be rewritten for different applications in different contexts.

To clearly explain this concept, Figure 4.1 shows the difference between these two types of data and how they are used within our application model.

On the lowest layer is the physical layer where physical context data is presented. These are the contextual data of the devices such as their locations, or their operating properties, such as available CPU or Memory resources. While the application data also come from the device’s physical context, such as an image of its surroundings, the main difference is that they are used as the inputs to the subcomponents, processed by the subcomponents and flow among them. Contextual data, however, is used by the system to coordinate the interaction among the subcomponents.

This clear separation improves the reusability of the application as the computation activities within the subcomponents do not work on a mix between the contextual data and the application data. That is, when a new application is developed, the developers only need to describe new contextual requirements for each subcomponent and annotate them with a context-dependent primitive. For example, consider a traditional context-based application procedure is as follows: if (outside(getLocation(A), area X)), disable(A). This simple logic can be presented by a much simpler declarative statement: @In(X) A where subcomponent A is annotated with context-dependent primitive @In(X).

Therefore, we propose that this is an important approach that can improve the reusability of the application model in a complex fog computing environment.
4.2 Proposed Coordination Model for Fog Applications

4.2.1 Participatory Coordination

We consider task assignment and participatory coordination models for coordinating distributed applications. In both cases, the application is seen as a set of tasks to be executed. The tasks could depend on one another, i.e. there is a presence of precedence constraints. In task assignment model, the participants contribute their computing resources and capabilities while the scheduler(s), by running scheduling algorithms, assign tasks to these participants based on some optimisation criteria. In this case, the pool of participants and their resources do not change dramatically over time, thus the dynamic nature of the system depends on the dynamic of the arriving tasks. In a more dynamic system, where participants and their capabilities change over time, the schedulers need to constantly keep track of available resources in the system; consequently, the scheduling overhead is high. In turn, the task assignment model has an advantage when task completion needs to be guaranteed, such as in hard real-time systems.

In the participatory model, tasks are broadcast to all the participants and they decide to contribute tasks that they can execute. In this case, the schedulers do not have to keep track of the participants’ resources, but still have the power of selecting which participants to execute which tasks depending on their contributions. This is similar to the crowd-sourcing application model where people (or their devices) are recruited to solve a particular problem. However, in this type of application, people (or their devices) connect to a centralised crowd-sourcing platform to solve advertised tasks (or aggregate devices’ data) [20], this eliminates the need to coordinate the execution and cooperation of individual participants. In this work, devices contribute the application’s subcomponents that they can run and the coordinator connects and coordinates the contributed components together, to fulfil the application’s logic. In large scale, dynamic fog applications, we argue that this is the appropriate
4.2.2 Workflow

Once the devices participate in the platform and agree to share computing or communication resources, they are notified about any new application being deployed.

The applications are then modelled, in a descriptive way, as a dataflow graph of subcomponents involved. Each subcomponent is an independently developed software that communicates using ports. The application graph is illustrated in Figure 4.2.

The participating device then checks to see which subcomponents they do not have locally. They can determine by themselves if they want to proceed to download missing com-
ponents from the platform’s components repository. In any way, after processing the application graph, participating devices start to synchronise with the platform by updating their physical context as well as the list of subcomponents they can contribute to the system. As we shall see in the following sections, due to the application constraints, each device might not be able to contribute all of the subcomponents even if they have them installed already. Figure 4.3 illustrates our application design and deployment in large scale fog computing infrastructure.

This participatory and contributory fog computing resource model provides maximum flexibility and liberty for participants to ensure a sustainable system model. It also fully supports the dynamic nature of the fog computing systems. That is, if any participating device goes off-line, the overall application can still operate normally as long as other devices are actively contributing their subcomponents. In other words, the participating devices, as well as their subcomponents, are not statically bound to one another.

### 4.3 Context-dependent Coordination Primitives

Formally, a fog application comprises a large number of participating devices, denoted as $D$. Each participating device $D_i \in D$ has a set of properties $P$ that pertains to the device itself. These include, but not limited to, the device’s location, its available memory, or its bandwidth consumption. These properties change over time based on the current operation of the device. Each of these properties, with its variable value, contributes to the dynamic nature of the system in general. Thus, the set of values of all properties in $P$ defines the physical context of the device.

The fog application also involves a set of software components $S$, each of which is modelled as a computation activity with input and output ports. These components are independently developed, so they do not know one another. What they do is grabbing data from their input ports, do the computation and place the results on their output ports.
A composition graph of $S$ then represents the fog application, showing how data are originated, processed and consumed.

To facilitate the context-dependent application logic, we define several helper functions for the context acquisition as well as selecting appropriate devices for a set of software components as follows:

\[
V : D \rightarrow P \quad (4.1)
\]

\[
Vx : V(D) \rightarrow Px \quad (4.2)
\]

\[
F : S \rightarrow D \quad (4.3)
\]

First, function 4.1 captures the value of property set $P$ of a device $D$, which represents the context of $D$. Second, 4.2 is another function that extracts a single property $x$ from the set of device properties $V(D)$. Finally, function 4.3 is used to select all the devices that are capable of running a given set of subcomponents.

The fog application also includes a set of constraints $C$, represents the application-level requirements on the software components set concerning the devices’ physical context $V(D)$. These constraints are defined in the following section.

To enable the developers to express a range of application requirements based on devices’ physical context, we propose a set of context-dependent primitives based on the devices’ physical context.

A context-dependent coordination primitive $\gamma(Si)_P$ that defines a subset of $C$ called context-dependent constraints. In essence, context-dependent constraints are restrictions on the execution of a particular software component $Si \in S$ with regard property set $P$. $\gamma(Si)_P$ is formulated as follows:

\[
\gamma(Si)_P \iff \exists Dm \in F(Si), satisfied(V_P(Dm)) \quad (4.4)
\]
where satisfied is a boolean function provided by the developers to define the required device properties for a software component to run. An example of such constraint is:

\[
\text{onServer}(S_1)_{\{\text{numberOfCores, freeMem}\}} \iff \\
\exists D_i \in D, (V_{\text{numberOfCores}}(D_i) > 8) \land (V_{\text{freeMem}}(D_i) > 2)
\]  \hspace{1cm} (4.5)

Here onServer is the constraint name, numberOfCores is a property of the participating devices \(D\) on which the application can be deployed. This constraint restricts the execution of software component \(S_1\) on only the devices where the number of CPU cores is greater than 8, and the available memory is greater than 2GB.

It is the definition of \(\gamma(S_i)\) that makes our application model a context-dependent model because it takes the physical context of the underlying device where the software components run into account.

More importantly, we propose a second novel type of context-dependent primitive \(\Gamma(S_i, S_j)_P\) that defines the other subset in \(C\) called inter-component constraints. Inter-component constraints are restrictions of the cooperation between any two software components \(S_i, S_j \in S\) with regard the physical context of the devices they operate on. \(\Gamma(S_i, S_j)_P\) is formulated as:

\[
\Gamma(S_i, S_j)_P \iff \\
\exists D_m \in F(S_i), \exists D_n \in F(S_j), \text{related}(V_P(D_m), V_P(D_n))
\]  \hspace{1cm} (4.6)

where related is a boolean function provided by the developers to define the required context-dependent relationship between any two or more software components. Some examples of such constraints are:
\[ \text{nearby}(S_1, S_2)_{\{\text{lat}, \text{lon}\}} \iff \\
\exists Dm \in F(S_1), \exists Dn \in F(S_2), \\
|V_{\text{lat}}(Dm) - V_{\text{lat}}(Dn)| < M \land |V_{\text{lon}}(Dm) - V_{\text{lon}}(Dn)| < N \]

\[ \text{sameHost}(S_1, S_2)_{\{\text{macAddr}\}} \iff \\
\exists Dm \in F(S_1), \exists Dn \in F(S_2), \\
V_{\text{macAddr}}(Dm) = V_{\text{macAddr}}(Dn) \]  

Here \( M, N \) denote the range that defines the nearby situation in one application. \( \text{lat} \) and \( \text{lon} \) are the latitude and longitude properties of the participating devices.

It is worth noting that the implementation of boolean functions satisfied and related in our coordination model is similar to how First Order Logic (FOL) was used in many context-aware projects [43], notably in Gaia [75]. In Gaia, FOL is used to declare the context information in the form of: \( \exists \text{person } x \text{ Location}(x, \text{Entered}, 2401) \).

Such context information is then used to construct rules that denote higher-level context, such as: \( \text{Location}(\text{Manuel}, \text{Entered}, 2401) \ OR \text{Location}(\text{Chris}, \text{Entered}, 2401) \).

Eventually, such logic rules are then used to control the behaviour of the context-aware application, such as:

\( \text{Location}(\text{Bhaskar}, \text{In}, 2401) \ AND \text{StockPrice}(\text{MSFT}, >, 50) \)

\( \text{PlayHappyMusic()} \)

\( \text{Priority} : 3 \)

It has been recognised that FOL provides a very expressive and powerful way to represent context that is independent of the programming language being used [99] [75] [36]. In our work, we leverage this advantage of FOL in describing the context-dependent constraints that are applied to subcomponents of our applications.
As we explained in our literature overview, our work differs greatly from these context-aware applications by the use of contextual data. In our work, the contextual data of fog devices is used to coordinate the cooperation of the subcomponents they are hosting. In contrast, contextual data in context-aware applications are tight to a presence of a user, and are used to drive the core application logic (the contextual data are actually the application data - see Section 4.1).

4.4 Communication Cardinality

With the replicated deployment of fog applications’ subcomponents, the communication among them should be carefully coordinated. The reason is, in deployment, each subcomponent is deployed onto a large number of participating devices. This deployment scenario creates a large number of the subcomponent’s instances. Inter-Component communication then becomes communication among their deployed instances. For example, a fog application might be as simple as two software components connect to one another: a source component which generates data and a sink component which consumes the data. However, in fog computing deployment, thousands of devices can run these two software components. Thus, each device effectively contributes to two instances of them. Therefore, there are thousands of these instances running in the fog computing infrastructure. The coordination question is, therefore, how should the instances of the source component communicate with instances of the sink component.

The answer to this is communication cardinality [58]. Communication cardinality can also be referred to as dependency cardinality, which denotes the semantics of the dependency between any two entities. Cardinality itself is a popular concept, especially in relational database theory, where it expresses the type of relationship between two entities. There are generally four types of cardinality: 1-1, 1-*, *-1 and *-*.

Accordingly, a 1-1 cardinality means one instance of the source component should communicate with only one instance of
context-bounded Communication Cardinality

the sink component. A 1-* cardinality denotes broadcast-style communication where a *-1 cardinality denotes aggregation-style communication. Lastly, a *-* cardinality represents a freestyle communication pattern among the instances so that they are free to send and receive data to and from all the other instances. While this might seem straightforward, it is different from coordination in small scale systems where there is little or no replication of the same software component.

We have seen that, in fog computing, a broadcast message is not always supposed to go to all the participating devices because of the consideration for their physical context. For example, it is common in fog applications that the broadcast of messages is restricted to a particular group of devices that are related to one another (e.g. traffic situation is broadcast to nearby cars). However, when a fog application is deployed in a large area such as the whole city, there are multiple places that such broadcast communication happens. The same phenomenon happens to aggregation-style communication where one device might not want to aggregate data from all the participating devices, but only from the ones that are related.
Due to this notion of restricted communication, special types of cardinality arise as seen in Figure 4.4. That is, in fog computing, we have context-bounded communication cardinality such as 1-N, M-1, N-M, where N and M are any positive integer number, rather than having solely unbounded ones (1-*, *-1). They represent the size of the group that takes part in the group-based communication.

Interestingly, by choosing these appropriately, the application developers can also enable different settings for fault-tolerance or load-balancing capabilities. For example, both 1-N and M-1 cases represent a load-balancing capability where the load of the component on the "1" side is guaranteed to be less than the number of devices on the other side. Meanwhile, N-M cardinality represents a fault-tolerance capability as the same data is replicated to M devices, and each of these can receive the replicated data from N devices.

4.5 Evaluations

4.5.1 Distributed Node-RED Project as a Coordination Platform

Recall from chapter 2, we started the experiments with fog computing using Node-RED, an open source project that features a dataflow-based visual programming model. Since the original Node-RED project only runs as a single process on a single machine, we augmented it to create an application platform for fog computing, the Distributed Node-RED (DNR). Our DNR lets application developers specify the deployment constraints of an application flow so that once deployed, nodes in an application flow can be executed on multiple devices while still connecting to one another to deliver a fully functional application.

We have implemented our context-dependent primitives into DNR to express application constraints based on the device's physical context. These currently include resource-based constraints (CPU, memory), communication constraints (bandwidth) and spatial constraints (location).

The platform is deployed across a network of nodes ranging from hosted cloud infrastruc-
ture, through lab servers to edge devices such as Raspberry Pis or similar. A single server or cloud instance is designated as the coordinator and also serves as the component repository.

Since Node-RED comes with a comprehensive software component library, especially when there are existing components that can interact with Internet services (e.g., Facebook or Google), integrating these components with our fog devices is straightforward. Therefore, a fog application can be easily programmed. Furthermore, with its visual programming interface, the path from design to development has been shortened significantly.

Figure 4.5 shows the development canvas of our DNR platform. To build an application, the developer only needs to drag the software components from the left column to the canvas and wire them together. The application constraints can be added by using the Node Requirement button at the top right corner. The constraint definition dialogue allows defining constraints based on physical locations of the devices, or any of their computing resources, such as the number of CPU cores or the amount of free memory. For example, the developer can request that an individual node should only run within a particular geographical location. In cases where the node is deployed to a moving vehicle, it is activated only when the vehicle passes through a specific area - defined by the geographical constraint.

4.5.2 Applications Developed with DNR

Real-world deployment

During this thesis, we had a chance to work closely with Keio University in Japan under the EU-Japan joint project Bigclout [74] to deploy the DNR platform to Keio’s smart city infrastructure in Fujisawa city.

In our current trial, we employ a fleet of garbage trucks in Fujisawa city as our fog computing platform. Each garbage truck has a camera onboard to capture video footage of the road segments to where they travel. These video streams are analysed to classify the lane markers on the road surface to see if a repaint needs to be done.
Figure 4.5: Building fog applications using the DNR platform
Our DNR platform allows the city developers to quickly build and deploy such application on top of the fleet and associated city infrastructure. For example, in Figure 4 of their related publication [47], they can annotate subcomponents of the application with context-dependent primitives that dictate under which context each of those subcomponents should operate. Thus, a unified development environment is provided to take advantage of computing resources across the network edge (computing elements installed on the garbage trucks) to the cloud (their backend server).

**Lab-based deployment**

Since evaluating such an application in a real-world setting is difficult, a lab-based implementation has been developed that mimics the core application scenario. In our application, we take video streams from security cameras mounted on lamp posts to recognise and classify cars running on the streets. While it seems contrived, it can be extended easily to convey a more serious application scenario, such as in searching for a suspect car using the city’s security camera infrastructure.

In our evaluation setup, we have two embedded boards, one desktop computer and one laptop that participate in the distributed application. About the software components involved, there is one component that captures the camera feed (through playing video footage), the data is sent to a background subtraction component to extract the foreground running cars, which are subsequently classified by a neural network classification model.

The lab-based experimental setup is seen in Figure 4.6. In our setup, we name the desktop computer 152, the two embedded boards 117 and 198. For the sake of simplicity, these numerical identification is manually named after their IP addresses in our local network. For example, if one’s IP address is 192.168.1.117, it is named 117.

Since this naming becomes irrelevant in large scale deployment, our DNR platform can constrain the execution of software components on devices using their physical contexts such as the level of memory usage, or their location. To simulate this, device 117 and 198...
Figure 4.6: Lab setup for our example fog application.
are configured with a predefined latitude and longitude (this is a common practice in case of lamp posts deployment). Our example application is then designed such that, the video source and background subtraction components are constrained to run on a desktop computer while the neural network image classification component is constrained to run at a certain location (e.g. downtown). On deployment, the desktop computer resolves to the device 152 and the "downtown" devices resolve to the device 117 and 198 (two embedded boards).

In our lab-based experiment, it is difficult to simulate the mobility behaviour of the participating devices (i.e. dashcams mounted on cars). While we can undoubtedly replay car traces from available open data sets, or use traffic simulation suite such as SUMO [51], the small number of participating devices make it irrelevant to have moving devices while still keeping the data streams continuous. In a real-world scenario, with a large number of participating devices (i.e. cars, lamp posts, traffic elements), maintaining the data streams among them is more manageable. When a car moves out of the coverage of a fog node, e.g. wireless-enabled street light, another one could move in immediately, allowing the fog node to get input for its processing task continuously. In our lab-based experiment with only a limited number of devices available, the data streams are less stable; thus, the outcome is intermittent.

Since the mobility of devices in the studied scenario is used to show the dynamic nature of the system, we opted to look for a different approach to study this dynamic characteristic. We realised that in fog computing since the computing activities are tightly bound to the physical context of the devices, the system load for each device also frequently fluctuates. For example, when we process camera footage of a street scene, the number of cars extracted from the footage varies from second to second. Thus, we use the system load factor to illustrate the dynamic nature of the system in our lab-based experiment and incorporate a load balancing feature into our lab-based setup. To do this, a device capability constraint that is based on the network load is created and applied to the image classification component.
Thus, this component is constrained so that it only runs on devices with a required amount of network bandwidth.

When the number of extracted cars is small, the input to the image classification component is small; therefore, the network load of the underlying device is low. The device continues to do the job. When this number increases, the underlying device experiences higher load and, through context synchronisation, the external coordinator starts to route the data stream to a second device that also satisfies the constraint to run the image classification component.

This approach essentially demonstrates the load-balancing feature of the system, which, in turn, illustrates its dynamic nature. It also illustrates that, to support the dynamic nature of fog applications, the coordination platform has to periodically monitor the devices’ network bandwidth and other resources to make the coordination decision as to which device to send the car image stream to.\(^1\)

Since we let the developer to explicitly specify the network bandwidth threshold at which the load-balancing coordination is triggered, the example also shows how the physical status of the underlying computing infrastructure (e.g. current network consumption of devices) can affect the correctness of the application logic (e.g. only run this component on devices that have minimum available bandwidth).

Our lab-based setup allows us to verify the behaviour of our example application when it is distributed to a number of devices. It also shows that the application model itself would support system scalability as it is participatory-based and does not rely on any specific deployment details such as what types of devices are involved, where are they located, which network connection they are on, and so on. Thus if a device decides that it cannot run a particular component, it just needs to move on with other components. Meanwhile, the external coordinator always has the collective component instances contributed by the devices;

\(^1\)As we shall see in the next chapter, this process is costly in terms of coordination overhead, which leads to our next contribution in building a large scale, dynamic coordination platform
Figure 4.7: Fog applications design and development in DNR.
it can coordinate the communication among those available instances so that the application’s requirements are met. In large scale applications, this is an essential characteristic of the application model, which is also known as a macro programming approach.

Figure 4.7 shows the development of our example application using DNR. As can be seen, the application model consists of a data flow graph of the software components involved (top of the figure) and the context-dependent constraints that are applied onto each software component (bottom of figure). When the application is deployed onto the pool of participating devices, each device processes the application and all the associated context-dependent constraints. It then reasons about whether or not to enable certain components, as well as to fetch/redirect data from/to any external devices.

Besides supporting an application model based on exogenous coordination, DNR extends the capability of the Node-RED project by allowing developers to distribute the application flow onto multiple devices as well as to express context-dependent constraints in their applications. These are essential capabilities as they now have the tools needed to deploy complex applications that take advantage of the large scale, dynamic and distributed fog computing environment.

Other Application Scenarios

Industry Example: wizr.com - During this thesis, we had a chance to work with wizr.com, a US-based startup that delivers intelligent edge processing solution to clients who possess a large number of security cameras.

A typical client of wizr.com might have as many as eight hundreds of security cameras surrounding its venue (e.g. manager of a stadium). However, security staff whose tasks include the monitoring of these cameras can sometimes be very limited. Wizer’s solution was to deploy an edge computing platform that consists of embedded boxes that are capable of doing AI-based vision processing within the client’s venue. Each embedded box can
process as many as 20 cameras, resulting in a total of 40 boxes for a typical client.

Common application logic is, once suspicious activity is found by one of the embedded boxes, a warning sound is sent to other nearby boxes that have a speaker attached to it, to be played. This pattern illustrates an essential aspect of our fog applications, which is context-dependent coordination logic.

Clearly, such a use case can be realised using our coordination model by treating the video stream to be the main application data whereas the location (context) of each computing device to be the contextual data. Thus, the output of one processing component can be sent to nearby speakers to be played. Additionally, since there are many speakers and cameras, as well as computing devices within the network, context-bounded communication cardinality provides an important coordination primitive to properly control the stream of data within the network.

**Digital Signage Systems** - In big cities with many shopping centres, Digital Signage Systems are prevalent. These systems consist of large digital displays that are assigned with various advertisement content, such as images or video. In many places, a context-aware
ads-placement is demanding where the nearby physical context influences the content of the ads displayed. Examples of these contexts are, the crowdedness of the shopping locations or city squares, the noise level, whether there is any special event currently going on or simply the number of people standing in front of the display.

We assume that the cities can leverage their existing computing infrastructure such as their surveillance camera network, noise sensors, city’s communication backbone or traffic elements to produce contextual information of the real scenario and use these data to drive the ad displays for their smart bulletin boards.

In this example, the camera network, city’s communication backbone, traffic elements and the digital displays represent the large scale, widespread distribution of fog computing infrastructure, which is used to deploy context-aware bulletin boards application. To implement such an application, it is desired that the displays only based contextual data produced by nearby cameras or noise sensors to deliver their content.

Similarly to the earlier use case, we can easily model this scenario using our coordination model. That is, we have the location of such devices to be the contextual data to our application. The camera streams and eventually, the crowdedness level extracted from those stream become the application data that are communicated among the participants. The ad display component is then based on the crowdedness level data to decide how it should display the ads.

*Ensuring Quality of Service* - In some applications, Quality of Service (QoS) is an important criterion in assessing the correctness of the application logic.

While our work does not tailor specifically to QoS-aware applications, a similar mechanism can be achieved by adding priority constraints to our coordination model in addition to the context-dependent primitives. Since our the participating devices follow the same coordination protocol, such priority constraints serve as a guideline for how individual participants can schedule their resources appropriately. Priority constraints can also be applied
to communication links in addition to the subcomponents to expressing the QoS for inter-
component communication.

4.5.3 Analysis

Let us revisit to our second research question, namely, (RQ2) *How fog applications can be
developed with minimum requirements on the developers to cope with the complexity of fog
computing infrastructure (large scale and dynamic)?*

It is clear that by separating contextual and application data, we leave the technical bit
of acquiring and abstracting contextual data out of the computation activity. Since these
are well-discovered topics in context-aware research community, it is safe for us to assume
the development of fog applications can take these capability for granted. Effectively, our
fog application developers do not need to care about the complexity of the underlying fog
context.

Further, by employing exogenous coordination and developing necessary primitives, the
computation activities are highly abstracted from the heterogeneity in devices capability as
well as programming language aspects. While these primitives are simply first-order predi-
cates, such simplicity makes our coordination model highly accessible, meaning developers
do not have to go through a steep learning curve to build application for the complex fog
computing infrastructure. It is also highly flexible as various requirements can be captured
by combining such simple primitives together. For example, "a rainy afternoon" is
a requirement that can be composed of two context-dependent primitives based on weather
and time. In many cases, this is the desirable approach for building large scale and dynamic
fog applications. However, certain types of fog applications such as mission critical applica-
tions will need a more complex set of primitives. This is to describe more stringent deadlines
or quality of service requirements. Since these primitives are specifically catered to such use
cases, it is difficult to combine them to express different requirements. In a broader perspec-
tive, this comparison represents a high level trade-off between complexity of the primitives

78
Regardless of this trade-off, the use of exogenous coordination allows fog computing developers to leverage software components of different languages and glue them together in a complete fog applications. Our evaluated application scenarios show the advantage of such an approach. For example, our lab-based application experiment is developed from many components of different languages. Notably the visualiser components are written in web languages such as Javascript and HTML, the background subtraction component is written in C using OpenCV library and the neural network car classification component is written in Python using Caffe.

Furthermore, the very similar components are used in the real-world field trial in Fujisawa. This demonstrates the flexibility and reusability of the application model when it comes to developing large scale fog applications.

Therefore, our context-dependent exogenous coordination model proves to support fog applications developers by abstracting away the complexity of fog computing environment.

4.6 Summary

In this chapter, we showed that exogenous coordination is a suitable approach and a great starting point for supporting the development of fog applications. We also showed that coordination in a fog setting needs to incorporate the contextual data of the participating computation activities and that the clear separation of contextual data and application data makes it easier to separate communication and computation activities.

We developed two context-dependent primitives that expose contextual data of the underlying device to the application components running on top of it. We also addressed the importance of coordinating the context-bounded communication cardinality among instances of application’s subcomponents.

To evaluate the proposals, we developed DNR, a coordination platform for fog comput-
ing that is based on exogenous coordination. DNR provides necessary context-dependent primitives that allow developers to express a range of common fog application requirements without having to care about the complexity of underlying fog computing infrastructure.

While we focused mainly on the coordination primitives and feasibility of the application model with DNR, in the next chapter we discuss the challenges and our approach in bringing such an application model to large scale fog computing system such as a smart city.
Chapter 5

Coordinating Large Scale, Dynamic Fog Computing Applications

The application model described in the previous chapter provides a clear and expressive way to develop applications in complex fog computing systems. In this chapter, we discuss how such application model, notably with its context-dependent primitives and communication cardinalities, is coordinated in large scale fog computing infrastructure to meet all the application requirements.

5.1 Design Considerations

While exogenous coordination sets the basis for our application model, existing exogenous coordination models and languages do not have the focus on the "coordination in the large" problem where software components are replicated and deployed in a large number of geographically distributed devices. To date, these models and languages only pay attention to the software components themselves rather than the underlying computing elements on which they are operating.

In large scale fog computing systems, the coordination model has to take into account, first, the underlying computing elements of the software components and second, their large
number of replicated instances. In our application model, the former requirement manifests itself in the context-dependent primitives that govern the communication among subcomponents based on the context of their underlying devices. The latter requirement is considered by introducing special communication cardinalities that govern the communication among the subcomponents’ instances.

### 5.1.1 Large Scale Coordination

Large scale deployment of subcomponents requires a fine grain coordination task at the individual device level. The reason is that the devices have various contexts that might affect the execution of the application logic. Thus, when a subcomponent is deployed into the infrastructure, there are a large number of its instances. Each instance runs on a particular device under a specific context. The coordination platform has to coordinate at this subcomponent instances across the deployed environment.

One of the apparent considerations is that the coordination platform has to be aware of the communication pattern among instances of subcomponents. Unlike parallel data processing components in data centres who collectively process a large amount of data, the deployment of software components in wide-area fog computing environments results in a multitude of component instances that are independent and different from one another. The data processing pipeline in fog computing is, therefore, inherently distributed from data acquisition to consumption.

In our example application scenario in Chapter 1, data are acquired by many cameras across the city, and they are routed to appropriate fog computing elements that are also widely distributed; the final processing results are then fed into relevant dashboards. While similar data processing pipeline can be trivially realised in the data centre environment, such fog-based data processing pipelines need to be coordinated appropriately in large-scale settings. The reason is, all of the participating computing elements should be properly cooperate with each other to keep the application logic correct concerning their physical context (i.e.
processed data from some particular dashcams should not be sent to a dashboard in a separate location.

Another consequence of the large-scale distribution of components is their bounded communication cardinality, which again should be coordinated appropriately. For example, at one given location, a dashcam should only send its data to some processing peers instead of freely broadcast its stream to all the participating devices. If this number is too big, it is possible that the data stream has reached a peer at a different location that eventually could violate the application’s context-dependent logic. The coordination logic behind this is, therefore, has to take into account the context information beyond the traditional technical metrics such as throughput, availability or scalability.

5.1.2 Coordination Overhead

As discussed in the previous chapter, the large-scale deployment of fog computing resources pushes them closer to the physical world, making their physical context an important part of the application model. To enable context-dependent coordination primitives, the platform has to periodically capture and reason about the context of all the computing elements. Coordination decision is then made based on the up to date context of the system. This process of re-evaluating the situation of fog computing resources is costly due to a large number of participating devices. That is, whenever the application platform has to coordinate the communication among the instances of the subcomponents, it has to take into account a large pool of devices to generate an appropriate coordination decision.

To support our inter-component coordination primitive, the coordination platform has to select appropriate communicating peers (instances of subcomponents) and bind them together so that groups of peers who all satisfy the application’s requirements can be generated. With some applications that have complex constraints, this becomes an NP-complete problem where the platform has to try many combinations of peers to generate satisfiable groups. Given a large number of fog computing elements, the search space can easily explode.
This proves that building a context-dependent coordination platform for large scale fog computing can be very costly. Thus, in this chapter, we carefully look at the coordination overhead issue in designing the coordination platform so that a large number of participating devices are supported.

### 5.2 Large Scale Coordination Problem Formalisation and Issues

In order to fulfil the proposed context-dependent coordination primitives, we model our coordination problem as Figure 5.1. We assume that the number of participating devices \((D_1, D_2, \ldots)\) is potentially large, in the order of thousands or tens of thousands of devices while the number of software components \((S_1-S_4)\) is small (less than 10). This assumption can be justified given our fog applications that were described in Chapter 2. Each of these devices can decide to contribute some or all of the subcomponents involved. This effectively generate a pool of the subcomponents’ instances that are identified by the subcomponents and the hosting devices. For example, in this table we have the following instances for sub-

<table>
<thead>
<tr>
<th>Device</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>D3</td>
<td>O</td>
<td></td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 5.1: Coordination Problem Formalisation**
component S1: S1D1 and S3D1. Beside this table, the fog application also has a set of constraints C that are imposed by the developers. The task of the coordinator is to deliver combinations of the subcomponents instances so that the constraints are met, for example, S1D1, S2D1, S3D2 and S4D3 can be one possible combination provided that the constraints are met (e.g. S1 and S2 are nearby).

Due to the large number of devices and their geo-distribution over large areas, the coordinator has to calculate as many such combinations as possible to utilise the contributed instances of the involved subcomponents.

Depending on the constraints and requirements of individual applications, we can solve this coordination table in different ways. For example, with applications that do not have inter-component constraints, simple procedure with polynomial time complexity is sufficient in delivering the right assignments of S to D. Meanwhile, for applications that have inter-component constraints, constraint solvers (e.g SAT solvers) might be used. In extreme cases where the applications are so complicated that many inter-component constraints are used, even the most efficient constraint solvers might fail to converge. The coordination platform we develop for our study provides the flexibility to swap in different algorithms (i.e. solvers) for different application scenarios.

Our focus here is to study the coordination problem in large scale and dynamic scenarios, i.e. how the large scale and dynamic nature of fog devices affect the design of the coordination platform. That is, from this point of view, regardless of the algorithm choices, the supporting system still has to deal with the large scale and dynamic nature of the computing infrastructure. Although the complexity of application constraints also affect the way such a coordination platform is designed, and specially the way algorithms are chosen, we leave the study into this topic for our future work.

In this thesis, since we aim at providing a solution for a general scenario where there is a moderate level of constraints complexity (the number of subcomponents involved is small),
we model our coordination table as a Constraint Satisfaction Problem (CSP) as follows:

– Variable: the involved set of software components - \{S_1, S_2, \ldots S_n\}

– Domain: the participating devices - \{D_1, D_2, \ldots D_m\}

– Constraint: application-level requirements on the execution of S - \{C_1, C_2, \ldots C_k\}.

These are expressed using our proposed context-dependent coordination primitives in Chapter 4.

– Goal: deliver the right assignments of S to D that satisfy all constraints C.

The definition of these variables can be found in Chapter 4. Figure 5.1 further illustrates this coordination problem modelling.

To solve this CSP, we need to meet two requirements, namely, to support a large number of participating devices (i.e. domain space), and to fulfil the application constraints given the dynamic nature of the application.

First, due to the large domain space, solving this problem proves to be resource consuming, which potentially overload a single constraint solver. The standard solution here is to exploit a distributed cluster of constraint solvers who collectively address the problem. The coordination table is, therefore, divided into smaller chunks where each of them is assigned to a constraint solver within the cluster. An essential question arises, which is how many constraint solvers do we need and how can we provision an optimal number of solvers given the frequent change in the physical context of the participating devices.

Second, even though we could potentially solve such large scale issue with a distributed collection of constraint solvers, the results of solving such coordination problem can be invalidated shortly after they are derived. This issue is due to the changing in the context of the participating devices, which invalidates the fulfilment of application requirements within the current system state. Thus, a dynamic, periodic coordination activity has to be maintained. This is costly with our large scale fog computing system.
To answer the first question, we conduct an experiment to study the capability of the constraint solver when the number of participating devices increases. We then study the effect of adding more coordinators into the system and observe the application performance in different scenarios (e.g. dynamic and static). The findings are reported in Section 5.5.

To answer the second question, we propose an incremental coordination approach that based on past coordination evaluations to reduce the domain space. The following section describes this question in detail and our approach.

### 5.3 Incremental Coordination for Dynamic Fog Applications

The dynamic nature of the edge network requires the coordination layer to be also dynamic. Since there is no prior knowledge of how the system is going to behave and it is also unpredictable, a periodic coordination execution is required to ensure the current combinations of software instances still meet the application’s context-dependent constraints. This continuous coordination activity is costly due to the resource consumption of constraint solvers.

Our hypothesis here is that even though the context of the system constantly changes, the context-dependent constraints might not be invalidated immediately. For example, if a dashcam component is constrained to take video at a certain location, this requirement is continuously met unless the car moves out of such a location boundary. The same observation applies to a situation where two components are constrained to communicate only when they are nearby. That is, if such two components are deployed on two cars that are nearby and move in the same direction, the requirements are going to be met until they go apart.

We propose an incremental coordination approach where we exclude instances of sub-components that still meet the context-dependent constraints on subsequent coordination execution. This can be done easily, given that the context of the participating devices is synchronised with the coordination platform. Besides, peer devices can also work together
directly to decide if they should participate in the subsequent coordination executions based on their current context (see Section 5.4). As a result, we can see that the coordination table is significantly reduced in subsequent runs of the constraint solvers due to the exclusion of such instances of the subcomponents. Because of the reduction in size of the coordination table, our approach works not only for our chosen algorithm but also for other implementations as well.

5.4 Implementation

Based on our participatory application model, our coordination platform consists of two types of coordinator, a local coordinator running onboard of participating devices and a coordination platform that includes a collection of constraint solvers running in the cloud.

The local coordinator is responsible for context-dependent constraints that apply to a single subcomponent of the application. Since the constraints only involve the context of the underlying device, the local coordinator can coordinate the communication of such subcomponent based on the device’s context.
The coordination platform is needed when there are inter-component constraints applied to more than one subcomponents. The reason is, when two or more subcomponents are involved in an inter-component constraint, each of them might be running on a different device. Since one subcomponent running on one device cannot infer about the context of the other subcomponent running on the other devices, these devices need to communicate with a centralised coordinator who has a global view of all the participating devices. The coordination platform is also needed where a pair of subcomponents have special communication cardinality that requires their precise identification.

Figure 5.2 presents our system architecture design. Our system consists of three layers: coordination platform layer, coordination medium layer, and participating devices layer.

5.4.1 Participating Devices

Participating devices are the central computing infrastructure where fog applications are deployed. They operate in a dynamic environment where they can physically move from place to place and interact with the physical world. For some static devices, their computing workload might also be very dynamic based on the captured data of the surrounding environment (e.g. counting the number of people in shopping centres and analysing their mood).

We assume that participating devices are heterogeneous and managed by different organisations. While we do not impose any standard middleware or runtime to be deployed in these devices, we assume that they follow our platform’s protocol of application development and deployment. That is, we do not control the individual devices, but rather, have a standard protocol that they should follow if they want to contribute their computing resources. This design choice obviously helps to deal with the dynamic nature of fog computing system. That is, if some devices are not available (i.e. moved out of the network coverage), they stop contributing their resources. Unlike task assignment coordination model, this device unavailability hardly affect the system operation in general because 1) the large number of participating devices ensures a certain level of service fulfilment given that the majority of
them can contribute resources; and 2) the coordination platform periodically execute the coordination problem, thus it should be able to keep up with any changes within the system, including the fluctuation in devices’ availability.

We describe the procedure when an application is available to be deployed as follows.

When the coordination platform notifies participating devices about new application available, the local runtime downloads the application description and starts the onboarding process. First, it filters all the subcomponents in the application graph that are not locally available (i.e. not installed) and decides on its own whether or not to install any of them. This decision process is totally up to the policy of the devices’ owner, which is out of the control of our coordination platform. If a subcomponent is considered to be not available
locally, the local runtime marks it in the application graph as unavailable. Second, for every edge in the graph that represents the communication link between two subcomponents (including the unavailable ones), a local coordinator is spawned and injected into such edge to intercept and coordinate this communication. The local coordinators, based on the application’s constraints being expressed by using our context-dependent primitives, decide how to proceed with the data flowing between the pair of subcomponents they coordinate. They also subscribe to the local runtime to receive the contextual data of the devices. These contextual data are matched against the application’s constraints and are used as the basis for the coordination decision.

There are generally four possibilities or coordination states in which these local coordinators can be. Figure 5.3 illustrates this process. When communication cardinality between any two subcomponents is taken into account, there are two other possible states. We describe the process of deriving these states as follows. The decision tree is also depicted in Figure 5.4.
In the NORMAL state, the local coordinator passes the data through its outputs, thus acting like a regular communication link between the pair of subcomponents (A and B in this case). NORMAL is the state where the local device satisfies the context-dependent constraints regarding both A and B.

There is a DROP state where the local coordinator drops all the data they receive. DROP state is useful when the local coordinator believes the data have already been consumed elsewhere. For example, consider a scenario where a subcomponent A does not have any constraint while subcomponent B does. In this case, all devices enable the execution of subcomponent A while some devices disable the execution of subcomponent B due to the unmet constraints of B. On such devices, the local coordinator drops the data generated by their subcomponent A because they cannot run the subcomponent B. However, a more important reason is that they believe there is no other instance of subcomponent B who needs such data; they then decide not to forward the data to an external device.

Note that here we assume that all participating devices have the subcomponent A installed. Otherwise, there are instances of subcomponent B (where an instance of subcomponent A is not available) that need the data input from subcomponent A. In such cases, DROP state should not be used.

Another state is the FETCH_FORWARD state, where the local coordinator requires to get data from an external instance of subcomponent A and forward the data to subcomponent B. This happens when the device can run subcomponent B but not subcomponent A.

Lastly, a similar state with FETCH_FORWARD is the RECEIVE_REDIRECT state, where the local coordinator sends data from subcomponent A to an external instance of sub-component B. Again, this happens when the device can run subcomponent A but not subcomponent B.

When communication cardinality is taken into account, there are two other states which are COPY_FETCH_FORWARD and RECEIVE_REDIRECT_COPY. Similar to FETCH_FORWARD
state but when the communication cardinality between subcomponent A and B is N-1, the subcomponent B needs to get data from N other instances of subcomponent A. This also includes also the co-located instance of subcomponent A on the same device. Therefore the local coordinator has to get data from the local instance of subcomponent A instead of only the external devices, hence the COPY_FETCH_FORWARD state. A similar observation holds for 1-N communication cardinality where the RECEIVE_REDIRECT_COPY state is needed.

In these last four states, there is an extra piece of information that has to be determined to construct the state: the destination to which the data is sent, or the origin of the data to be received. If the communication cardinality is either 1-1, 1-N, N-1, or N-M, they need to specify a particular instance of the external subcomponent for the data. In this case, the coordination platform plays its role by providing the external instance of the subcomponent. In rare cases, if the cardinality constraint is *-* (i.e. unbounded cardinality), the local coordinator does not need to specify any particular instance for the communication because it happens freely among the instances of subcomponents. In this case, the target for the data is merely the name of the target subcomponent (B in this case).

To engage with the coordination platform, the local coordinators periodically send two messages to the coordination platform. The first message is called instance contribution message, which contains the sub-component instances that the local device can host depending on the application constraints. The second message is called instance request message, which asks the coordination platform for a peer instance that is required to form a communication. These two messages can be combined into one request package. Depending on the application constraints (i.e. with or without inter-component constraints), these messages might be routed directly to the communication network or the centralised coordinator. Where needed, the local coordinators receive instance assignments responses, which specify the external instances of a subcomponent with which, the local subcomponent should
communicate.

From time to time, each local coordinators update their state based on the physical context of the participating device. This activity effectively coordinates communication among the subcomponents of the deployed application. They also periodically synchronise their local devices’ context with the coordination platform so that context-dependent application constraints can be coordinated.

5.4.2 Coordination Medium

To coordinate the communication among instances of the application’s subcomponents, we based our platform on publish/subscribe communication model. As we shall see in the next section about the role of the centralised coordinator, the use of a publish/subscribe communication model is not strictly required. That is, in our modern Internet environment with the wide adoption of Internet Protocol version 6 (IPv6), it is possible to coordinate the communication among instances of subcomponents using direct communication between devices. However, this peer-to-peer communication model is only suitable for the 1-1 communication cardinality where any instance of one subcomponent should send the data to only one instance of the other subcomponent.

In our fog computing setting, the vast majority of communication patterns require at least some levels of replications of the data messages, such as in broadcast-based communication, or when enabling load balancing, fault tolerance capabilities. This characteristic makes the direct communication model inefficient. For example, in broadcast-based communication, due to the limitation in the device’s bandwidth, it might be inappropriate or not possible for an instance of a subcomponent to broadcast its data directly to a large number of peers. In this case, exploiting an intermediate message broker is useful for the data dissemination.

Thus, we opt to take a publish/subscribe communication model as the underlying coordination medium in our platform. We build on top of this and propose a unique publish/subscribe topic naming scheme for our coordination model.
Subcomponent Instance Identification

Any instance of a subcomponent is identified by the subcomponent’s name and the identification of the underlying device that hosts such a subcomponent. Recall that when participating devices host the application’s subcomponents, they contribute an instance of it. Thus, the device’s identification becomes the identification of the instance of such subcomponent. Since our platform operates in a participatory model, the device’s identification should represent the hardware rather than the software generated one. Thus, the MAC address of the network interface or device’s serial number are some good options. It is also possible to have a mix of this identification among devices, as long as it remains globally unique across the platform.

Publish/Subscribe Topic Naming

In our platform, every communication between any two subcomponents is realised by having one subcomponent to publish to a commonly agreed topic and the other to subscribe to it. It is the naming of this topic that binds instances of subcomponents together. Thus the publish/subscribe communication model with our topic naming scheme become the coordination medium of our platform.

To illustrate this, consider two subcomponents A and B who are dependent on one another such that data flow from A to B. That is, A is the data source, and B is the data sink in this setup.

The most basic form of communication among these components is with the *-* cardinality. This form is where communication happens freely among all the instances of these subcomponents so that, any instance of A can send data to any or all the instances of B. In this case, the topic naming to bind these subcomponents instances together is simply the names of the subcomponents without any identification of their instances. This is denoted as A-B, or generally according to our dataflow-based application model, <source_component> – <sink_component>.

Other slightly more complicated forms of communication are with 1-* or *-1 cardinal-
ities. In 1-* cardinality, only one instance of A is active at any time, who broadcasts the data to all instances of B within the system. A similar setting is with the *-1 communication cardinality. In this case, the naming of the publish/subscribe topic involves the identification of either A or B.

That is, for 1-* case, all the instances of A publish to the topic named $A_i - B$ where $A_i$ denotes the identification of such instance of A (i.e. $i$ is the identification of the underlying device that host A). Accordingly, all the instances of B subscribe to the same topic structure, except that the $i$ part is unknown. Since there should be only one instance of A available to all instances of B, they can subscribe to a particular topic named $A_{\text{active}} - B$. The platform then selects one instance of A to rename its publish topic from $A_i - B$ to $A_{\text{active}} - B$, thus allowing all the rest instances of A to become inactive.

A similar pattern applies to the *-1 communication cardinality. The common topic, in this case, is $A - B_{\text{active}}$, where the platform selects one instance of B to rename its subscription topic and allows all the rest to be inactive.

Apart from these basic communication types, all others require, to a certain extent, the involvement of an external communication coordinator. This is where our coordination platform differs from the existing works. The reason is that all the remaining types of communication involve specific instances of a subcomponent. These are 1-1, 1-N, N-1 and N-M cardinalities and are described as follows.

In the most extreme case, in 1-1 cardinality, one instance of A only communicate with one instance of B. Since there are many instances of both A and B, there are many pairs of such communication. The publish/subscribe topic naming is then $A_i - B_j$ where $i, j$ are the instance identification of A and B. All of these instances need to know who is their peers so this requires an external coordinator to support the local coordinators in finding the corresponding peer instances.

In the most relaxed case, N-M cardinality, at any time, N instances of A should com-
municate with M instances of B. This is done by creating instance groups for both A and B. Accordingly, the communication broker creates and assigns instances of both A and B into groups. The local coordinators are notified about which groups their associated sub-components are assigned. The publish/subscribe topic naming is then the sub-component name and the group identification (rather than sub-component instance identification). That is, instances of A now publish to a topic named $A_{gi} - B_{gj}$ to where instances of B subscribe.

In these cases, the instances of A and B have to refer to the external coordinator to obtain the group identification for their peer communication (e.g. instances of A request for $gj$ and instances of B request for $gi$). It is worth to recall that is communication cardinality represents the fault-tolerance or replication capability of the platform where data are replicated to be processed by multiple sinks.

In remaining cases, which are 1-N and N-1, this is completely different from the 1-* and *-1 cases. That is, in 1-* case, there is only one instance of A active at any time; however, in 1-N case, many instances of A can be active at any time provided that any instance of B does not receive data from more than one instance of A. The same situation applies to N-1 case. Thus, the publish/subscribe topic naming is then $A_{i} - B_{gj}$ or $A_{gi} - B_{j}$. In these cases, the instances of A and B have to refer to an external coordinator to determine the corresponding identification for subcomponent instances and instance groups.

In the next section, we discuss the role of the coordinator in supporting the local coordinators with these communication cardinalities and in coordinating inter-component constraints.

5.4.3 Coordination Platform

The role of the coordination platform is to support the local coordinators in situations they cannot by themselves find the right communication peer for their local subcomponent instances. These situations are when they need to find the specific external peer for the application data. There are generally two application scenarios, to coordinate some of the
communication cardinalities and the inter-component constraints.

Recall that the local coordinators engage with the coordination process by sending instance contribution and instance request messages to the coordination platform. They also synchronise their context with the coordination platform. The coordination platform based on the instance contribution messages to build a repository of subcomponent instances available throughout the system. This repository and the context store are used to answer to the instance request messages from all the devices.

To do this, the repository of subcomponent instances (i.e. the content of the coordination table) is used as the input to the constraint solvers. The context store is used also by the constraint solvers to verify the application requirements against the context of the devices. Since we have a cluster of constraint solvers, such a repository and the context store are divided into chunks and assigned to the constraint solvers. There are different methods to implement this division and assignment. For example, based on the nature of the fog application being developed and the deployment of constraint solvers, the division could based on the proximity between the devices and the solvers. While such specialised assignment methods would produce better performance in certain application scenarios, they might need special algorithms to be implemented. In our implementation, we choose a uniform association to achieve the generality of the approach and also to minimise the implementation overhead.

5.5 Evaluations

5.5.1 Omnet++ Simulation

To evaluate the coordination platform in a large scale setting, we use the Omnet++ network simulator. Figure 5.5 shows our network design in the Omnet++ simulator. The network consists of one coordinator that oversees all network elements and a number of mobile and fog devices. The mobile devices are configured to move with a certain speed and angle,
Figure 5.5: Omnet++ Simulation Model for Large Scale Fog Applications
Table 5.1: Omnet++ Simulation Configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>moving speed</td>
<td>normal(mean: 10mps, std: 20mps)</td>
</tr>
<tr>
<td>change angle</td>
<td>normal(mean: 10deg, std: 20deg)</td>
</tr>
<tr>
<td>sensor stream rate</td>
<td>every 20s</td>
</tr>
<tr>
<td>nearby definition</td>
<td>within 300m in both axes</td>
</tr>
<tr>
<td>simulation area</td>
<td>1000 Sq m</td>
</tr>
<tr>
<td>simulation time</td>
<td>300s</td>
</tr>
</tbody>
</table>

which vary over time. Communication among devices is direct Omnet++ point-to-point channels with pre-configured communication delay. The simulation configuration can be seen in Table 5.1. The fog devices are configured to be stationary and randomly distributed over the simulation area. The simulation area is configured to be a large square of 1000 square meters.

The application is represented by three software components that are connected. The application is constrained so that the stationary fog devices can only run the data processing nodes (node b). Also, the data source and sink nodes cannot be located on the same device. Lastly, node a and node c are constrained to only participate in an application flow if they are 300 meters or closer to one another. Apart from the data processing node, node b, all other nodes run on mobile devices. Each device also has its local coordinator which periodically updates the centralised coordinator with the device’s location (i.e., x,y coordinates in Omnet++ simulation).

5.5.2 Evaluation Metrics

There are two performance metrics used in the evaluation. The first metric is the performance of the coordination layer, which is the total number of subcomponent instance compositions generated from the context store and instance repository. This is normalised by dividing this total number of subcomponent instance compositions to the number of subcomponent instances who need to participate in a composition. Accordingly, 1 indicates that all sub-
component instances can participate in a compositions while less than 1 means there are instances that cannot find their peer (i.e stall resources).

The second metric is the application performance metric, which is the message delivery ratio. This metric is calculated as the ratio between all the messages received at all the sink instances versus all the messages sent from all the source instances. This metric is also normalised based on the case where all devices are stationary.

### 5.5.3 Results and Discussions

We first study how the coordination layer copes with the increase in the number of participating devices. Figure 5.6 shows the results of varying the number of devices from 50 to 300 for both categories (stationary fog devices and mobile devices).

In this graph, 1 means all devices are getting their needed coordination solution. That is, they all know their peers to which they should send the data to. On the other end, 0 means none of the device knows where to send the data to. As the number of devices increases, the domain space of the constraint solver increases proportionally. When there is only one coordinator, the system is able to keep up with the increase in the number of devices until 150 devices and started to quickly degrade. By adding a second coordinator to the coordination

![Figure 5.6: Performance of a distributed coordination platform](image)
layer, the problem space of each coordinator is reduced in half, which results in a much better, scalable results. This result verifies that a distributed coordination layer is required to build a coordination platform for our class of fog applications. It also raises a question on how many coordinators do we need? To answer this, we conduct a second set of experiments where we vary the number of coordinators in our system and watch for the second evaluation metric.

As described above, the application’s performance metric is set to be the normalised ratio between all the messages received by the sink components’ instances and all the messages sent by the source components’ instances. We tested with two scenarios in the simulation, which are static coordination where the devices do not move, and fast moving where the devices randomly change their speed and bearing.

We first study the effect of adding more coordinators to the coordination layer. Obviously, to support the increasing number of devices, more and more coordinators should be added to the coordination layer. However, when more coordinators are added to the platform, the number of devices per coordinator is decreased. Up to a certain point, the individual coordinators might not have enough input devices in order to derive the optimal results, which are the combinations of the instances of the involved software components. Recall that each instance of a software component is equivalent to a single device that hosts it.

Figure 5.7 illustrates this interesting findings. As the figure shows, when the number of coordinators increases, the application performance increases as well, but only to a certain point where it starts to degrade even if more coordinators are added.

The figure also shows the performance difference between a static and a dynamic setting. Again, it is evident that when the system is static, the application performance is much higher than when it is dynamic. However, the performance trend stays the same, whether it is static or dynamic. That is the performance increases, peaks and degrades at around the same point as far as the number of coordinators is concerned. This shows that the optimal number of
coordinators is almost orthogonal to the level of system dynamics. This is an interesting and notable observation because based on this fact, the system operator can roughly estimate the optimum number of coordinators required for a particular system without having to worry about the dynamic characteristics of the computing infrastructure.

Secondly, we study how our incremental coordination strategy can support the system’s dynamic nature. Recall that our coordination platform periodically monitors the overall system context and calculates the combinations of software component instances. Whenever a new coordination result is produced, it is broadcast to all the participating devices so that their software component instances can update their peers. Sometimes, this is redundant as the existing combinations still satisfy all the application’s constraints. Whenever this happens, the existing communication links are disconnected, and new links are formed. This leads to messages being dropped during their routes. To overcome this, our coordination platform keeps the previous coordination results and incrementally calculates the subsequent results so that if existing combinations still meet the application’s constraints, they are excluded from the coordinator’s input.

Figure 5.8 shows the positive result from doing incremental coordination. As shown, the number of required coordinators can be reduced roughly from 150 to 50 while still achieve
a performance peak. Thus, only a third of the original number of coordinators is required to achieve the same peak in the application’s performance. Note that in this experiment, it is unnecessary to record the results for the static scenario as the coordination results are never changed. So it is dropped from the experiment.

5.5.4 Reflect on RQ3

Let us revisit our third research question (RQ3), namely, (RQ3) How an application platform for fog can keep up with its large scale and dynamic nature in an efficient way while still maintaining the correctness of the application logic?

While we use a common practice in solving large scale problems (exploiting a distributed cluster of constraint solvers for processing our coordination table), our experiments showed the necessity of doing incremental coordination in maintaining the correctness of application logic in a dynamic fog computing infrastructure. Incremental coordination technique significantly reduces the number of coordinators needed in such a dynamic environment. Eventually, it helps to coordinate our fog applications in a more efficient way.

Our experiments also showed that there is an optimal number of coordinators that helps to achieve the highest system performance in terms of message delivery ratio. More importantly, if we apply a uniform association of devices to coordinator, such number remains the same regardless of the system’s dynamic characteristics. This observation means, one can approximate the number of coordinators needed by estimating the number of participating devices without worrying about the dynamic nature of the system.

5.6 Summary

In this chapter, we present a coordination platform for realising our context-dependent, exogenous coordination model in developing fog applications. The platform fulfils the role of system supports for our application model developed in the previous chapter. We validate our platform by building and running a large scale fog computing study using the network
 simulator Omnet++ where we address the problem of dynamic nature of the fog computing application at large scale.

This simulations triggers some interesting observations, notably the effect of dynamic characteristic on the application performance and its relationship with the number of coordinators employed. That is, more coordinators in the system do not always guarantee better application performance, and the optimum number of coordinators is not affected by the system dynamics. The important advantage is that such a number of coordinators can be provisioned based upon the estimated number of participating devices. It is not affected by the dynamic nature of the system.

Our simulations also show that by doing incremental coordination, the number of optimal coordinator can be reduced by nearly 60%. Importantly, this approach does not depend on which algorithm one implements to solve the large scale coordination problem because the size of the coordination table is significantly reduced from subsequent coordination executions. Since the input space is reduced, the approach works across different implementations. For such a large scale and dynamic fog computing infrastructure, this improvement helps to coordinate our fog applications much more efficiently.
Chapter 6

Literature Survey

In Chapter 3, we developed an analysis of various application models, with which we laid out the requirements toward our class of fog applications. We also pointed out that exogenous coordination is a closely related concept and a suitable approach to build fog applications. This chapter surveys existing application models surrounding this analysis. We first look at general application models found in literature, including ones in distributed system research and more recently, in the WSN, IOT and fog computing areas. We then turn our attention to a large class of coordination-based work that is related to our approach. In each of these groups, we introduce the related work and analyse them according to our requirements.

6.1 Related Application Models

This section surveys related application models based on the five aspects of an application model and the corresponding requirements as set out in Chapter 3, namely:

• Scope: component, inter-component, inter-device and inter-context.

• Application Structure: flat, hierarchical and hybrid.

• Abstraction Level: data querying, data processing, component cooperation, rule-based and goal-based.
• Programming Model: imperative, declarative and hybrid.

• Communication Model: shared memory and message passing with different types of communication cardinality.

6.1.1 Scope

Recall that the scope of an application depends on the developer’s perspective. If a developer is tasked with building a component within a larger application, from his point of view, such component becomes a complete application. Meanwhile, if another developer is tasked with building a larger application that consists of many such components, the engineering of the individual components becomes irrelevant, and his scope of application is broadened to how such components are combined to form a larger application. We defined four levels of application scopes, including component, inter-component, inter-device, and inter-context scopes.

Most of the related work does not exhibit such a clear cut regarding our definition of application scope. Overall, many existing programming models for distributed systems in general, do treat the application as a collection of components, or tasks. However, they blend the development of the components together with the development of the overall application. For example, the common practice of blending communication code within computation code [35] (the component developers have to write the communication part of the component) illustrates the blurry line between the component and inter-component scopes.

Furthermore, while we have seen application models with component scope (e.g. regular programming activities) inter-component scope (e.g. coordination models and languages), and inter-device scope (e.g. distributed database systems), application models with the dependency on the underlying devices’ physical context (i.e. inter-context scope) are still new.

The closest form of application models with inter-context scope, as we found, is a large class of context-aware applications. However, in context-aware applications, their logic
tends to surround the presence of a user and his/her physical context [13]. Thus, the common approach is based on the user’s physical context to execute the application accordingly. When we develop large scale fog application, the notion of users become less important, and we see more of the interactions among components from different devices. The application now has to take the context information of all the participating devices into account when executing its logic.

In recent years, there is another trending group of context-aware applications that utilise large scale pervasive systems [80]. However, the focus is more on the issue of how to, in a scalable way, expose context information to the application for use without requiring the developers to pay attention to the lower level context acquisition tasks (e.g. see Figure 6.1). This is significantly different from our class of applications. While both of these depend on the physical context to some extends, our class of applications uses context information as the metadata to coordinate the component’s activities, as opposed to using the context information as the primary input data in existing large scale context-aware applications. Thus, the task of context acquisition become less critical in our work. Instead, we are more interested in how to use such context to coordinate the interaction among components. In fact, as we discuss later in this section, one of our contributions to support a reusable application model is to clearly separate application data and contextual data.
6.1.2 Application Structure

As pointed out in Chapter 3, we define the application structure aspect of an application model to be flat, hierarchical or hybrid. From our literature survey, the vast majority of application model for fog computing opts for the hierarchical structure. This is understandable when the computing resource distribution in fog computing tends to be hierarchical (e.g. from the cloud to the intermediate fog nodes and to the edge devices). However, as we pointed out in Chapter 3, with regard to the coordination among application components, a flat structure offers a more flexible communication model and is less prone to the single point of failure.

Mobile Fog

Hong et al. proposed Mobile Fog (MF) [39] that allows the deployment of a fog application across multiple devices in a hierarchical system architecture from the network edge to the cloud.

The application is structured following a hierarchical model where application components are distributed across the layers between the fog and the cloud. Programming primitives are provided in order to pass data up and down the hierarchy among the application components. MF relies on a dynamic node discovery process to associate devices together in a parent-child relationship where parent nodes lend their computation resources to process data received from child nodes. That is, data that require more computing resources to process are pushed upward to higher levels (closer to the cloud). Inversely, processed data are pushed down to lower levels (closer to or within the fog).

An excerpt of an MF program is provided in Figure 6.2. To distributed and coordinate the computation across the hierarchy, programming primitives are provided to query for the device capability where the code is executed. Based on this device capability, the code decides to process the data locally, send it upward to be processed at a parent node, or send
it downward to actuate the child node.

MF’s hierarchical system architecture naturally allows applications to aggregate and process data locally along the way from the edge network to the cloud. However, as we described in Chapter 3, hierarchical structure is not suitable for applications with frequent data passing between a large number of components. Thus, communication patterns such as broadcasting or notification that happen among any arbitrary nodes are hard to accomplish.

CloudPath

CloudPath [64] is a multi-tier cloud computing framework that generalises the computing infrastructure across the edge to the cloud.

CloudPath programming model allows developers to express deployment constraints based on execution time. This is done by annotating the executing function with necessary deadline requirements. Based on this, the system chooses the appropriate computing resources across the edge to the cloud to carry out the computation tasks.

The interplay between application components in CloudPath is powered by a hierarchical data store deployed over the geographical span of the network. The data access is made transparent to the developers, therefore hiding the complexity of the heterogeneous and hi-

```plaintext
function on_sense(Type t, Data frame)
    if ¬query_capability(CAP_MOTION_DET) ∨
        detect_motion(frame) then
        send_up(MSG_VIDEO_FRAME, Message(frame))
    end if
end

function on_message(Tag t, Node sender, Message m)
    if t = MSG_VIDEO_FRAME then
        Interest ← ∅
        Vehicles ← detect_vehicles(m.frame)
        for ∀vehic ∈ Vehicles do
            if vehic ∈ Tracking then
                update_position(vehic)
                put_object(vehic, vehic.license_no, m.time, m.location)
        end for
    end if
end
```

Figure 6.2: Hong et al.: Mobile Fog

110
erarchical network environment. Internally components interact by writing and reading to their common ancestor.

Obviously, this is another work that chooses the hierarchical component structure to implement the application logic. The programming model, therefore, only addresses the deployment of components based on the vertical layer across the edge and the cloud without taking into account the horizontal position of each component.

CloudPath could be seen as purely an extension of cloud infrastructure toward the network edge, rather than a solution that focuses on the characteristics of the edge. This is because many distinct features of edge networks such as context-dependent replication of components and their interaction, as well as the dynamic nature of the computing elements are not taken into account.

**FogFlow**

Bin et al. [22] proposed a data flow-oriented application model that allows developers to distribute the computing activities to a large scale fog computing infrastructure. FogFlow follows a distributed, flat application structure where data flows freely between computing devices in a peer-to-peer fashion.

A notable feature is the use of declarative hints that specify how the execution plan should be carried out. For example, a data aggregation task could be declared to be at city level, which constrains it to be deployed once per every city. This can be seen as an important contribution toward exogenous coordination of fog-based applications as the combination of the individual software components is externally coordinated. However, in this work, the coordination mechanism is still simple and based on only deployment scope (e.g. city in the above example). They also do not tailor to the dynamic nature of the system.
Crystal

Jeong et al. [42] presented a distributed application model for fog computing where an application is decomposed into tasks that are sequentially distributed to participating fog nodes. The decomposition is done automatically provided that the application is written in object-oriented programming models. Accordingly, the objects are crystallised into executable units, and remote procedure calls are used for inter-object communication.

The decomposed components are structured into a directed acyclic graph (DAG) and are deployed into participating fog devices. If a child node in the graph fails, the parent node automatically spawns a new child node.

Crystal applications follow a flat structure where components (i.e. crystals) are deployed freely to the fog devices, and their communication happens in a peer-to-peer fashion. However, similar to the work introduced, it does not target inter-context scoped applications. Therefore, no notion of context-dependent application logic is addressed.

6.1.3 Abstraction Level

Recall that the five levels of application abstraction that we defined are data querying, data processing, component cooperation, rule-based and goal-based. In this section, we go through each of these levels and discuss the corresponding related work.

Data Querying Applications

TinyDB - TinyDB [60] is a popular query engine for WSN where the developers can issue a query on top of a WSN and such query is processed collectively by the network as a whole. Having a low level of abstraction, TinyDB allows the developers the full flexibility in expressing the application’s logic. However, this could be too complex for some simple recurrent tasks, and thus, a higher level of abstraction is desired.
Many of the Works based on Imperative Programming Models - [68], [49], [1], [39] all provide an application model based on imperative programming languages. They provide data querying primitives such as to fetch data from the sensors given some criteria (e.g. number of hops).

In these works, the users of the application not only need to master the data querying primitives but also have to program themselves to meet their application. Thus, the notion of end users fades away and is replaced by the notion of expert users.

Data Processing Applications

Mobile Agent-based MapReduce - Ichiro [79] proposed a Mobile Agent-based MapReduce, inspired by the MapReduce data analysis model. Instead of transferring all sensing data to a centralised server for processing using MapReduce, the authors performed MapReduce locally at the edge devices to minimise the network consumption. The system allows developers to write and deploy MapReduce procedures on heterogeneous devices.

This is a typical application model based on data processing abstraction where the developers provide the functions to process the data (mappers and reducers). It also follows the flat, role-based structure application with mapper and reducer components. Since this is a flat structure, a coordinator (i.e. a master node) is required to map tasks onto the participating devices (i.e. the slave nodes) and facilitate the interaction among those (e.g. which reducers receive results from which mappers).

Component Cooperation

Cosmos - Awan et al. designed Cosmos [10], which includes a middleware called mOS and a programming language called mPL for developing WSN applications based on dataflow abstraction. mOS is installed in WSN nodes that run applications written in mPL, a macro programming language. mPL syntax allows effective coordination of network nodes based
on their capabilities (e.g., available sensors, memory) while mOS is capability-aware.

Cosmos applications are modelled as a set of functional components (FCs) which are wired together in a distributed architecture. FCs are described by mPL language with information such as FC id, types of input and output data, relevant node capabilities. Whenever the application is deployed (installed) into the network, individual mOS on each node decides to execute relevant FCs based on the node’s capabilities.

**Flask** - Slightly different to Cosmos, Flask [61] offers a compiler that compiles the designed flow directly into native codes (to nesC programs) that are downloaded into individual sensor nodes without the need of middleware like mOS. Unlike Cosmos, the flow implementation of Flask allows programmatic wiring of nodes instead of static nodes configuration. Moreover, the Flask programming model supports writing of dataflow programs for individual nodes with the help of Flow, their publish/subscribe communication model for composing different dataflow together.

**Abstract Task Graph** - Abstract Task Graph (ATaG) [12] is a data-driven programming models for WSN application. In ATaG, the functionality of an application is modelled as a set of abstract tasks with abstract channels that connect tasks via a common data pool. This data pool is where they can get data from and send data to. Abstract tasks and abstract data are the only code that the programmer provided to the platform. The programmed abstract task graph is then compiled and deployed specifically for different target WSNs. To make the application agnostic to target deployment, ATaG provides some abstract annotations such as neighbourhood-hops:n, neighbourhood-distance:d, parent or children to refer to all nodes within n hops, all nodes within distance d, the parent node or the children nodes respectively.

Although it uses a common data sharing medium to facilitate communication among components, ATaG is another application model with component cooperation-based abstraction level. The reason is the cooperation among components is exposed to the developers so
that they can specify how the components interact with one another. The shared data medium becomes an implementation technique that does not affect the abstraction of the applications.

**Exogenous Coordination-based Models** - Coordination models and languages [71] themselves advocate for a clear distinction between computation and communication activities, which naturally follows the component cooperation abstraction level. This separation of concern is pushed further with exogenous coordination, where computation activities are encapsulated in independently developed components, and the users/developers describe the cooperation among those. An external coordinator is employed to coordinate the participating entities with regard to the users’ requirements.

Since this is the closest approach to our requirements of building a fog computing application platform, we dedicate a separate section in the latter half of this chapter to discuss in detail about related coordination models and languages.

**Rule-based**

*If This Then That* [91] is a popular application platform for combining Internet-based services. It allows users to express simple rules such as *if [insert service condition here] then [insert service invocation here]*.

Slightly less flexible than the Component Cooperation abstraction level, Rule-based application model targets less experienced users who have little experience about the programming skill. For many user-oriented tasks, this is easily programmable and yet, is still user-friendly enough. However, for large scale and complex application, its expressiveness is limited.

**Goal-based**

Goal-based application model offers the least flexibility in expressing the application’s requirement. However, it is the easiest way that a user can interact with the application to
describe his needs. Thus, this is the other extreme of the flexibility and expressiveness versus ease of use scale.

**RESTdesc** - RESTdesc [50] is a machine-readable description language for describing the semantic of RESTful Web Services. It has annotations to specify the current state of a RESTful Web Service and necessary requests to get the Web Service to transition from this state to the other. In this work, computing resources are modelled as RESTful Web Services, and RESTdesc is used to describe their semantic.

An application is developed without the prior knowledge of available service APIs. Instead, it relies on the RESTdesc description language to reasoning about the available services and makes necessary decisions. A semantic reasoner is used to help with the reasoning process and allows IoT applications to make decisions.

This work incorporates a coordination model behind the scene that coordinates the user’s things so that optimal settings are in place. The coordination is done by reasoning over the RESTdesc-enabled things to find an execution path that consists of RESTful Web Service requests to be made to the things.

### 6.1.4 Programming Models

**Imperative Programming**

**PyoT** - Azzar et al. [11] proposed PyoT, an application model based on the Python language for multi WSNs architecture. In PyoT, device nodes run a CoAP server and expose their services via CoAP protocol. An application is modelled as a set of tasks that coordinate available devices’ resources. A PyoT system contains several WSNs, each of which is administered by a PyoT Worker Node (PWN) who monitors the availability of devices and executes tasks deployed by developers. There is a Storage Element that indexes all available device resources in a PyoT deployment. PyoT can combine with T-RES, a middleware that
allows executing tasks on individual nodes rather than on PWNs.

In Pyot, the author follows an imperative programming model for building IoT applications. Imperative programming model tends to express the application behaviour through explicit statements that indicate how to change the program states [65]. This indicates that an imperative programming model alone is not suitable for building large scale IoT systems due to the high level of detail a developer has to deal with when processing such large number of simultaneous events.

Actinium - Kovatsch et al. [49] developed Actinium, a RESTful runtime container for writing IoT applications. Actinium accepts application scripts from the developers and executes them in a similar way to traditional web servlets. Developers dictate the application logic by imperatively instruct the runtime container where to get data from, how to do the processing and where to send the data.

Due to the centralised execution model, such scripting environment is suitable for small scale settings such as in smart home or smart building applications. It is also difficult to express large scale application logic using the same scripting technique as it involves the coordination of many components whose deployed instances are unknown at development time. This is a typical example of how imperative language is not suitable for expressing the uncertainty of large scale and dynamic applications.

SenseLet - SenseLet mimics the way Web Servlet technology works and provides a similar Java-based technology called SenseLet that runs on embedded devices. Each device is installed with the SenseLet middleware that can execute SenseLet code. Each SenseLet code is a server-like application that responses to external requests, which is similar to Web Servlets. SenseLets can be programmed over the air to be installed to the target devices, facilitating IoT application development.

SenseLet is a typical example of how imperative programming models are suitable for
node-level programming.

*Patricia* - Patricia is a programming platform that aims at developing IoT applications. It models IoT applications as Intents and Scopes, which are expressed by the developer following an imperative programming style. Scopes are used to query or find relevant things while Intents are used to execute application logic with the query results. The applications are developed and deployed on a multi-layered Cloud infrastructure consisting Development Support layer, Cloud Runtime layer, Data and Device layer. Patricia programming model is exposed to developers on the Development Support layer where developers write IoT applications using Intents and Scopes. Domain experts who have deep understandings about physical things participate in the Cloud Runtime layer to develop domain-specific libraries for connecting the Patricia programming model with physical devices. Data and Device layer monitor and administer available devices connected to the system.

Similar to Actinium, Patricia provides a centralised way to develop IoT applications that involve a fleet of devices. Due to the centralised model as well as the imperative nature of the programming model, it is difficult to extend Patricia to support a class of large scale fog applications.

**Declarative Programming**

*D-Lite* - Cherrier et al. [24] proposed D-Lite system and its corresponding SALT programming language for building IoT applications. D-Lite is a distributed middleware implemented in Contiki OS for motes based on Finite State Transducers. An IoT application
Figure 6.4: Bischoff et al.: A Compiler for the Smart Space

is modelled as a set of rules that decide the behaviour of the underlying mote based on the messages it receives and its current state.

The application model is presented in Figure 6.4. Despite using a declarative programming model (rule-based), the target of the work is not suitable for large scale applications due to the lack of generality within the application’s rule. Furthermore, with a large number of simultaneous events, this programming model easily become unmanageable.

A Compiler for the Smart Space - Bischoff et al. [15] developed another state machine-based application model for expressing smart space applications, such as smart homes, smart offices. Similar to D-Lite, an inherent limitation of this approach is the explosion of state space encountered when applying the model to large scale applications that consist of thousands of entities.

Exogenous Coordination-based approaches - Exogenous coordination [7] is another approach in describing the application logic. This is done through defining the dependencies among application’s components and associated application-specific requirements.

Unlike many WSN or IoT application model, coordination-based approaches is purely a
software engineering process. Except for some recent research into coordinating mobile ad-hoc network [58], the notion of hardware in coordination-based approaches stays minimal.

However, since coordination-based approaches have a clear separation of concerns between computation and communication activities, it makes the application model itself reusable. This is an essential feature that we advocate that it would promise to solve the complexity of large scale fog applications. We dedicate a separate section at the latter end of this chapter, to discuss in detail the coordination-based approach.

**Hybrid**

A hybrid programming model as defined in [65] is a combination of imperative and declarative programming where the imperative model is used to develop the software component, and the declarative model is used to develop the interaction among components. This aligns with our vision of a coordination-based approach for complex fog applications. However, the authors did not focus on the clear distinction between computation versus communication. They rather described a programming model that consists of both worlds to be hybrid. If we apply our concept of application scope, this definition becomes irrelevant because it does not clearly distinguish component scope developers from inter-component scope developers.

As discussed in Chapter 3, we rather define a hybrid programming model to be either imperative or declarative, but with the use of annotations to express orthogonal concerns. An example is the use of annotations or decorations in popular programming languages such as Java or Python, where Aspect Oriented Programming [30] is a concrete example.

The use of annotations expresses cross-cutting concerns that affect multiple components of an application and are orthogonal to the application logic. Essentially, in such programming model, the annotations declare how should the same application logic should behave under various running conditions.

While annotation-based hybrid programming models have been seen in [64], [22], their usage tends to be ad-hoc, and there was no formal definition of them or how they are de-
signed. Here we aim at abstracting the annotations’ functionality to be the coordination primitives that dictate under which context such a component should be executed or interaction among multiple components should be triggered. Thus, our programming model can be seen as a superset of those work.

6.1.5 Communication Models

Message Passing

Service Oriented Architecture-Based Service Oriented Architecture has been recognised as one of the most prominent approaches in providing interoperability between components [9] [90]. In SOA, applications consists of services that interact with one another via standardised interfaces [94] [73] [29] [2]. Thus, services here become the components of the applications. There are several levels of component composition ranging from static to dynamic, orchestrated or choreographed [27] [52]. Data are shared following a message passing style in the form of service invocations.

Shared Medium

Tuple Space Based TS-Mid [56], LIME [72] and TeenyLIME [26] are efforts to extend the tuple space model appeared in [33] to support distributed tuple space, tuples observance and mobility of sensor network. Unlike LIME, sensor nodes in TeenyLIME only share their tuple space with their one-hop neighbourhoods instead of with the whole network. This design choice is due to the spatial locality of WSNs in which related sensors and actuators are closely located (fire extinguishers and smoke detectors). TS-Mid opted for a hierarchical system of tuple spaces that splits the network into different clusters with cluster head being responsible for computation tasks.
ActiveStream  ActiveStream (AS) [63] is a platform for cross-domain Internet of Things applications. It features a shared data medium that is similar to a Publish-Subscribe message broker. In AS, components publish their activities to the common channel in the form of “Alice added a picture to her album” and other components can subscribe to be notified when such activities happen. In this way, AS become a shared data medium that allows different components to integrate with each other by publishing and listening to valuable activities without having to know the detail implementation of one another.

6.1.6 Analysis

A summary of all the surveyed research bodies is listed in Table 6.1

With regard to application structure aspect, although flat structure remains the most popular among many application models in distributed systems, WSN or the IoT, hierarchical structures seem to be a prominent architectural model in fog computing [38] as it supports the data filtering and aggregation operations. However, we observe that it has a drawback in facilitating inter-component, inter-device, or inter-context communication across different branches within the hierarchy. With our class of application, its context-dependent replication of application components requires a certain level of flexibility for the inter-context communication (e.g. our target class of applications is characterised by its necessity to pass data arbitrarily from one device to one another). Therefore, we argue that a flat architecture with bounded communication cardinality is more suitable.

With regard to application scope aspect, at different scopes, there is a different set of problems for the application development process.

At component scope, conventional programming practices apply. That is, issues such as programming languages (e.g. C, Java), programming models (e.g. functional, object-oriented or aspect-oriented), or algorithms/optimisation techniques play a central role in the development process. Accordingly, development supports come from debugging tools, integrated development environments that tailor to the programmer’s need.
<table>
<thead>
<tr>
<th>Researchers</th>
<th>Code Name</th>
<th>Application Scope</th>
<th>Application Structure</th>
<th>Abstraction Level</th>
<th>Programming Model</th>
<th>Communication Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hong et al.</td>
<td>Mobile Fog</td>
<td>Inter-Device</td>
<td>Hierarchical</td>
<td>Data-Querying</td>
<td>Imperative</td>
<td>Point-to-Point (parent-child)</td>
</tr>
<tr>
<td>Sehic et al.</td>
<td>CloudPath</td>
<td>Inter-Device</td>
<td>Hierarchical</td>
<td>Data-querying</td>
<td>Imperative</td>
<td>Shared Medium</td>
</tr>
<tr>
<td>Bin et al.</td>
<td>FogFlow</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Jeong et al.</td>
<td>Crystal</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Madden et al.</td>
<td>TinyDB</td>
<td>Component</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Ichiro</td>
<td>Map-Reduce</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Awan et al.</td>
<td>Cosmos</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Mainland et al.</td>
<td>Flask</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Bakshi et al.</td>
<td>AToG</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Rule</td>
<td>Imperative</td>
<td>Point-to-Point</td>
</tr>
<tr>
<td></td>
<td>IFTTT</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Mayer et al.</td>
<td>RESTdesc</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Azzar et al.</td>
<td>PyoT</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Kovatsch et al.</td>
<td>Actinium</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Akrivopoulos et al.</td>
<td>SenseLet</td>
<td>Component</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td>Nastic et al.</td>
<td>Patricia</td>
<td>Inter-Context</td>
<td>Flat-&gt;Role-based</td>
<td>Data-Querying</td>
<td>Imperative</td>
<td>Point-to-Point</td>
</tr>
<tr>
<td>Cherrier et al.</td>
<td>D-Lite</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Rule</td>
<td>Declarative</td>
<td>Point-to-Point</td>
</tr>
<tr>
<td>Bischoff et al.</td>
<td>SOA</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Rule</td>
<td>Declarative</td>
<td>Point-to-Point</td>
</tr>
<tr>
<td>Mayer et al.</td>
<td>ActiveSpace</td>
<td>Inter-Device</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Unbounded Cardinality</td>
</tr>
<tr>
<td>Giang et al.</td>
<td>Distributed Node-RED</td>
<td>Inter-Context</td>
<td>Flat-&gt;Role-based</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Flat-&gt;Context-Dependent Roles</td>
<td>Component-Cooperation</td>
<td>Hybrid-&gt;Annotations</td>
<td>Bounded Cardinality</td>
</tr>
</tbody>
</table>
At a larger scope, inter-component scope, interaction among components become the new central issues of the development process. This is where the separation of concerns between computation and communication became a topic of research from which a plethora of coordination models and languages [71] are designed to ease the design of inter-component interactions. When the computing power gets cheaper, more and more physical devices are equipped with computing resources. The computing paradigm is transiting toward the smart embedded devices rather than staying at the resourceful computers.

We start to see a larger application scope that involves not only the software components but also the underlying devices and their physical context. Thus we say that another class of application models that are dependent on the context of the underlying computing devices needs to be proposed. That is, at this level of application scope, we need context-based coordination that takes into account the underlying computing devices in addition to the application’s components themselves. From the summarised table, we found that this has not been studied in the literature. While some studies address some characteristics of the underlying computing devices, a clear notion of inter-context application scope is still missing.

About the communication model aspect, it is generally accepted that message passing communication model provides a clearer control flow that represents the interaction among components, thus leads to better coordination activity. Complex applications with larger scopes are therefore more likely to benefit from message passing communication models.

At the same time, the application models that have inter-context scope have bounded communication cardinality model as well. This is the result of the mass deployment of many embedded devices over a large area. Since each device carries its own set of components, this large scale deployment results in a high level of replication among components whose physical context is different.

Unlike computers running in data centres where the replications of components are iden-
tical and are used to improve the availability or scalability of the system, replication of components in large scale applications are not identical. This is due to the context of the underlying devices that makes each replica unique on its own despite the component code, or even its data are the same. For example, although traffic situation data produced by a software component are the same, such data when delivered to the cars in front of their source (the cars which generate data) versus when delivered to the cars behind their source have different effects on the drivers’ decision.

Thus, we have bounded communication cardinality where a bound number of replications of one component interacts with another bound number of replications of another component. From the summarised table, we see that bounded communication cardinality is not the focus of most work. This is understandable since only a few application models are developed with the inter-context scope.

Based on these analyses, we see that it is not straightforward to apply application models of one system scale to a different one. Meanwhile, none of the surveyed large scale systems meets our requirements for fog applications. This is because most of the works are based on cloud computing, which is a centralised computing model. While other works target distributed applications with heterogeneous devices, they focus on a small scale of deployment where the context-dependent replication of components is minimum.

Another important aspect of application models for fog computing that we focus on is how it supports the dynamic nature of the system. This includes how the application obtains and makes use of the physical context surrounding its execution environment. Among the surveyed application models, none of the work incorporates the dynamic nature of the devices. Even though the work done by Bin et al. [22] appears to be an exact match in comparison to our proposal, they do not focus on supporting the dynamic nature of the edge network. A common reason is that these works treat these dynamic devices as the end user devices, which do not belong to the fog computing infrastructure. According to Bonomi et
al. [17], end-user devices are also a part of fog computing architecture; thus an application model for fog computing should take into the dynamic nature of these devices into account.

### 6.2 Coordination Models and Languages

Coordination-based approaches are the closest to our target application model as they provide a clear distinction between computing and communication activities, consequently. They allow to express the cooperation of components while having some other desired features such as flat structure and declarative language. This section is devoted to related coordination models and languages that are comparable to our target class of applications.

Generally, the main limitation of existing coordination models and languages is the lack of support for large scale application scenarios, which includes concerns about inter-context application scope and the context-dependent replication of components and their interaction. Also, the dynamic characteristics of the systems are bound to the fluctuations in the availability of computation activities (software components).

Many existing coordination models and languages only support this level of dynamic nature without taking into consideration the constant changes in the physical context of the computation activities. These works also mainly focus on coordinating different communication patterns among the computation activities rather than coordinating the dynamic nature of the system. For example, although the coordination models and languages proposed in [55] [81] [14] do have the primitives to identify on which processor, devices should a computation activity run based on their hardware capabilities, they are static properties rather than dynamic.

While context-dependent connectors have received much attention over the past ten years [44] [18] [25]. However, the notion of context referred to in these works is limited to the pending activities on each of the connector’s ports. The goal of the coordination model is then to react to the presence of components on each port and to propagate this event further
into the connector’s composing channels.

Song et al. [87] proposed spatial and temporal selector primitives for the Actor, Role, Coordinator coordination model [78]. While this work taps into the physical context of the devices, it did not target the large scale application scenario. As a result, many requirements that are specific to large scale and dynamic fog applications are not addressed.

In [58], the authors addressed the problem of coordinating mobile distributed systems that have frequently changing network topology. While this is the closest work to our research as far as we are aware, it only addresses the intermittent connectivity of participating devices without taking into account their actual movement and changes in context. Moreover, it focused on a small scale mobile distributed systems with local network broadcasting used as the coordination medium among participating components.

### 6.3 Summary

In this chapter, we showed that what is limited among various application models for distributed systems and coordination-oriented research is their lack of interest in the underlying devices where components run and their physical context.

Figure 6.5 illustrates the proposition of this thesis in compared to the state of the art in programming and coordinating distributed systems.

From early application development in distributed systems, there is the notion of module interconnection language or MIL [28] that expresses the interconnection of software components within a large software system. This is in contrast to module-level programming where individual computation activity is the main focus of the developers (component-scope according to our analysis in Chapter 3). The components were then abstracted into well-encapsulated processes that exchange information with one another regardless of their internal implementation languages. This is where coordination models and languages show their importance in coordinating such heterogeneous inter-process cooperation.
Before the research into Wireless Sensor Networks gained much traction, there was little or no notion of underlying devices that run a computation activity. The focus was to coordinate the communication among software components or the processes due to the small scale of the hardware deployment. In WSNs and with the advent of embedded computing systems that are more prevalent and widely distributed, the programming models and languages start to incorporate the notion of underlying devices that host the computation activities. Thus, many programming models in WSNs were designed to support a variety of hardware or device heterogeneity.

WSN can be seen as a large scale distributed system with the distribution of computing elements over a wide geographic area. However, the computation tasks in WSN are single purpose and predefined, due to the resource constraints of the sensing nodes. For example, some networks are designed specifically to monitor forest fire, capture structural deficiency, or to monitor the air quality. As a result, many WSN programming models only focus on data collection and dissemination aspect of the applications; while some others provide

---

**Figure 6.5:** Context-dependent Exogenous Coordination in Fog Computing
simple aggregation primitives. A full-fledged application framework with heterogeneous computation tasks is therefore, has yet to be seen in WSN.

In 2012, Bonomi et al. [17] proposed fog computing, a new computing architecture that involves computing resources across the network edge to the cloud to support a range of applications that are delay sensitive or data intensive. In one way, fog computing is similar to WSN in that it is also a large scale distributed system. However, computation activities in fog computing are more generic than in WSNs scenario. This makes it more important to study an application model for the fog computing infrastructure. At the same time, we have seen throughout this thesis, computation activities in fog computing are deployed in very dynamic infrastructure.

In this context, the physical context (e.g. device, location, time) of the devices which run the computation activities becomes an integral part of the application. This, along with the need for a high-level application model for large scale deployment of fog computing, become the core contribution of this thesis.
Chapter 7

Conclusion

7.1 Summary of Contributions

Within our target class of fog applications that involve a large number of highly dynamic, geographically distributed devices, we made the following contributions.

First, through an analysis of various application models ranging from general distributed applications to the recent classes of WSN, IoT and fog applications, we showed that the development of fog applications benefit significantly from an exogenous coordination approach, where a clear separation of communication and computation aspects is advocated. Our developed taxonomy not only serves as a guideline for our research but also present new set of trade-offs that promise to help fog computing developers to make the appropriate design choice that is suitable for their applications.

Second, since existing exogenous coordination models and languages do not focus specifically for large scale and dynamic scenario, we develop a set of coordination primitives to make the exogenous coordination concept more applicable to the development of our fog applications. Notably, we proposed the notion of separation of concern between contextual and application data. This paves the way for the development of our context-dependent primitives that help describe many common scenarios in our class of fog applications. We
also brought up the importance of context-bounded communication cardinality as a unique communication pattern that we found, is prevalent in our class of fog applications.

Third, to realise our context-dependent exogenous coordination model in the large scale fog computing infrastructure, we simulate a distributed coordination platform. Our notable proposal is an incremental coordination technique that remembers the past coordination activities to execute the subsequent ones. This technique largely reduces the input space to the large scale coordination problem, which eventually reduces the overall resource consumption. Since the input space is reduced, the technique is applicable regardless of the choice of algorithms for solving the large scale coordination problem. Furthermore, our simulation showed that the dynamic nature of the system does not affect the coordination activities with regard to the overall application performance. This finding allows us to provision the number of coordinators needed for a particular fog computing setting based on the estimated number of participants rather than having to care about the dynamic nature of the system.

7.2 Limitations and Future

There are also a number of items within our current thesis that can be extended in future work.

1) Currently, our application model does not adequately address the configuration side of the software components. Each independent application comes with its own set of configurations that dictates the application specific behaviour under certain circumstances. This configuration aspect has not been studied in this work. We envision that the coordination platform has to coordinate not only the cooperation of the subcomponents but also their configuration in a dynamic way that is suitable for the run-time condition. For example, in our lab-based setup, the camera component comes with a set of configurations to adjust its execution, such as the captured frame rate or resolution. This configuration data, while pertaining to the component-level application, also affects the composition of large scale ap-
application. Thus, the application platform should allow the developers to also dynamically coordinate the configuration of individual subcomponents based on the underlying hosting devices. One possible solution is to let developers define a set of configurations for certain subcomponents so that at run-time, the system can selectively choose the right configuration for their execution.

2) We did not address a range of mission-critical applications in our solution. This is not because that mission-critical applications are not important in fog computing settings. In fact, we believe the computing model provided by fog computing can also facilitate a large number of critical applications. In this thesis, however, our interest is more focused on delivering solutions for a range of general fog applications that are large scale and dynamic. The applications are also data intensive that demand a cheaper solution in comparison to the traditional cloud computing scenario.

Admittedly, this is only one half of the application classes that fog computing is proposed for, which are data intensive and delay sensitive applications. However, we believe by relying on techniques such as priority-based scheduling and feasibility analysis in real-time systems, we can introduce priority-based coordination primitives to extend the current work in supporting this class of mission critical applications.

To support priority-based scheduling, our coordination model needs to support the execution of partial combinations of software components. This is to help devices to prioritise which software component should they contribute. For example, if an application graph includes a logging component, it can be given a low priority so that, when resources are limited, its execution can be skipped in favour of higher priority components. We can also add priority-based primitives on top of our context-dependent primitives, such that, based on the dynamic nature of the context data, the priority of certain instances of software component, or the links among them changes accordingly. For example, if a certain device is contributing a video processing component and a temperature sensing component, depending on the de-
vice’s context, such as available network bandwidth and computing capability, the developer can choose to prioritise the device’s contribution of these two components. He can express a requirement such as, the priority of the video processing component should depend on the available network bandwidth of the device. By doing this, when network bandwidth is more available, the device chooses to contribute (execute) the video component instead of the temperature sensing component, and vice versa.

To support feasibility analysis, the software components need to be annotated with real-time constraints, such as computation time required. The participating devices will then use this information to further calculate their contributions based on their live context and the new constraints. Accordingly, QoS-based metrics such as deadline distances [86] can be used to evaluate the results.

For applications that have requirements on the communication delay, models and tools from Software-Defined Network (SDN) can be used. For example, if the application requires a specific communication delay among a pair of software components, whether or not they are directly connected, SDN is a suitable approach. That is, by relying on smart routers that are capable of routing the packages based on application-level requirements, we distribute our coordinators further down to the network edge. Since these routers have explicit knowledge about the communication links within the network, we can coordinate the communication among software components based on lower level Quality of Service (QOS) requirements such as network bandwidth or throughput. For example, based on the data transfer among the components, SDN can prioritise certain links over the others. This is helpful when we need to react to unexpected events such as when a vehicle crash occurs, the communication links from the nearby camera are prioritised over other locations. From the developers’ point of view, this is similar to priority-based coordination primitives we proposed above, except that it works on the links rather the components. We can also use primitive connectors [6] that have special properties, such as delivery semantics (e.g. exactly once, at most once), or...
synchronisation semantics, to express a broader range of communication requirements.

3) We did not study the complexity of our coordination primitives. While our coordination primitives are simple, they are very flexible and powerful in allowing developers to express their application requirements. Since it is very easy to express the application logic, our system supports might fail, or take an unacceptable amount of time to guarantee the runtime execution of the application. Thus, one important topic is to formalise and study the complexity of the coordination primitives. Ultimately we should be able to verify whether each application submitted by the developers can be easily coordinated or not.

Since the study into these subjects can quickly grow out of the scope of this thesis, we leave these extensions for our future work.

Besides, we also recognised several weaknesses, notably the ability to carry out large scale, real-world experiments. That is, we did not have access to a real-world testbed to evaluate our coordination primitives fully. Although our developed coordination platform Distributed Node-RED has been field trial in Fujisawa, Japan, this remains an academic experiment with a lack of large scale city infrastructure. Thus, some of our coordination primitives were not evaluated fully.

Second, while our work addresses issues in the design and development of large scale applications, there are topics that are outwith the scope of our work. One of these issues is the physical communication layer that interconnects devices and software components together. In this thesis, one simplifying assumption we make is that the communication layer is provided so that inter-device connection can be established between any pair of devices efficiently and at low cost. With the advent of new communication technologies such as 5G, we believe this is soon to be within reach for fog applications.
7.3 Other Directions

Here, we only addressed the development phase of the fog applications. In practice, more phases need to be considered. An important and common requirement is the ability to debug the applications before deploying them to the large scale infrastructure.

Thus, a static analysis of the coordination primitives is expected to be important. It allows the developers to simulate the practical running environment so that requirements that are not logical or conflicting primitives can be avoided at the development time.

Although this thesis proposes a participatory coordination model, it does not address a range of issues surrounding this model, such as cost, incentive or trust model. While we argued earlier that we could learn much from the crowd-sourcing research community, we believe the unique characteristics of fog computing infrastructure is going to pose interesting research questions in this area.

This direction is also highly related to realising a multi-tenant fog computing platform where multiple players and stakeholders participate in contributing, sharing and managing the resources. Resource conflict resolution is a natural consequence of such a requirement, with which we also expect to see many other research questions.

To fully support the practical deployment of fog applications, it is also necessary to go down the application stack to solve lower level technical problems. For example, to coordinate various communication constraints such as message delivery semantic, or interplay between multiple communication modes (Wi-Fi, LTE,...) concerning the context of the devices. A possible direction is to integrate or push the coordination platform down to the network core, following the software-defined network approach. This can be expected to improve the coordination overhead further.
Bibliography


Appendices
Appendix A

Fujisawa Field Trials

The open source implementation of our coordination primitives, the Distributed Node-RED, has been used to build a field trial in Fujisawa, Japan. This is under the collaborative research project BigClout, where we worked with a team of researchers at the Keio University (Fujisawa, Japan) to evaluate our coordination platform in a real world environment. This appendix re-creates the discussion here for the sake of completeness, the full text can be found in [47].

A.1 Background

Recently, the advancements in Machine Learning (ML) and AI have been helping cities to become smarter. For example, there have been projects that leverage these advancements to autonomously detect road damage by looking at video streams from dashcam mounted on cars [45]. In these projects, city data are gathered and sent to centralised cloud servers where AI software is deployed to analyse the data and produce the detection results.

This centralised approach is well suited to ML algorithms, but unfortunately does not reflect the reality of Smart Cities. Typically, cities have a multitude of sensors and “things” (e.g. automobiles, mobile devices and robots) that are distributed throughout the city and communicate with each other via the Internet (e.g. Wi-Fi, LTE, 3G or Ethernet). This results
in city data exhibiting a wide variation in time and space both in short term, i.e. daily cycles, and in longer term cycles as the city, its infrastructure and its citizens evolve. In short, the city is a spatial-temporal distributed edge environment (STDEE) and requires us to adapt our ML techniques to this environment. However, developing and operating ML applications in a STDEE is not straightforward. This is primarily because the development process, which is already complicated because of the need for multiple trials to aid learning, needs to also accommodate the distributed nature of the STDEE. Obviously this leads to significant development costs, both in terms of time and resources, and is by nature inherently complex.

To ease both the cost and complexity, an integrated development environment is desired for ML application development suitable for STDEEs as found in smart cities. To solve this problem, we have designed and implemented CityFlow, which supports the development of ML applications in the STDEE city. CityFlow makes it possible to easily describe the flow of data which comes from devices in the city, and to deploy trained ML models to a variety of devices distributed throughout the city. By making these processes easy, we can conduct data pre-processing and preservation quickly and repeat proof of concept (PoC) that verify the performance of the ML model. CityFlow is built using Distributed Node-RED (DNR).

A.2 Challenges

A.2.1 Urban Data Collection and Preprocessing

In statistical machine learning, it is generally assumed that the data have already been collected, formatted and normalised to use for the training datasets - often using offline processing and assuming homogeneous datasets. In contrast, cities exhibit highly heterogeneous datasets, which change over time as the city changes and where data is streamed in real-time.

This results in increased processing and storage and requires a dynamic learning model that retracts as data changes. While there exist a number of algorithms which are able to handle streaming data, such as CQL [3], Apache Spark [98] and Apache Storm [93], they of-
ten assume cloud based servers with load balancing and scaling, and work on homogeneous datasets. Which, as discussed, is not often the case with real world city datasets.

In addition, we usually conduct some pre-(post-)processing of the data. For example, for privacy protection, since the data from edge devices is often bound to the real world, there is a risk of privacy invasion. For example, camera data from city sensors, or car data captured in real time has significant privacy issues. However, the diversity of devices makes it hard to cope with the data processing required for dealing with this privacy invasion.

These requirements lead us to the adoption of fog computing paradigm where data can be processed in place and in a distributed manner across the city area. Therefore, we need a mechanism to partition our machine learning applications into separate components, a way to disseminate them to the fog infrastructure and a way to coordinate their running instances and their communication.

A.2.2 The Machine Learning Model Execution Environment

After the PoC phase, it is necessary to distribute the trained ML model to all (or a subset) of the edge devices in the city. In typical ML applications, high-performance cloud computers are utilised. This is often necessary because the ML model has a large number of parameters requiring significant memory and high-performance CPUs. However, it is often too expensive for cities to own and manage such servers due to budget constraints.

In contrast, we can use edge devices in a STDEE city that typically have significantly lower specifications than cloud based servers. However, while we can use traditional approaches on edge devices with reasonable computational resources, the latest approaches which have good performance, such as deep neural networks, are not always appropriate due to their resource needs. In order to benefit from their performance, we need to divide the ML model into subsets of small ML models or processes that can be distributed across a set of edge devices [41] [53] or compress it to load it on the edge device memory [40]. This is another aspect, where we need a way to partition and distribute our ML applications in a
large scale smart city environment. Essentially, this is where component-based abstraction model demonstrates its benefit.

Additionally, since the edge devices are highly heterogeneous, it is not just a simple matter of distributing partial modes to a set of edge devices, rather the constraints and context of each device needs to be considered as part of any distribution algorithm. That is, we need to cater specifically to the context of the edge devices and processes that we are working with.

A.2.3 Covariate Shift of the City Data

In statistical machine learning, typically it is required that there is no difference between the marginal probability distribution of the training dataset and that of the test dataset. Namely, assuming the distribution of training dataset is $p$ and that of test dataset is $p'$, $p(x) = p'(x)$ is required.

In an STDEE based city, the network status and the context of the location, where the edge devices are installed, changes. Therefore, the marginal distribution becomes different, although the relationship between the data $x$ and the desired output $y$, that is posterior $p(y|x)$ is consistent. This phenomenon is known as covariate shift [83, 89]: $p(x) \neq p'(x), p(y|x) = p'(y|x)$.

When considering smart cities, we use the hypothesis that the covariate shift exists in the city, as the distribution of the data from the city often varies significantly in spatial and temporal domains. We call the covariate shift caused in the city as Spatial-Temporal covariate shift. In a broader perspective, covariate shift of ML applications in large scale smart city environment is highly context-dependent.

The covariate shift of the city data is illustrated in [A.1]. The prediction accuracy of the model is high in the shopping street, because the model is trained with the data collected in the shopping street. In contrast, in other areas away from the shopping centre, the more the data distribution differs from that of the shopping street, the lower the prediction accuracy is.
Figure A.1: Influence on accuracy due to a spatial-temporal covariate shift. For models taught from data around commercial facilities, prediction accuracy is high in similar commercial facilities. The prediction accuracy decreases as the spatial features differ.

Figure A.2: Road damage detection application developed by CityFlow. Owing to the ST covariate shift.

For instance, supposing car detection based on video image analysis is required, a model which is trained with the video of shopping streets can detect cars with high accuracy in similar streets. By contrast, at the sea coast, the model struggles to precisely detect cars because the background is sea water instead of buildings. Similarly, if the model is trained with the videos taken in daytime, it is difficult for it to detect cars during the night, even though the location is the same. Therefore, it is important to select appropriate domain specific adaptation for the ML application based on the physical context of the data source.
A.3 The use of Distributed Node-RED

In this project, we use a road damage detection application as a first case study, with a focus on road line markings. Although road damage is a common problem, in many cities, inspection is still conducted by sight. This manual visual inspection is resource intensive and expensive. Therefore, we have to explore ways to inspect the city infrastructures, especially roads, at low cost.

Recent work by Kawano et al. [46] proposed a method for road inspection using recorded images from cameras mounted on garbage trucks. However their approach relied on central cloud processing which is inefficient and costly in terms of processing and communications and fails to handle local privacy issues (the first challenge). Simultaneously, since they adopted an object detection approach [76] which uses a large neural network, it is difficult to work on edge devices (the second challenge). Moreover, the cityscapes of images changes in some areas, so that the ST covariate shift might occur between those areas (the third challenge). Our application to address these problems, implemented using CityFlow, operates as follows A.2:

A.3.1 Detecting Road Damages

In order to deploy the networks to edge devices and cope with ST covariate shift, the neural network is separated into feature extractor node and damage detector node and deployed to different devices on the truck, respectively. These application partitioning and deployment are done thanks to the component-based programming model offered by DNR exogenous coordination platform.

Consequently, we can adapt domain adaptation approach [32]. Then, when the damaged road is detected through these networks nodes, the location of it is published to SOXFire via SOX Node (sox-out in the flow).
A.3.2 Anonymising Sensing Data

When multiple garbage trucks confirm an area of road damage, one truck is designated to upload a partial video of the area. A SOX-in node receives the location information, and the next node compares the location of the truck with it. If it is true, the driving recorder mounted on the truck sends the videos after anonymising to the cloud visualisation application. Before uploading, information related to the person such as faces or car matriculation plates is removed if the video contains them. Using DNR, it is easy to specify where private data should be anonymised (GANonymizer at garbage truck sensors) before sending up to the cloud.

Overall, DNR helps to solve the challenges raised in the previous section by providing a context-dependent coordination model that is designed for large scale, distributed computing environment. It helps distribute ML components across the devices in a smart city scenario, which is an essential requirement for smart city ML applications. Moreover, anonymising sensing data becomes an easy task thanks to the context-dependent primitives that specify from where (physically) the data should be anonymised. This is also another important requirement for smart city applications. DNR also provides a high level programming abstraction that helps developers to work with the complexity of the smart city environment without being too technical-savvy.

Having said that, the field trial does have several limitations, with which we were not able to fully evaluate the capability of DNR. We discuss these limitations in the following section, along with the future work items.

A.4 Things to Do

CitiFlow, which is powered by Distributed Node-RED, is a proof of concept implementation that show how complex ML applications can be realised in a geographically distributed computing environment - Fujisawa city. However, to achieve a fully productive fog computing
infrastructure with application support, there are additional work to be done.

First, although we worked with a large number of garbage trucks, which represent an important part of a fog computing infrastructure, we still lack additional fog computing elements, such as road-side units. Thus our setup represents only two layers of the whole system, the edge network and the cloud. To fully embrace fog computing paradigm, more intermediate computing resources between the edge and the cloud are necessary. This is one of the limitations of the project in evaluating the fog computing model and our proposed coordination model.

Second, since there is only two layers in the experiments computing infrastructure, the inter-component context-dependent primitive and context-bound communication cardinality were not evaluated. That is, due to the lack of intermediate, edge computing infrastructure, such as road side units or mini data centres located in cell towers, there is no actual requirement to constrain the communication among some software components based on their physical context. While we can certainly apply inter-component, context-dependent primitives such as \texttt{nearby} to software components running on the trucks, we do not have sizeable fleet of garbage trucks in compare to the very large area they serve. This situation makes it nearly impossible to evaluate such inter-component requirements (e.g. the trucks rarely run close to one another). Thus, the next step when these extra computing elements are available, is to define the physical context involved in these elements and evaluate the inter-component coordination primitives based on the provided context. This step will also enable us to test the context-bounded communication cardinality, however, it is necessary to establish direct communication channels between software components. This is also one of the limitations of the current experiment (the fourth item bellow).

Third, to fully take advantage of the context-dependent coordination primitives, we need to implement/employ context acquisition and reasoning components to concretely define the available context of the participating devices. This will be made available to the fog
developers so that they can deploy and coordinate the communication among their software components. Thus, DNR will need more development effort to accommodate the range of context that is available to use. In addition, it also needs a deployment policy to govern how each participating devices process each application flow. For example, if a participating devices does not have a particular software component installed, should it actively install the missing one or relying on its peers.

Fourth, an important characteristic of fog computing is to have "bulky" data sent to and get processed by a "nearby" software component so that communication cost/delay can be minimised. This means inter-device communication has to be strictly peer-to-peer. In the current deployment of our field trials in Fujisawa, we still use the IPv4 network where devices are usually behind firewall. This is a roadblock in realising a fully peer-to-peer communication stack for our fog computing platform. We believe with more adoption of IPv6 and 5G communication technology, this can be implemented in the future.