Enabling emergent mobile systems in the IoT : from middleware-layer communication interoperability to associated QoS analysis

Georgios Bouloukakis

Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Inria de Paris / Equipe-projet MiMove

Systèmes Mobiles Émergents dans l'IoT : de l'Interopérabilité au niveau Middleware de Communication à l'Analyse de la Qualité de Service Associée

Par **Georgios Bouloukakis**

Thèse de doctorat en Informatique

Dirigée par Valérie Issarny et Nikolaos Georgantas

Le jury est composé de :

Nalini Venkatasubramanian (Université de Californie, É-U)	Rapporteur
Fabio Costa (Université fédérale de Goias, BR)	Rapporteur
Dimitris Plexousakis (Université de Crète & ICS-FORTH, GR)	Examinateur
Pierre Sens (Université Pierre et Marie Curie, FR)	Examinateur
Gilles Privat (Orange Labs, FR)	Examinateur
Valérie Issarny (Inria, FR)	Directrice de thèse
Nikolaos Georgantas (Inria, FR)	Co-directeur de thèse

Résumé

Les applications de l'Internet des objets (IdO/IoT) se composent de divers objets en grande partie mobiles et avec des ressources limitées ou riches. Des tels dispositifs exigent des interactions légères et faiblement couplées en termes de temps, d'espace et de synchronisation. Les protocoles au niveau middleware de l'IoT prennent en charge un ou plusieurs types d'interaction (par exemple, échange de messages synchrone, streaming) assurant la communication entre objets. De plus, ils supportent différents niveaux de Qualité de service (QDS) pour cette communication par rapport aux ressources disponibles sur les dispositifs et les réseaux. Les dispositifs utilisant le même protocole middleware interagissent de manière homogène, car ils exploitent les mêmes caractéristiques fonctionnelles et de QDS. Cependant, la profusion de protocoles middleware pour l'IoT se traduit par des objets très hétérogènes. Cela nécessite des solutions d'interopérabilité avancées intégrées à des techniques de modélisation et d'évaluation de la QDS.

La principale contribution de cette thèse est d'introduire une approche et de fournir une plate-forme pour la synthèse automatique des artefacts logiciels permettant l'interopérabilité. De tels artefacts permettent l'interconnexion entre des objets mobiles qui utilisent des protocoles hétérogènes au niveau middleware. Notre plate-forme prend en charge l'évaluation de l'effectivité de l'interconnexion en termes de la QDS de bout en bout. Plus précisément, nous dérivons des conditions formelles pour des interactions réussies, et nous permettons la modélisation et l'analyse des performances ainsi que le réglage du système de bout en bout, tout en considérant plusieurs paramètres système pour l'IoT mobile.

Notre objectif est de permettre la conception et le développement de *systèmes mobiles émergents*, composés de manière dynamique à partir d'objets disponibles dans l'environnement. Notre approche s'appuie sur des abstractions d'architecture logicielle, le développement axé sur les modèles, des techniques des automates temporisés et les réseaux de files d'attente. Nous validons notre approche par l'implémentation d'un prototype et une étude de cas intégrant des protocoles hétérogènes au niveau middleware. De plus, nous analysons statistiquement à travers des simulations l'effet de la variation des paramètres du système. Les valeurs de ces paramètres sont dérivées à la fois des distributions de probabilité et des données provenant de déploiements réels. Les expériences de simulation sont comparées aux expériences effectuées sur le prototype d'essai pour évaluer la précision des résultats. Ce travail peut fournir aux concepteurs de systèmes un environnement de modélisation, d'analyse et de synthèse des logiciels, afin d'assurer un comportement précis de ces systèmes à l'éxécution.

Mots clés

Internet des objets, Interopérabilité, Middleware, Composition de logiciel, Analyse statistique, Réseaux de files d'attente

Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Inria Paris / MiMove Team

Enabling Emergent Mobile Systems in the IoT: from Middleware-layer Communication Interoperability to Associated QoS Analysis

By **Georgios Bouloukakis**

Doctoral thesis in Computer Science

Under the supervision of Valérie Issarny and Nikolaos Georgantas

The jury is composed of:

Nalini Venkatasubramanian (University of California, USA)	Reviewer
Fabio Costa (Federal University of Goias, BR)	Reviewer
Dimitris Plexousakis (University of Crete & ICS-FORTH, GR)	Examiner
Pierre Sens (Université Pierre et Marie Curie, FR)	Examiner
Gilles Privat (Orange Labs, FR)	Examiner
Valérie Issarny (Inria, FR)	Advisor
Nikolaos Georgantas (Inria, FR)	Co-advisor

Abstract

Internet of Things (IoT) applications consist of diverse Things including both resource-constrained/rich devices with a considerable portion being mobile. Such devices demand lightweight, loosely coupled interactions in terms of time, space, and synchronization. IoT middleware protocols support one or more interaction types (e.g., asynchronous messaging, streaming) ensuring Thing communication. Additionally, they introduce different *Quality of Service* (QoS) features for this communication with respect to available device and network resources. Things employing the same middleware protocol interact homogeneously, since they exploit the same functional and QoS features. However, the profusion of developed IoT middleware protocols introduces technology diversity which results in highly heterogeneous Things. Interconnecting heterogeneous Things requires mapping both their functional and QoS features. This calls for advanced interoperability solutions integrated with QoS modeling and evaluation techniques.

The main contribution of this thesis is to introduce an approach and provide a supporting platform for the automated synthesis of interoperability software *artifacts*. Such artifacts enable the interconnection between mobile Things that employ heterogeneous middleware protocols. Our platform further supports evaluating the effectiveness of the interconnection in terms of end-to-end QoS. More specifically, we derive formal conditions for successful interactions, and we enable performance modeling and analysis as well as end-to-end system tuning, while considering several system parameters related to the mobile IoT.

Our aim is to enable the design and development of *emergent mobile systems*, which are dynamically composed from available Things in the environment. Our approach relies on software architecture abstractions, model-driven development, timed automata techniques and queueing networks. We validate our approach through the development of a prototype implementation and experimentation with a case study employing heterogeneous middleware protocols. Furthermore, we statistically analyze through simulations the effect of varying system parameters. The values of such parameters are derived from both probability distributions and actual data from real deployments. Simulation experiments are compared with experiments run on the prototype implementation testbed to evaluate the accuracy of the results. This work can provide system designers with precise design-time modeling, analysis and software synthesis by using our tools, in order to ensure accurate runtime system behavior.

Key Words

Internet of Things, Interoperability, Middleware, Software Composition, Queueing Networks, Timed Automata, Statistical Analysis

To my father, *Ioannis Bouloukakis*.

Table of contents

Chapter 1

Introduction

Contents

Networking technologies enable physical world sensing and actuating devices to connect with each other. Such devices penetrate our environments and are deployed in a variety of domains such as smart-home, transportation, health-care, agriculture, etc. For instance, *Parrot Flower Power*[1] is a smart plant sensor that assesses plants' needs in real time and sends alerts to a smartphone. Connecting this device to other sensors and actuators enables the development of a wide range of systems and applications (smart agriculture). Such possibilities have appeared in massive numbers and in uncountable applications via the immediate connection to the Internet of sensors and actuators but also any physical object embedding them, a "Thing". This has established the "Internet of Things" (IoT).

A major constituent of the IoT is mobile with several IoT-enabled devices, such as, smartphones, drones, vehicles, etc. Today's smartphones may contain 15 different sensors starting from the most commonly used: accelerometer and microphone, to the latest one: near-infrared spectrometer. Mobile Things can be connected to the Internet through cellular technologies (e.g., 3G/4G/5G) or via a local area network (WiFi).

These advances contribute to the realization of a truly ubiquitous computing, or what we call *emergent mobile systems*. Such systems are dynamically composed from available mobile Things in the environment, which is at the scale of the whole Internet. However, the IoT hardware and technology diversity hamper the composition of Things. Specifically, multiple device

[1] http://global.parrot.com/au/products/flower-power/

manufacturers, operating systems, device capabilities (energy, CPU, memory) and more importantly the lack of common standards result in highly heterogeneous Things. To enable the rapid development of IoT systems that combine heterogeneous Things, HTTP-based protocols and *Application Programming Interfaces* (APIs) have been widely used. However, these protocols are often too heavyweight to be deployed directly on resource-constrained Things operating in low bandwidth environments. Accordingly, multiple middleware protocols and APIs have been introduced in the IoT to face these limitations. This effect has further been magnified by the intense industrial competition in the domain.

The introduced IoT middleware protocols integrate diverse communication styles and data representation models. Specifically considering communication styles, the *Client/Server* (CS), *Publish/Subscribe* (PS), *Data Streaming* (DS) and *Tuple Space* (TS) styles are among the most widely employed ones today, with numerous related middleware platforms. Hence, an additional level of heterogeneity is introduced at the middleware protocol level. To enable the interconnection of heterogeneous Things, advanced interoperability solutions at the middleware layer are required. Below we provide the definition of system interoperability by A. Tanenbaum:

Definition 1. (Interoperability) "characterises the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard." (in [1]).

The *European Telecommunications Standards Institute* (ETSI) has further classified interoperability as *technical*, *syntactic* and *semantic* interoperability:

Definition 2. (Technical Interoperability) "is centred on (communication) protocols and the infrastructure needed for those protocols to operate" (in [2]).

Definition 3. (Syntactic Interoperability) "deals with the format and structure of the encoding of the information exchanged among Things. It includes the middleware layer of TCP/IP stack" (in [3]).

Definition 4. (Semantic Interoperability) "is the capability of two components to interpret exchanged data identically and share a common understanding of it" (in [4]).

The above definitions focus on functional aspects of system interoperability. The annual IoT Developer Survey[2] carried out by the Eclipse IoT Working Group since 2015 identifies *interoperability* and *performance* as two out of the three most important IoT concerns (the third being security).

To deal with performance and other *Quality of Service* (QoS) requirements, middleware developers provide their APIs and protocols on top of reliable or unreliable transport-layer pro-

[2]https://goo.gl/YCXMNQ

tocols and introduce additional protocol features and mechanisms. These take into account the resource constraints of mobile Things and their possibly intermittent connectivity. However, when composing heterogeneous Things, differences in their QoS requirements and the QoS features of their middleware protocols may hamper their interconnection. Hence, enabling only functional interoperability while ignoring QoS aspects leads to ineffective IoT applications. Below we introduce the definition of *interoperability effectiveness*, which must be evaluated when interconnecting two heterogeneous Things:

> **Definition 5.** (Interoperability Effectiveness) "characterises the degree by which two heterogeneous systems can connect and collaborate, by taking into account their functional co-operation, as well as their end-to-end QoS.".

In the next sections, we introduce the research questions regarding functional and QoS-related interoperability that have motivated this work. We then provide an overview of the contributions of this thesis and the structure of this document.

1.1 IoT middleware-layer Interoperability

As already pointed out, the profusion of developed IoT middleware protocols introduces technology diversity which results in highly heterogeneous Things. IoT middleware protocols make part of the following protocol stack related to Things:

1. *physical and data-link layers*: define the characteristics of the network hardware and provide access to the local medium for data transmission. Common protocols include: ZigBee, WirelessHART, LoRaWAN
2. *network and transport layers*: manage network addressing and (reliable or unreliable) data transfer between Things. Common protocols include: IPv6/6LowPAN, TCP/UDP
3. *middleware layer*: enable Things' interactions. Common protocols include: CoAP, MQTT, XMPP, HTTP
4. *application layer*: enables data exchange with unambiguous, application-specific meaning.

Things employing different data-link layer protocols cannot interact with each other since they belong to different local networks. Accordingly, such protocols include gateways for connecting a set of Things to the Internet and thus, potentially to any other local network (via another gateway). This solution is typically applied to resource-constrained Things. To enable direct Internet connectivity for Things, solutions relying on IPv6 (successor of IPv4, offers approximately 5 x 1028 addresses for every person in the world) and on its adaptation for resource-constrained Things (e.g., 6LowPAN) have been applied. These solutions enable further the deployment of a complete protocol stack, including a middleware layer, on Things. The focus of this thesis is to provide a solution for middleware layer protocol interoperability.

As already pointed out, middleware protocols introduce different types of communication styles (CS, PS, DS or TS); these require different levels of coupling. In particular, Eugster et. al. [5] identified *semantics* for three primary dimensions of coupling:

- *Space coupling* – determines how peers identify each other and how interaction elements are routed from one peer to the other. For instance, PS interactions are decoupled in space, since publishers and subscribers interact through an intermediate broker.
- *Time coupling* – determines whether two peers need to be present and available at the same time to complete an interaction. For instance TS offers a shared memory (tuple space) that decouples peers in time.
- *Synchronisation coupling* – determines whether the initiator of an end-to-end interaction blocks or not until the interaction is complete (i.e., if the interaction is executed in a synchronous or an asynchronous way). For instance CS supports both synchronous and asynchronous interactions.

The above semantics concern functional aspects of interaction. When two Things interact using the same middleware protocol, they support the same semantics and expect the same semantics from each other. However, when interconnecting heterogeneous Things, their semantics and expectations may differ. In this case, we need to map between these heterogeneous semantics. This is undertaken by *interoperability artifacts*, which are additional software components deployed between the interacting Things. Based on the above, the research questions that this thesis aims at tackling regarding the functional semantics interoperability are the following:

- When heterogeneous Things interact, how can we map their functional semantics in order to satisfy their expectations with regard to the end-to-end functional semantics of the interaction? In particular, how can we map the space semantics? Finally, can we deal with different synchronization semantics?
- Since there is no dominant IoT protocol and the number of protocols used is growing rapidly, how can we come up with an interoperability approach covering such a technology diversity? Can we rely on some widely established integration paradigm?
- Based on the dynamic topology of the IoT, can we support the dynamic composition of heterogeneous Things? Can we automate the synthesis of interoperability artifacts for enabling the Things' dynamic composition?

1.2 QoS in the Mobile IoT

A plethora of existing approaches try to deal with the IoT interoperability challenge. However, these attempts mainly focus on the mapping of end-to-end functional semantics between hetero-

geneous Things while they ignore the mapping of their QoS (or non-functional) semantics. QoS semantics of mobile Things typically relate to their resource constraints and intermittent connectivity. More specifically, due to their limited energy resources Things may – spontaneously or driven by their users – disconnect for energy saving purposes (application-layer disconnections), while they have to deal with forced disconnections caused by the underlying wireless network connectivity (middleware-layer disconnections).

The effect of disconnections on message delivery depends on the reliability semantics of the middleware protocol, which builds atop a reliable or unreliable transport protocol. Furthermore, depending on the IoT application context, traditional request/response interactions must be completed within a *timeout* period, while the availability or validity of data in data feeds may be constrained by a *lifetime* period, after which messages are discarded. Finally, tiny Things with limited memory resources may employ small-capacity buffers and introduce additional message losses. To deal with the QoS semantics of heterogeneous Things, as a first step, it is essential to ensure the mapping of their functional semantics and afterwards map their QoS semantics. In this thesis we evaluate the *interoperability effectiveness* of heterogeneous Things by taking into account their end-to-end functional and QoS semantics. Interoperability effectiveness refers to the following problems:

- For a given deployment topology (interconnection of heterogeneous Things), load (number of Things, input arrival rates), and configuration (intermittent connectivity, data lifetime, etc), what performance would the system exhibit?
- Interconnecting heterogeneous Things, requires interoperability artifacts. What is the performance overhead of such components? Can we evaluate the performance of such interconnections?
- Can we evaluate end-to-end interconnections of Things as part of a given real-life application scenario (e.g., commuters connecting inside the metro)?

1.3 An overview of this work

In the previous sections, we introduced the IoT middleware-layer interoperability issue and underlined the research questions concerning achieving effective interoperability under both functional and QoS constraints of the mobile IoT.

The goal of this thesis is to achieve interoperability between heterogeneous IoT peers and evaluate its effectiveness. Our thesis statement is the following:

> "To enable heterogeneous interactions in the (mobile) IoT, it is required to ensure the Things' interoperability along with a certain QoS level. By analyzing the Things' middleware protocols semantics (functional and QoS), we can automatically

synthesize interoperability artifacts, as well as build end-to-end performance models. Such artifacts enable functional middleware-layer interoperability between two Things, while the performance models can be used for evaluating the Things' interoperability effectiveness."

In the next subsections, we outline the contributions of this thesis, followed by a summary of the remainder of this document.

1.3.1 Contributions of the thesis

As depicted in Fig. 1.1, the target of this thesis is to provide a platform for enabling application developers and system architects to solve the interoperability problem and evaluate the interoperability effectiveness in heterogeneous IoT applications. Towards this, below we describe our contributions, which follow three main steps:

Automated synthesis of interoperability artifacts. The 1st contribution of this thesis concerns the automated synthesis of runtime artifacts (step (1) in Fig. 1.1) which ensure the mapping of functional semantics between heterogeneous Things. More specifically, we introduce models for the core communications styles (CS, PS, DS and TS) that map space and synchronization semantics of concrete middleware protocols. Subsequently, we build the *Generic Middleware* (GM) *connector model*, which comprehensively abstracts and represents the semantics of the CS, PS, DS and TS styles. Hence, such an abstraction supports multiple existing middleware protocols, as well as future protocols that will follow one of the supported styles. By relying on *model-driven development* we create a metamodel that abstractly represents the communication and data interface of Things and is used for the automated synthesis of the interoperability artifacts. Inspired by the *Enterprise Service Bus* (ESB) paradigm, we introduce the *eVolution Service Bus* (VSB) which is a lightweight bus applying our interoperability approach. VSB is utilized as core component of the H2020 CHOReVOLUTION project[3] and enables heterogeneous interactions in IoT choreographies.

Formal timed protocol analysis. The 2nd contribution of this thesis concerns the timed protocol analysis of heterogeneous mobile Things, and further of their interconnections (step (2) in Fig. 1.1). Our model captures several QoS semantics, such as limited validity/availability of data, as well as intermittent availability of the data recipients. In particular, we rely on the GM connector model, which abstracts middleware protocol semantics. Our analysis provides us with formal conditions (by relying on *Timed Automata* models) for achieving successful GM interactions after taking into account the introduced timing parameters. We further perform statistical analysis by varying these parameters. We demonstrate that varying them has a significant effect on the rate of successful interactions.

[3] www.chorevolution.eu

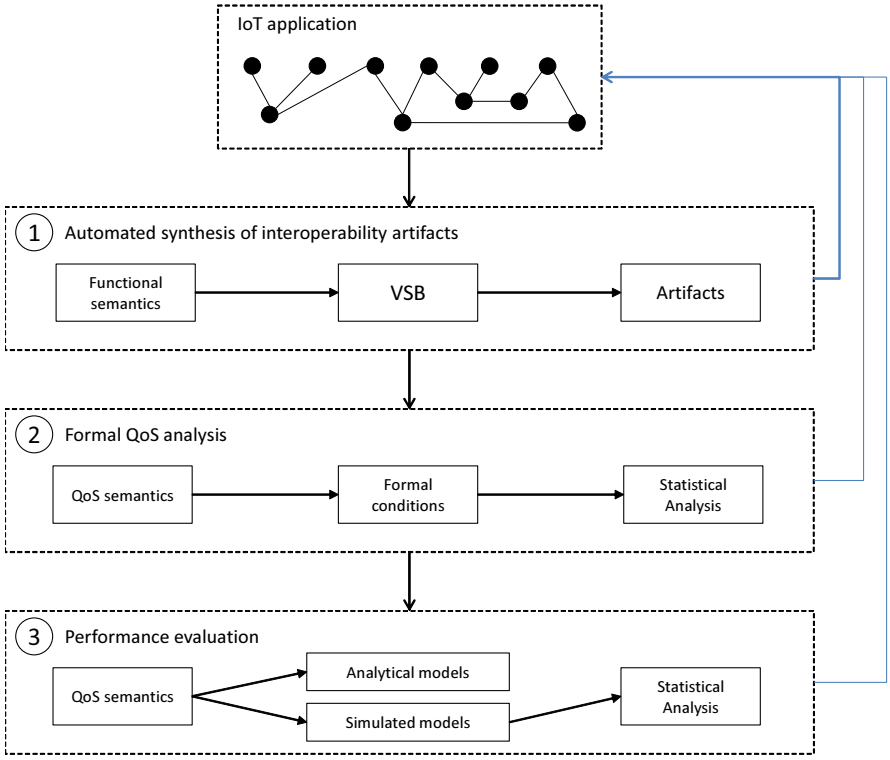


Figure 1.1: Platform for interoperability and interoperability effectiveness evaluation in IoT applications.

Performance evaluation of heterogeneous interactions. The 3rd contribution of this thesis concerns the performance evaluation of heterogeneous mobile Things, and further of their interconnections (step (3) in Fig. 1.1). To deal with semantics related to the Things' intermittent connectivity, we introduce an analytical model for the "ON/OFF queueing center", which can then be used as separate component inside *Queueing Network Models* (QNMs). By relying on QNMs, we are able to model the end-to-end infrastructure of IoT middleware protocols. Thus, we introduce performance models for typical interactions of the CS, PS, DS and TS core communication styles, which we call *performance modeling patterns*. These patterns capture QoS semantics such as: reliable/unreliable protocols, arrival rates, disconnections, service times, lifetimes/timeouts and buffer capacity. Furthermore, we provide a methodology for the composition of the performance modeling patterns in order to evaluate interconnected heterogeneous Things. Finally, we apply the above models to perform simulation-based analysis of heterogeneous IoT systems.

Experimenting with real-world cases. As part of our research, we concerned ourselves with the applicability of our results. We demonstrate the validity of our approach through the comparison of the analytical and simulation model with experiments run on our VSB testbed with respect to the accuracy of predicted results.

Furthermore, we validate our analytical and simulation models, by applying two real-world

workload traces:

- the 1st dataset was provided to us by *Orange Labs* in the context of the *D4D challenge*. The D4D dataset contains *Call Detail Records* (CDRs, which also contain SMS information) for each 3G/4G antenna in the whole country of Senegal, collected over a period of approximately one year. Using this dataset we apply realistic traffic flows (or arrival rates) to our queueing models by assuming that mobile IoT applications relying on data crowdsourcing have similar user access patterns to 3G/4G mobile services. Utilizing antenna traces in this way can provide insights to system designers for resource capacity planning in related areas.

- the 2nd dataset was collected by us in the context of the *Sarathi* international project between Inria and IIIT-Delhi inside the metro of Paris and Delhi. The Sarathi dataset was collected through an android application that captures the users' network connectivity, which is utilized to parameterize and validate our queueing models. By analyzing the network connectivity traces, we provide additional insights to system designers for selecting the proper middleware protocols and application-layer QoS semantics.

Much of this work has already been published in peer reviewed conferences, or as research reports. Hereafter, we enumerate the most important ones:

- In [6–8], we present our CS, PS, DS, TS and GM connector models. We then demonstrate the case of IoT middleware interconnection using the VSB framework.

- In [9], we present the timed analysis and formal properties of heterogeneous middleware interactions using timed automata models.

- In [10], we present our general approach for the modeling of middleware protocol infrastructures in the IoT using QNMs. We demonstrate the applicability of our approach by modeling data streaming protocols.

- In [11], we apply the same modeling approach to the publish/subscribe communication style. This work evaluates the end-to-end performance of peers operating in large scale.

- In [12, 13], we initiate the analysis of the D4D dataset and the collection/analysis of the Sarathi dataset.

1.3.2 Structure of the document

The remainder of this dissertation is structured as follows.

- Chapter 2 surveys state of the art approaches for: *i)* achieving IoT interoperability at different protocol layers; *ii)* achieving middleware-layer cross-protocol interoperability;

iii) evaluating system performance in the mobile IoT by using formal analysis techniques and queueing theory.

- Chapter 3 identifies the functional semantics of the core communication styles: CS, PS, DS and DS. Then, an abstraction of these semantics is provided through the GM connector model. Finally, the VSB framework implements the GM connector and provides the automated synthesis of artifacts enabling cross-middleware protocol interoperability in the IoT.

- Chapter 4 provides time modeling of GM interactions using parameters such as, intermittent Things connectivity, limited data validity, etc. Then, a formal analysis is performed by using timed automata, which provides us with formal conditions for successful/failed interactions. Finally, a statistical analysis is carried out, based both on simulation and a real deployment, in order to study the trade-off between delivery success rates and response times.

- Chapter 5 explores the end-to-end infrastructure of middleware IoT protocols and provides a more comprehensive solution to the performance modeling by utilizing QNMs. Real-world datasets are utilized to validate the presented models and evaluate the end-to-end interoperability effectiveness.

- Chapter 6 provides our conclusions and identifies challenges and research perspectives that require further exploration.

Chapter 2

Interoperability and QoS in the Mobile IoT: State of the Art

Contents

The IoT promises the easy integration of the physical world into computer-based systems. In effect, real-world objects become connected to the virtual world, which allows for the remote sensing/acting upon the physical world by computing systems. Improved efficiency and accuracy are expected from this paradigm shift. However, enacting IoT based systems is still raising tremendous challenges for the supporting infrastructure from the networking up to the application layers. Key challenges [14, 15] relate to *deep heterogeneity*, *performance*, *scale*, and many others.

This chapter surveys the state of the art approaches that deal with the heterogeneity and QoS challenges. More specifically, in Section 2.1 we provide an overview of the existing approaches

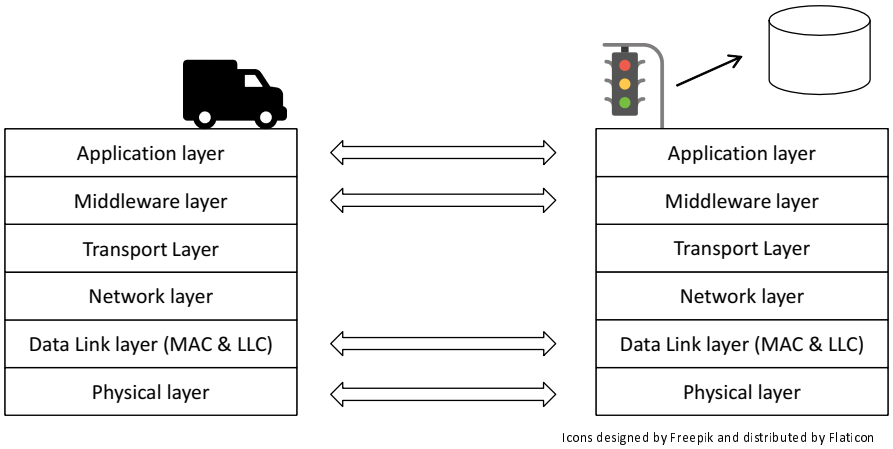


Figure 2.1: IoT Interoperability at multiple layers.

in the IoT for enabling interoperability at different system layers. Section 2.2 focuses at the middleware layer presenting the existing IoT protocols and the research efforts for ensuring interoperability among Things employing such protocols. Then, in Section 2.3 we provide an overview of existing techniques for evaluating the performance of mobile systems employing IoT protocols. We finally complement this chapter with a brief summary in Section 2.4.

2.1 IoT Interoperability

To deal with the IoT *heterogeneity* challenge, the industrial and research community work to achieve the interoperability at multiple system layers. While there is an undoubted agreement about the ubiquity of IP protocols at the network and transport layers, there is a large diversity in the other layers (as depicted in Fig. 2.1). This results in different radio and medium access technologies at the physical/data link layers, numerous protocols, APIs and data representations at the middleware layer, and finally different data and protocol semantics at the application layer.

In what follows, we provide a clear picture of the interoperability solutions at each layer except for the middleware layer, which is discussed in detail in Section 2.2.

2.1.1 MAC Layer

IoT applications contain a variety of *Things* - such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile devices, etc. Depending on the application domain, Things interact with each other to reach common goals. For instance, home automation can be achieved by coordinating smart thermostats, ventilation, air conditioning, security sensors, as well as home appliances such as washers/dryers. Well known communication protocols provide radio access to Things such as WiFi, Bluetooth, ZigBee and 2G/3G/4G cellular.

The above protocols correspond to the physical and data link layers which are combined by most standards. The data link layer is slitted into the MAC (Media Access Control) sub-layer and

the LLC (Logical Link Control) sub-layer. In the remaining sections we refer to the data link layer as MAC layer. By employing a specific protocol, two sensors of the same local area can interact with each other or a set of sensors can interact with the gateway device that connects them to the Internet (sensor gateway). Depending on the application and the participating Things, factors such as range, data requirements, security and battery life, determine the appropriate choice of one or the combination of more protocols to support the application. Bellow we present the most commonly used IoT MAC protocols, as well as the most recent ones. Note that we do not present their technical details, since our aim is to provide an overview of the IoT protocols at this level. Interested readers will find references to technical publications for each specific technology.

IEEE 802.15.4 [16] is the most commonly used IoT standard in the MAC layer. ZigBee [17] and WirelessHART [18] operate on top of IEEE 802.15.4. ZigBee supports low-power operation, high security, robustness and high scalability using stochastic address assignment. In comparison, WirelessHART is more suitable for industrial applications and requirements [19]. Bluetooth low energy (BLE) [20] is a short range protocol widely used for in-vehicle networking. In comparison to the classical Bluetooth, its energy consumption is ten times lower and its latency 15 times shorter [20].

Z-Wave [21] is a low-power protocol primarily designed for home automation for products such as lamp controllers and sensors among many others. It is reliable, of low-latency and it covers about 30-meter point-to-point communication. It is suitable for small messages, light control, energy control, wearable healthcare control and others. While Wi-Fi is widely adopted by digital devices including laptops, mobiles, tablets, and digital TVs, it is not suitable for IoT applications. Hence, the IEEE 802.11 working group initiated 802.11ah [22] task group to develop a standard that supports low overhead and power friendly communication suitable for sensors and actuators. While the above protocols do not support coverage for wide areas, recent protocols such as LoRaWAN [23] and Weightless [24], support wide-area wireless networking for the development of IoT applications. These protocols are optimized for low-power consumption and they support large networks with millions of devices. Data rates range from 0.3 kbps to 50 kbps for LoRaWAN and from a few bits per second up to 100kbps for Weightless.

The aforementioned protocols constitute a small set of existing IoT MAC protocols. Authors in [25] provide a comprehensive survey and technical details of additional related protocols. Selecting the appropriate protocol for each application may result in multiple networks overlapping in a local area (e.g., smart building with multiple IoT networks). Creating interoperable networks, requires solving issues related to interference and wireless coexistence. Authors in [26] introduce *Gap Sense*, a novel mechanism that can coordinate heterogeneous devices without modifying their physical layer modulation schemes. Similarly in [27], authors exploit cross-technology interference to set up a low-rate bidirectional communication channel between

Protocol	Data Rates	Range	IP Support
ZigBee [17]	250Kbps	10-100m	ZigBee IP
WirelessHART [18]	250Kbps	50-250m	HART-IP
BLE [20]	1Mbps	50-150m	BLE 4.1
Z-Wave [21]	9.6/40/100Kbps	30m	Z/IP Gateways
802.11ah [22]	150Kbps to 346Mbps	1km	yes
LoRaWAN [23]	0.3 to 50Kbps	2-5km	IP Gateways
Weightless [24]	up to 100Kbps	10km	IP Gateways

Table 2.1: MAC layer protocols.

heterogeneous WiFi and ZigBee networks. To enable interoperability between sensor networks operating in different local areas, *sensor gateways* [28–30] can be utilized for connecting them to the Internet. Among many approaches, authors in [28] introduce an IoT Gateway solution where a smartphone becomes a universal interface between various Things and the Internet. Bellow we provide some key network level (IP-based) technologies aiming to solve the heterogeneity of MAC protocols at the network level.

Network layer protocols. A key attribute for the network layer is IPv6, which has be a key enabler for the IoT. IPv6 is the successor of IPv4 and offers approximately 5 x 1028 addresses for every person in the world, enabling any embedded object or device in the world to have its own unique IP address and connect to the Internet. However, one major issue is that IPv6 addresses are too long and cannot fit in most IoT datalink frames which are relatively much smaller. Hence, IETF is developing a set of standards to encapsulate IPv6 datagrams in different datalink layer frames for use in IoT applications. A key related technology is 6LowPAN (IPv6 Low-power wireless Personal Area Network) [31]. 6LowPAN defines encapsulation and header compression mechanisms. The standard has the freedom of frequency band and physical layer and can also be used across multiple communications platforms, including Ethernet, Wi-Fi and IEEE 802.15.4.

Table 2.1, provides a summary of the above MAC layer protocols along with some of their characteristics, and indicates the ones that have been extended to support connection to IPv6 (directly or via a gateway). In the next subsection we provide a brief discussion regarding the interoperability efforts at the application layer.

2.1.2 Application Layer

Different APIs and data representations between Things can be mapped with each other at the middleware layer. However, this only does not make the interacting peers fully interoperable. For instance, a traffic light may provide information regarding its status through the following operation: `get_traffic_lights_status (id, status)`. However, a vehicle may require the traffic light status through: `query_traffic_signal (signal_id, signal_color)`. Such issues at the application layer can be qualified as *semantic interoperability* issues. Ensuring end-to-end data consistency is among the challenges listed in [25], and it is one of the goals of semantic

interoperability.

There are two basic solutions for achieving semantic interoperability between two Things. The first solution is an one-to-one model mapping. However, this approach is not scalable in complex systems, where several different data models can coexist – this is the case in many IoT architectures. Another more suitable approach is to use shared data meta-models that can be used to unambiguously define the meaning of terms in existing models, such as ontologies. Data models can be annotated to be aligned with ontologies, and raw data can be enriched to become semantically enabled. To enrich raw data using ontologies it is essential to create a knowledge base, which is the association between the data and the ontology. SemioTics [32] is an autonomic application, featuring a knowledge base as its core component. In particular, data generated by sensors are enriched by SemioTics using the following ontologies: *i) Semantic Sensor Network* (SSN) [33] for sensors and observations, and *ii)* IoT-O [34] for IoT-actuators, devices and services.

Various research projects exploit *Semantic Web* technologies to ensure semantic level machine to machine interoperability [35–38]. Authors in [35] present a novel semantic level interoperability architecture that enables different devices to interact with each other only by sharing semantic information via common knowledge publish/subscribe brokers. INTER-IoT project [39] studies multiple ontologies developed to cover the semantics of several IoT domains (e.g., health and transportation) in order to facilitate interoperability across the IoT landscape. Generally, there are multiple ontologies dealing with various aspects of sensors and sensing (with different scope, granularity and generality). Authors in [40] propose an ontology named FIESTA-IoT (under the EU H2020's FIESTA-IoT project [41]) aiming to achieve semantic interoperability among heterogeneous testbeds. To build the ontology, they have integrated a number of mainstream ontologies and taxonomies, such as SSN [33], M3-lite [42], IoT-lite [43], Time [44], and DUL [45], into a single and holistic one, in order to fulfill the needs of the testbeds (and experimenters).

The next section provides an overview of the most widely used IoT middleware protocols, as well as recent efforts for enabling interoperability among them.

2.2 Protocol Interoperability at the Middleware Layer

In the previous section we provided a brief overview of the research landscape in the *IoT interoperability* topic where researchers work at different layers, i.e., at MAC and application layers. This thesis focuses on the problem of interoperability among IoT middleware protocols. As partially discussed in the previous section, depending on their available resources, Things may or may not host a complete protocol stack (including a middleware protocol) enabling their direct connection to the Internet. In the latter case, access to the Things is performed through a proxy/gateway. With the technological evolution of sensor nodes, the former approach is now attracting much attention, as it enables autonomous Things. The authors in [57] undertake this

Protocol	Interactions	Weaknesses	Strengths	Other Characteristics
DPWS [46]	request/response; streaming;	noticeable protocol overhead; amount of memory used;	SOA; large-scale deployments; for resource constrained devices;	introduced in 2004; OASIS open standard;
OPC UA [47]	request/response; streaming;	not suitable for IoT;	SOA; highly resource constrained devices;	designed in 2008 by the OPC foundation;
CoAP [48]	request/response; streaming;	high latency for large payloads; request/response affects battery usage;	highly resource constrained devices; suitable for small payloads;	designed by IETF;
REST [49]	request/response;	not suitable for resource constrained devices;	mobile development;	supported by multiple IoT platforms;
XMPP [50]	request/response; streaming;	additional overhead due to XML data formats;	suitable for real-time applications;	standardized by the IETF a decade ago;
JMS [51]	streaming;	focused on Java-centric systems;	support for underlying messaging protocols; widely used;	standard by Sun Microsystems;
DDS [52]	request/response; streaming;	development and configuration complexity;	real-time applications;	brokerless messaging protocol;
MQTT [53]	streaming;	not suitable for large payloads;	highly resource constrained devices;	centralized architecture; OASIS standard;
AMQP [54]	streaming;	not suitable for resource constrained devices;	supports high traffic load;	ISO/IEC standard;
SemiSpace [55]	request/response;	not widely used;	distributed architecture of shared spaces;	based on JavaSpaces;
WebSockets [56]	streaming;	not suitable for resource-constrained devices;	real-time full duplex interactions; only 2 bytes overhead;	part of HTML 5 initiative;

Table 2.2: Comparison of IoT protocols.

approach by deploying SOAP-based Web services directly on the nodes without using gateways. DPWS [46] was introduced in 2004 as an open standard by OASIS. It is suitable for supporting large-scale deployments and mobile devices. However the introduced protocol overhead is noticeable and it requires large amount of memory. Hence, at the same time, deploying the middleware component directly on the device might cause several issues, such as message delays, limited supported interactions, limited computational capacity, high energy consumption, etc.

Several other middleware protocols have been developed to address the above issues, along with standardization efforts that will guarantee interoperability. Table 2.2 summarizes existing middleware protocols along with their supported interactions, strengths and weaknesses with regard to IoT applications. Specifically, OPC UA [47] was designed in 2008 by the OPC foundation targeting resource constrained devices. Similarly to DPWS, it introduces a large payload unsuitable for IoT applications. Due to the complexity and the limitations of the above protocols, IoT developers turned to simpler protocols. Among them, REST [49], is not really a protocol but an architectural style. It is ideal for mobile development but is not suitable for resource constrained devices. Hence, IETF designed CoAP [48], a lightweight protocol which supports highly resource constrained devices and the delivery of small message payloads. Despite the fact that CoAP supports extremely low-resource interactions, it is more suitable for request/response interactions. However, the performance of CoAP decreases significantly when transmitting large

message payloads and the request/response interaction style affects the battery usage. Finally, XMPP [50] despite the fact that it was standardized by the IETF over a decade ago, it re-gained a lot of attention as a suitable protocol for IoT real-time communications. However, it uses XML data formats that create considerable computational overhead.

To provide alternatives to the request/response style and offer time decoupled communication, middleware developers introduced several middleware protocols that follow the publish/-subscribe communication style. JMS [51], a standard by Sun Microsystems, has been one of most successful asynchronous messaging technologies available; it defines an API for building messaging systems. It is not a messaging protocol, hence, it is possible to build on top of several messaging protocols. DDS [52] is a messaging protocol designed for brokerless architectures and real-time applications. AMQP [54] is another messaging protocol designed to support applications with high message traffic rates. However is not suitable for resource constrained devices. To support highly resource constrained devices, MQTT [53] offers a publish/subscribe centralized architecture. However, MQTT performance decreases significantly when sending large message payloads. Leveraging the grouped reception of messages in response to a request addressed to a shared-memory, developers of Semispace [55] developed a lightweight middleware by relying on JavaSpaces [58]. Such a middleware reduces energy consumption since it receives grouped messages and avoids HTTP long polling notifications which affect battery usage. Finally, as part of the HTML 5 initiative, WebSockets [56] was introduced to support real-time full duplex (streaming) interactions, using only two bytes of overhead in message payloads.

The authors in [59–61] compare the most promising IoT middleware protocols: DPWS, CoAP, MQTT, Websockets, XMPP, REST and AMQP. They recommend combining one or more protocols in an IoT application to better exploit the physical network infrastructure. However, this comes with increased heterogeneity. Solving the interoperability problem is challenging, especially due to the fast development of protocols and APIs aiming to support IoT applications. Below we provide the most recent efforts of the research and industrial community coping with Things interoperability at the middleware layer. In particular, paradigms such as, *Service-oriented Architecture* (SOA), *Gateways*, *Cloud computing* (CC), *Model driven Architecture* (MDA) and *Software Defined Networks* (SDN) have been used to provide middleware interoperability solutions for Things.

2.2.1 Service-oriented approaches

Traditional SOA involves three main actors that interact directly with each other: a *Service Provider*, a *Service Consumer*, and a *Registry* for services. Any service-oriented middleware adopting this architecture supports three core functionalities: *Discovery*, *Composition* of, and *Access* to services. More specifically, *Discovery* is used to publish (register) services in registries that hold service metadata and to look up services that can satisfy a specific request. *Compo-*

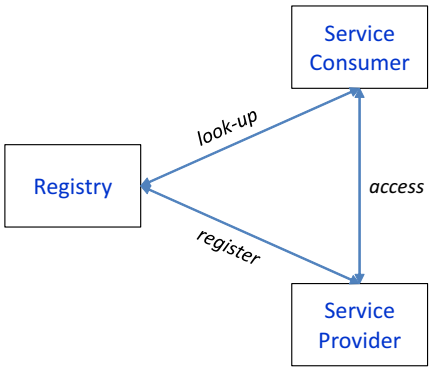


Figure 2.2: Service-oriented architecture (SOA).

sition of services is used when discovered services are unable to individually fulfill the request. In such case, existing services are combined to provide a new convenient functionality. The composed services can further be used for more complex compositions. Finally, *Access* enables interaction with the discovered services. This basic SOA architecture is shown in Fig. 2.2.

While many attempts in the literature support *Discovery* and *Composition* for building service-oriented middleware platforms, this section is focused on early attempts regarding the *Access* functionality. The *access* mechanism of traditional SOA enables the interaction between service consumers and service providers. In particular, services interact in a unified way following specific data formats on top of common overlay infrastructures across different system platforms. Web services constitute the dominant technology in SOA, with well known protocols such as SOAP or REST as the overlay infrastructure. The research community and many businesses have adopted these protocols and their standards in order to describe and implement their services (i.e., the supported operations, data formats, etc). The existence of standards, WSDL for SOAP and WADL for REST, facilitates the development of frameworks and the wrapping of systems for interoperability.

The challenges that the IoT is raising in the development of computing systems along with perspectives on how to address them have been the focus of numerous papers over the last decade, such as in: [14, 62–64]. Among the software architecture paradigms envisioned for IoT-based systems, the literature suggests that service-orientation is promising due to its inherent support for interoperability and composability [65]. A large number of *Service-oriented Middleware* (SOM) platforms have then been proposed for the IoT, which subsumes revisiting the core elements of the service-oriented architecture paradigm starting with the service abstraction itself.

To build an IoT platform that supports the SOA functionalities, the starting point is to abstract Things or their measurements as services [66–71]. Compared to the classical *Business services*, *Thing-based services* must encompass highly heterogeneous software entities among which resource-constrained ones [72]. An early attempt in that direction is illustrated by the SenseWrap middleware, which features virtual sensors that deal with the transparent discovery

of the supporting resources using ZeroConf protocols [70].

Authors in [73] focus on the diversity of the IoT word and propose to deal with several issues by building dynamically the service needed and then integrate it in the whole composition. Based on this approach (Object-as-a-Service), a Thing is represented as an *object* that offers a way to developers creating on-the-fly (dynamically) a service with several functionalities such as sensing, actuating and computing. A similar approach with a complete architecture for designing IoT applications is presented in [74, 75]. D-LITe provides universal access to object's functionalities and its features are discovered and developed through the network, without any physical access. The logical behavior of an IoT application is described through a specific language (SALT) that includes the corresponding objects. The global composition of logical units corresponds to a service *Choreography*.

Choreography is seen like a system where the nodes accomplish some actions according to a set of rules that they previously learned and now they collaborate in order to realize a task. Such a system is not centralized. In order to bridge between different communication protocols, the *Enterprise Service Bus* (ESB) paradigm [76] is the predominant solution for the integration of heterogeneous systems. Swarm [77, 78] is a communication paradigm based on the ESB architecture where its nodes collaborate based on the service choreography approach for realizing a task. In comparison to a choreography, nodes interact with each other based on a set of rules created by smart messages. Authors have developed an open source project, SwarmESB, that enables integration between heterogeneous components.

In our previous work [7, 79, 80] we identified the three main communication styles at the middleware level, client/server, publish/subscribe and tuple space, and we introduced a higher-level one, the *Generic Application* (GA) communication style. For each one these styles, we elicited an abstract API comprising a set of primitives. By relying on these primitives, we had developed the *eXtensible Service Bus* (XSB), which relies on the EasyESB bus protocol. In [81–83], authors introduce the *Lightweight Internet of Things Service Bus* (LISA), for tackling the IoT heterogeneity. LISA facilitates developers by providing an API for resource-constrained devices offering discovery, registration and authentication. Devices deployed based on different standards interact via a common communications protocol.

The ESB paradigm is utilized in [84] as its infrastructure. It integrates five modules including an event processing module, a publish/subscribe module, a service coordination process, a control server module and an HTTP Binding Component. Among the various modules the publish/subscribe one decouples publishers and subscribers in terms of space and time semantics. The HTTP BC is used to retrieve requests from clients. The main scope of this system is to facilitate the asynchronous communication and on-demand distribution of sensory data in a large-scale distributed IoT environment. To support local wireless networks, authors in [85] adopt the ESB paradigm and provide an implementation of a *Home Service Bus* for solving

Gateways	REST	CoAP	MQTT	XMPP	MAC protocols
Ponte [87]	+	+	+	−	−
Intel IoT Gateway [88]	−	−	+	−	+
HyperCat [89]	+	−	−	−	+
GoThings [90]	+	+	+	+	−
Semantic gateway [91]	−	+	+	+	−
Centric gateway [92]	+	−	−	−	+
Enhanced MQTT [15]	+	+	+	+	+

Table 2.3: IoT gateways and their supported middleware protocols.

interoperability problems among embedded electronic devices and resource-contained sensors in smart-home environment. Similarly, an ESB-based industrial middleware is proposed in [86] where multiple sensor gateways (the main proxy that connects a MAC layer network to the Internet) are part of SOA.

2.2.2 Gateway-based approaches

The existence of middleware and MAC layer protocols (see subsection 2.1.1) in IoT applications, bring another dimension of interoperability in the IoT between different layers. A common approach to connect a set of sensors and actuators (interacting using MAC layer protocols) to the Internet is through *sensor Gateways*. Hence, to integrate Things that employ multiple (MAC and middleware) protocols it is essential to develop *intelligent IoT Gateways* [15, 25]. Table 2.3 provides an overview of existing (industrial and academic) gateways along with the supported IoT protocols.

Inspired by the fact that CoAP can be deployed on resource constrained devices, authors in [93] bridge the gap between Things and the Web. In particular, QEST broker is a gateway that enables interoperability between CoAP and REST protocols. Similar gateways exist in the literature proving a cross-protocol proxy, such as in [94] for HTTP-CoAP interoperability and in [95] for DPWS-REST interoperability. By extending QEST, Ponte [87], which is developed under the Eclipse IoT project [96], provides APIs to application developers enabling the automatic conversion between REST, CoAP and MQTT. Developers are able to utilize Ponte as a gateway and deploy their Things for enabling their interconnection. However, it does not provide support to MAC layer protocols. To facilitate the development of IoT applications, the Eclipse IoT project contains other sub projects such as Kura [97], SCADA [98], SmartHome [99] and Krikkit [100]. However, these projects provide limited support for IP-based protocols.

In addition to Ponte, Intel supports IoT applications through a commercial smart device that acts as an *IoT Gateway* [88]. Its primary purpose is to provide secure connectivity among Things and the Cloud. Supported protocols belong mainly to the MAC layer, such as ZigBee, Cellular 2G/3G/4G, Bluetooth and Wi-Fi. Regarding middleware protocols, MQTT is supported. Connecting local sensor networks to the Web enables web developers to access resources such as environmental sensors, home appliances, etc. Authors in [89] propose a hub based approach

in which multiple sensor gateways (one from each sensor local network) connect to the Web through the HyperCat proxy. Despite the fact that HyperCat aggregates Web applications and local sensor networks, its REST interface does not allow to interconnect applications developed using other middleware protocols.

Focusing on middleware layer protocols, authors in [90] develop an *inter-operable* and *extensible* gateway. Particularly, the gateway provides interoperability between REST, CoAP, MQTT and XMPP protocols. Request/response and publish/subscribe messaging patterns are supported. Furthermore, the architecture of the gateway allows the addition of new protocols via plugins. Indeed such an architecture provides extensibility, however authors do not specify in detail which message patterns they support (e.g., synchronous, asynchronous, streaming). By using semantic technologies, authors in [91] take a step further by providing a gateway that: *i)* bridge XMPP, CoAP, and MQTT; and *ii)* annotates exchanged messages with a sensor description through the SSN ontology.

Aiming to provide more intelligent gateways, authors in [92] propose a lightweight wireless gateway that integrates Web Services and Things communicating through MAC layer protocols. Mobile devices connect through a REST API and resource contained devices (e.g., actuator, light sensor, etc) connect through multiple sensor gateways. Sensor gateways associate metadata (in SenML, a JSON metadata representation) to the sensor and actuator measurements. The latter facilitates the Things' discovery, which is one of the main novelties of the wireless gateway. Authors extend the above work by encasing the associated metadata using the CoRE Link Format [101, 102]. Based on the aforementioned gateways, authors in [15] claim that an intelligent gateway should enable application developers to exploit application-specific communication patterns to achieve more efficient protocols translations. Accordingly, they revisit the MQTT protocol by introducing a high level rule-based language that enables application developers to interconnect different MAC and middleware protocols. The rule-based language enable them to switch between different communication patterns (e.g., broker-less, changing message priorities, etc) that results to the improvement of the QoS and reliability.

The above approaches focus more on the integration of MAC and middleware protocols through a gateway. However, the gateway can be shared among different users and it should be customized according to the use case needs. Thus, authors in [103] utilize container virtualization technologies which improve scalability and energy efficiency.

2.2.3 Cloud-based approaches

Connecting Things to the Internet arise new challenges in the area of *Big Data* for both the academic and industrial communities. Data coming from multiple IoT ecosystems such as environment, agriculture, transportation, etc, require mechanisms to store, process, and retrieve them. Since IoT employs a large number of devices which results to the generation of a huge

Platform	Sensor gateway	REST	CoAP	MQTT	XMPP
Axeda	+	+	−	−	−
Nimbits	−	+	−	+	−
OnePlatform	+	+	+	+	−
SensorCloud	+	+	−	−	−
SmartThings	+	+	−	−	−
Thingworx	−	+	−	−	+
Xively	+	+	−	−	+

Table 2.4: IoT platforms and their supported middleware protocols.

amount of data, *Cloud computing* (CC) enables researchers and businesses to use it and maintain many resources remotely, reliably and at a low cost.

Data analysis bring knowledge and subsequently a competitive advantage for each business. Accordingly, many commercial IoT platforms are provided with different capabilities and strengths, such as: *i)* exposed accessible APIs which allow the Things' deployment; *ii)* interoperability among several middleware protocols, sensor gateways and multiple data formats; *iii)* a user interface for Things control and data visualization. Table 2.4 summarizes some characteristics of several available Cloud platforms (summary derived from [25]) to support the development of IoT applications. We emphasize on the support of middleware protocols and gateways.

Each IoT platform can be used to offload data generated by sensors, typically accepted in a specific format and through a specific protocol. However, the provision of multiple IoT platforms has led researchers to investigate several solutions for their integration [104]. symbIoT [105] is an H2020 research and innovation project that aims to provide an interoperability framework for IoT platforms to simplify the cross-platform application development. This is achieved by introducing the *symbIoTe internetworking API* which enables generic access to visualized resources exposed by the actual underlying IoT platforms. Additionally, *symbIoTe high-level APIs* are provided facilitate the development of end-user applications.

Even symbIoT provides APIs to integrate multiple IoT platforms, an inter-operable IoT ecosystem may require data from multiple IoT domains (e.g., environment, transport), contexts (e.g., different cities), etc. Authors in [106] present the model of an IoT ecosystem including five key interoperability patterns, which need to be supported. For instance, the *cross application domain access* pattern which is employed by an application that gathers data form different domains (O3 air quality information, average speed of traffic monitoring for providing healthy bicycle routes).

2.2.4 Model driven approaches

To develop a simple application for home automation, one developer has to install the sensors, use a specific gateway for enabling their interconnection and finally implement the application through the offered API. Introducing a new device later at some point it might require the use of a different gateway (e.g., the communication protocol is not supported at the 1st gateway).

Thus, the developer has to be aware of the new API for introducing the new device.

Through the above procedure, application developers must be aware of several APIs for building IoT applications. Additionally, such a procedure does not facilitate them to identify interaction incompatibilities. For instance, the combination of different communication styles in the above scenario, such as request/response and publish/subscribe. *Model Driven Architecture* (MDA) [107] proposes to specify applications using an abstract model, i.e., the *Process Independent Model* (PIM). The PIM is deployed atop middleware platforms described by the *Platform Specific Model* (PSM). This decoupling enables the modeling of application-middleware data dependencies, which may facilitate interoperability when used in relation with an interoperability architecture.

Existing open source frameworks provide abstract models that facilitate application developers to specify and generate software artifacts. Vorto [108] specifies a metamodel using the *Eclipse Modeling Framework* (EMF) [109]. Using this metamodel, an application developer is able to provide the information model of its device (such as capabilities of a device, exposing properties, operations and events), which is then stored to a global repository. Vorto uses the introduced model and generates code that allows developers to include it into their applications (e.g., code script to measure the voice level in an android app) operating in various environments and ecosystems. Similarly, Franca [110] provides a tool environment to define software interfaces. Its purpose is to facilitate software and system integration for automotive and infotainment industries.

To tackle the heterogeneity issue, authors in [111] agree that MDA could be applied using abstract models representing heterogeneous systems in the IoT. Indeed, MDA can be used to define abstract models and through them generate code scripts. Nevertheless, its capabilities are not limited only to this procedure. With regard to the heterogeneity issue, MDA offers a principled approach to engineer interoperable solutions. In other words, multiple developers participate to the automated generation, and subsequently to the testing phase. In [112], authors define a model based interoperability testing approach and provide a modeling and testing tool. Multiple developers collaborate to develop a specific use case that contains multiple interactions. Results show that the tool has significant value to quickly identify interoperability errors in large complex environments (e.g., incompatibility between different interaction types) and hence reduce development costs.

2.2.5 SDN approaches

Traditional networks are static and inflexible. For instance, a network administrator is able to configure a router (a network layer device) via a command line interface and then use it to design a network. The resulting network is static and any modification on its configuration requires a command line interface. An estimate [113] from Cisco portrays that billion of devices will be

Protocol	Transport	QoS mode 1	QoS mode 2	QoS mode 3	Other Guarantees
DPWS [46]	TCP	–	–	–	–
OPC UA [47]	TCP	–	–	–	configurable timeouts;
CoAP [48]	UDP	non-confirmable	confirmable	–	–
REST [49]	TCP	–	–	–	–
XMPP [50]	TCP	–	–	–	–
JMS [51]	TCP	non-persistent (at-most-one)	persistent (only-and-only-once)	–	durable subscribers; priorities
DDS [52]	UDP	best-effort	reliable	–	rich support of QoS
MQTT [53]	TCP	fire and forget	deliver at least once	exactly once	–
AMQP [54]	TCP	at most once	at least once	exactly once;	transactions
SemiSpace [55]	TCP	–	–	–	–
WebSockets [56]	TCP	–	–	–	–

Table 2.5: QoS features of IoT middleware protocols.

connected to the Internet by 2020. Generated data may differ with regard to their rates, volumes, obtained devices, etc. Thus, it is essential to take into account advanced data management technologies. Authors in [114] propose that emerging technologies such as *Software-defined networking* (SDN), can be used to enable flexible management of the network environment.

Recent research efforts [115–117] show that SDN technologies can be utilized to deal with interoperability issues at the middleware layer. Particularly, authors in [115] develop a middleware with a layered IoT SDN controller to manage dynamic and heterogeneous multi-network environments, mainly for MAC layer protocols. Despite the fact that gateways (see subsection 2.2.2) deal with the heterogeneity issue and integrate multiple (MAC layer) sensor gateways with middleware protocols, an IoT application requires possibly multiple gateways to support a number of Things [118]. Such an issue is very challenging. To relate it with the well-known Web technologies, imagine requiring for each Web site a new Web server. Based on the above *gateway problems*, authors in [116] propose a new SDN-based architecture which includes the *SDF-Gateways* ensuring interoperability between different communication protocols. SDN interoperability approaches seem to be very recent and promising, however authors do not provide many technical details regarding the implementation and evaluation of them.

The next section provides an overview of existing techniques for evaluating the performance of mobile systems employing IoT protocols.

2.3 Performance Evaluation of Mobile Systems

IoT devices differ in terms of size (i.e., resource-tiny, resource-constrained and resource-rich devices). Accordingly, middleware protocols support the selection of several protocol overheads for achieving efficient interactions. For instance, a resource-tiny device with low memory and computational power will demultiplex faster the data payload of a message that encapsulates a

lighter header. Hence, the resulting end-to-end latency between Things depends on the applied overhead inside messages. To guarantee specific response times and data delivery success rates between Things, several IoT protocols provide QoS message delivery features. Initially, they inherit different characteristics from the underlying transport mechanisms and subsequently, they support different modes of message delivery. In case they support such QoS features, the common practice is to define their reliability (from the most unreliable to the most reliable) in several QoS modes (usually three). Table 2.5, summarizes these protocols and their characteristics with regard to the QoS features of each protocol.

IoT middleware protocols, such as DPWS [46], OPC UA [47], REST [49] and XMPP [50] and Websockets [56] do not provide any built-in QoS features since they rely on TCP's delivery mechanisms. On the other hand, CoAP [48] transmits messages over the unreliable UDP protocol. It supports two built-in QoS features: "non-confirmable" and "confirmable". The *non-confirmable* feature does not guarantee the delivery of messages, while the *confirmable* feature supports message re-transmissions using ACKs and NACKs. AMQP [54] and MQTT [53] support basically publish/subscribe interactions. AMQP and MQTT rely on TCP's delivery mechanisms and they introduce additional built-in features for the end-to-end (from the publisher to the subscriber) message delivery such as "fire and forget" or "at most once" (QoS mode 1), "at least once" (QoS mode 2) and "exactly once" (QoS mode 3).

It is worth nothing that through the specification of the MQTT API, developers are able to establish end-to-end interactions with a combination of QoS levels for each link. For example, developers can assign to a *publisher-broker* link the *QoS mode 1*, and to a *broker-subscriber* link the *QoS mode 3*. Tools such as RabbitMQ [119] and Kafka [120] are implementations of the above protocols. JMS [51] is one of most successful asynchronous messaging technology available. It defines an API for building messaging systems where a subscriber can be defined as "non-durable" or "durable". DDS [52] provides plenty of QoS parameters that makes the performance configuration a tedious procedure. SemiSpace [55] is a light weight implementation, inspired by the JavaSpaces [58] middleware protocol. Alternative light weight implementations include GigaSpaces [121], Terrastore [122] and Lime [123]. The above protocols do not provide any QoS built-in features and they rely on the transport protocol's delivery mechanisms.

The aforementioned protocol characteristics, especially the QoS features, enable a developer to build efficiently IoT applications. However, selecting the proper IoT protocol is not a trivial procedure. Despite the fact that several QoS modes are provided, a developer requires additional insights, such as the performance evaluation of the specific protocol (e.g., the timeliness and delivery success rates when transmitting messages of 1 KB using CoAP with the "confirmable" QoS mode). Recent efforts in the research community provide several protocol-specific performance evaluations [61, 124–130]. Such protocol-evaluation efforts help the developer to select the key IoT protocol. However, application developers have to consider the application's context as well.

Paper	QoS metrics - under parameters	Evaluation method
Mehmeti et al. (2013,2014) [131–133]; Lee et al. (2010) [134];	WiFi (on-the-spot, delayed) offloading efficiency and delay; - WiFi intermittent availability, reneging rate;	2-D Markov chains, probability distributions, real traces; probability generating functions (PGF); numerical solutions;
Hyytiä et al. (2013) [135], Wu et al. (2014) [136];	MCC offloading efficiency and delay; - WLAN intermittent availability;	M/G/1-FCFS-queue with intermittently available server, probability distributions;
Phung-Duc et al. (2010) [137];	performance metrics; - reneging rate;	quasi-birth-and-death (QBDs) processes, generator matrix, numerical methods;

Table 2.6: Literature survey in queueing theory.

For instance, the end-to-end latency between two metro commuters exchanging traffic related information depends on their intermittent connectivity. Generally, IoT devices can be mobile since there is an increasing number of embedded sensors into mobile devices (e.g., smartphones). The publish/subscribe and tuple space communication styles provide a loosely coupled form of interaction and thus, are the most employed ones for the creation of mobile systems.

Accordingly, building an application (or system) may require more than one (reliable or unreliable) protocols and applying several timing parameters (e.g., intermittent connectivity). Consequently, investigating generic evaluation techniques of such systems is crucial. We present our survey concerning the recent efforts for the design and evaluation of systems. For each *paper* we provide the *QoS metrics* (e.g., response time) in which the system is evaluated over a number of constraints (e.g., user's intermittent connectivity), and the *method* that has been used to model and evaluate them (e.g., Markov chains). We divide our survey into 3 subsections and for each one we provide a summary table. The 1st is related work relying on queuing theory applied to performance modeling of various systems (Table 2.6), the 2nd is related to the presentation of suitable QoS techniques for evaluating middleware systems (Table 2.7), and the 3rd is about literature regarding the performance of publish/subscribe systems (Table 2.8).

2.3.1 Systems Modeling using Queueing Theory

Concerning the related work in queueing theory, we begin with the works of Mehmeti et al. [131–133]. In these papers, *WiFi offloading* is analyzed extensively by providing performance metrics to improve the efficiency. The authors model the WiFi network availability as an ON/OFF alternating renewal process, which is similar to a mobile user's intermittent availability. Two categories of WiFi offloading are being studied: i) *on-the-spot*; and ii) *delayed* offloading. According to the 1st, when there is WiFi available, all traffic is sent over the WiFi network; otherwise all traffic is sent over the cellular network. On the other hand (delayed offloading), when there is no WiFi availability, (some) traffic can be delayed until WiFi connectivity becomes available. In both cases, an incoming packet during the OFF period can be still transmitted using the cellular connectivity (slower rate); or it can choose to wait until the next WiFi availability. Moreover, a user can choose a deadline to setup (e.g., per application, per file,

etc) concerning the OFF period. If up to that point no AP point is detected, the data are lost or transmitted through the cellular network. Therefore, some packets may be lost or the contract cancelled (renegated). In order to provide performance metrics of the above models, authors investigate a queueing analytical model based on the *2-Dimensional (2-D) Markov chains*. This model uses *probability generating functions (PGF)* to provide closed-form solutions for the mean system time [138]. Authors validate their models using probability distributions for the WiFi availability and real traces concerning the mobility of pedestrian and vehicular users. The proposed model consists of many constraints on probability of states (cellular or WiFi coverage, etc) in the Markov chain.

Authors in [134], also use *2-D Markov chains* to model the WiFi offloading, however they only provide numerical solutions. A similar approach is followed in [135, 136], concerning the offloading strategies in *Mobile Cloud Computing* (MCC). Authors specify the different existing options for task processing in a mobile device: *i)* locally (in the mobile device); *ii)* offload to a Cloud either at a WLAN hotspot or via a cellular network; and iii) being flexible providing both. The different options are modelled as single server queues. Concerning the queue, which offloads data to a Cloud, it is modelled as an *M/G/1-FCFS-queue with intermittently available server* due to the fact that the availability of WLAN hotspots is intermittent. Authors validate their models using probability distributions for the availability of WLAN hotspots.

Finally, a discipline within the mathematical theory of probability, the *quasi-birth-death (QBD) process*, describes a generalisation of the birth-death process. In general, a birth-and-death process is a Markov chain, where transitions are allowed only to the neighboring states. The birth-death process moves up and down between levels one at a time, but the time between these transitions has a more complicated distribution encoded in the blocks. Using this approach we are able to express the mobile user's intermittent connectivity as a QBD process [137] and derive several performance metrics. However, providing solutions by following this approach will result to high computational cost since the process is solved with numerical methods (using its generator matrix) [139].

2.3.2 Formal Analysis and Evaluation Techniques of Middleware Systems

Based on the previous subsection, expressing the intermittent WiFi availability for a mobile user using 2-D Markov chains, is a complex and tedious procedure. Extending this approach for expressing middleware systems, such as publish/subscribe, is even more complicated (see subsection 2.3.3). Long ago, existing investigated approaches express any finite state Markov chain [140]. *Queueing Network (QNs)* and *Performance Petri Nets (PPNs)* are both 'high level' flexible techniques for describing (primarily Markov) models which can be used for constructing performance metrics about computer systems and subsequently, middleware systems. The

Paper	QoS metrics - under parameters	Evaluation method
Vernon et al. (1986) [140];	performance metrics; - parallel systems, deadlock;	Queueing Networks (QNs), Performance Petri nets (PPNs), Extended Queueing Networks (EQNs);
Aldred et al. (2005) [141]; Kattepur and Nambiar (2015) [142];	coupling; - space, time, synchronisation; response time, throughput; - varying service demands;	colored petri-nets; queueing networks, MVA, closed-form solutions;
Basu et al. (2010) [143]; Waszniowski et al. (2009) [144]; Zhou et al. (2016) [145] Kim et al. (2007) [146]	failures; fault tolerant; - safety, bounded liveness; delivery success rate; - time; end-to-end timing/QoS; - packet loss rate, delay, speed;	(hierarchical) timed automata; (statistical) model checking; statistical analysis
Aziz (2016) [147];	QoS levels; subscriber semantics;	timed process algebra (TPi); static analysis;

Table 2.7: Literature survey for middleware systems.

notation used to describe the models, enables the user to develop and explore a large design space rapidly. Along the dimensions of *expressive power* and *solution efficiency*, PPNs enjoy an advantage over QNs in representing synchronization (parallel systems) and are probably best suited for design purposes. A closely related work is [141], where formal analysis (using colored Petri-Nets) of various types of time synchronization in distributed middleware architectures has been performed. On the other hand QNs provide convenient primitives for constructing models, guarantee that are well-formed (i.e., stable, deadlock-free, etc), and can be solved efficiently. Work done by Kattepur and Nambiar [142] makes use of QNs to estimate performance of Web applications using algorithms such as *Mean Value Analysis* (MVA). QNs have been utilized to model the performance of publish/subscribe systems (see subsection 2.3.3).

Timed automata [148] can be used to model and analyze the timing behavior of computer systems, e.g., real-time systems or protocols. They have been applied to a variety of real time system models to ensure accurate behavior under timed guards. Such models provide the ability for checking both safety and liveness properties and they have been developed and studied over the last years. Model checkers such as Uppaal [149], PRISM [150] and SBIP [151] have been proposed for timed and probabilistic properties of such systems. Timed automata are used in [144] for studying fault tolerant behavior (safety, bounded liveness) in distributed asynchronous real time systems. Furthermore, in [145] is proposed a hierarchical timed automata based approach to model and analyze the dynamic software evolution – both functional (with structural changes) and non-functional (with parameter changes) are considered. Hierarchical timed automaton (HTA) introduces a refinement function to describe the hierarchy relationship between states (e.g., the composite states with several regions).

In [152], the transmission channels of publish/subscribe middlewares are modeled using probabilistic timed automata to verify properties of supported interactions. The same authors do model-checking of publish/subscribe applications using Bogor [153] and the PRISM probabilistic model checker [152]. Finally, a very recent work in [147] demonstrates the necessity of applying formal models to IoT protocols. Particularly, authors model MQTT based on a timed message-passing process algebra. The analysis reveals that the protocol behaves correctly regarding the

Paper	QoS metrics - under parameters	Evaluation method
Pongthawornkamol et al. (2007,2010,2011) [154–156]; Kassa et al. (2011) [157];	event probability, end-to-end delay, subscriber's reliability; message reliability; - best effort networks, event lifetime, hand-off; transmission range, movement area dimensions, number of servers, message lifetime;	probabilistic QoS modeling, closed-form solutions, probability distributions; M/M/1, M/G/1, real traces; testbed;
Gaddah et al. (2008,2010) [158, 159];	message loss, message duplication, end-to-end latency, throughput; - hand-off;	mobility models, pro-active caching approach, continuous-time Markov chains (CTMC), generator matrix, numerical methods, probability distributions, testbed;
Kounev et al. (2008) [160]; Mühl et al. (2009) [161]; Martinec et al. (2014) [162]; Sachs et al. (2013) [163]; Singh et al. (2015) [164];	workload characterization, latency; hierarchical routing; latency, reliability; - distributed event-based systems; subscription lifetimes; traffic jams; bursty workloads;	Queueing Petri Nets (QPNs); Stochastic Analysis, testbed; Performance Evaluation Process Algebra (PEPA);
Setty et al. (2013,2014,2015) [165–167];	metrics for satisfaction requirements; - number of events, limited resources;	B3M, F-B3M and MCSS problems, workload analysis, real-world traces;

Table 2.8: Literature survey for publish/subscribe systems.

semantics of the QoS mode 1 and 2. However, the 3rd QoS mode is prone to error and at best ambiguous in certain aspects of its specification.

Alternatives to simulation based approaches, such as statistical model checking [143], may be applied in order to verify, for instance, probabilistic reachability properties. However, simulation techniques are needed as a starting point, in order to elicit distributions needed as inputs to statistical model checkers. This is the case in [143], where authors perform simulations of the system in order to learn the application context. This creates a stochastic abstraction for the application, which is verified using statistical model checking. In the work done by Kim et al [146], a formal specification is developed for each layer of a distributed system. To achieve the desired end-to-end timing/QoS properties, the formal specification is analyzed using statistical model checking and statistical quantitative analysis under various resource management policies.

2.3.3 Performance Evaluation of Publish/Subscribe Systems

Regarding the performance evaluation of publish/subscribe systems, we begin with the work of Pongthawornkamol et al [154–156]. Analytical models are provided in order to predict delivery probability and timeliness for content-based publish/subscribe systems. These models abstract the expressiveness of such systems under unreliable, best effort public networks. In this study the authors apply *lifetime (or deadline) periods* for each published event and the *intermittent availability* of each subscriber in order to estimate the *subscriber's reliability*. They also assume a specific network topology with a fixed number of brokers. To derive analytical models, they focus on the routing of the events into the fixed topology and they apply techniques from probability and queueing theory. More specifically, to estimate the subscriber's reliability they use the *end-to-end delay* between publisher and a subscriber based on *M/M/1* or *G/G/1* queueing models and they compare it with the event's lifetime. The broker's event processing time depends

on the array of the existing subscriptions, and the subscriber is defined as *disconnected* only during the hand-off between two brokers. In this way, the authors analyze the overall delay of events of publish/subscribe systems concerning the network layer (by considering routing of events and subscriber's hand-off). Probabilistic and real-world event traces used to validate the algorithms' accuracy and effectiveness. Regarding real-world event traces, a real-time stock market quote service has been used, where each publisher publishes real-time quotes of a stock to subscribers that are interested in that stock. NASDAQ stock quote event trace is obtained from Google Finance between the 4th and 5th December of 2009. The trace consists of 258,853 events from 2,792 stocks on the first day, and 272,974 events from 2,832 stocks on the second day. Furthermore, in [157] the authors extend the above work by providing closed form expressions of reliability as a function of the number of brokers, area dimensions and deadline parameters.

Subsequently, Gaddah et al. in [158, 159], focus on the users' mobility inside publish/subscribe systems for investigating a pro-active caching approach. Based on this work, to design new hand-off management solutions they consider a fixed network topology where transfer/caching of events/subscriptions between brokers occur prior to subscribers' movement. To evaluate this approach, it is necessary to simulate the network topology and estimate several performance metrics (throughput, in this work), in order to compare them with other approaches. Authors represent the subscriber's mobility with connections and disconnections for randomly generated exponentially distributed times. However, publishing an event during subscriber's disconnection (OFF period) is considered as lost and is not waiting to the broker until the subscriber's reconnection. To evaluate the above approach, they create a testbed using the JMS middleware and they perform experiments in order to compare it with other caching approaches. The subscriber's connectivity is represented using probability distributions. Finally, they utilize *continuous-time Markov chains (CTMC)* to express the subscriber's mobility and obtain the expected number of subscribers depending on the state (connected, disconnected, hand-off) for each broker. Performance metrics are derived through numerical methods whose solution demands high computational cost, as already mentioned above.

In [160], a methodology for workload characterization and performance modeling of distributed publish/subscribe systems is presented. In this study, authors use *Queuing Petri Nets* for accurate performance prediction. While this technique is applicable to a wide range of systems, it relies on monitoring data obtained from the system and it is therefore only applicable if the system is available for testing. Furthermore, for systems of realistic size and complexity, QPNs would not be analytically tractable. Mühl et al. [161] present an approach to stochastic analysis of publish/subscribe systems employing identity-based hierarchical routing. This paper, only considers routing table sizes and message rates as metrics. Moreover, in [164], authors study the tradeoffs between performance and QoS in publish/subscribe systems. Performance evaluation process algebra (PERA) language is used to express the systems. Finally, authors of

the three above efforts, try to tackle the basic functionalities of publish/subscribe systems and they do not consider subscribers' mobility.

To allocate resources (i.e., minimum amount of resources needed, effective way to allocate and cost of hosting them) for a large-scale publish/subscribe system it is critical to get insights from the workload it drives and maximize the overall quality of services given to the subscribers. In [165–167], authors analyze the traces from a real deployment of *Spotify* and *Twitter*, collected via public APIs. The analysis provides several interesting observations which can benefit publish/subscribe system designers. The *Spotify* traces consists of about 1.1 million topics and 4.9 million subscribers forming about 12 million topic-subscriber pairs. The traces were gathered for 10 days (from 9th Jan 2013 to 19th Jan 2013) from *Spotify's* datacenter in Stockholm. Twitter traces provided around 8 million active users, 30 million subscribers, and around 683.5 million topic-subscriber pairs. This data was gathered for 10 days (from 30th Oct 2013 to 9th Nov 2013).

2.4 Summary

In this chapter we introduced the general context of the IoT interoperability issue. In particular, we focused at the middleware layer, providing the most recent efforts (such as SOA/Gateway/-Cloud/MDD/SDN) for solving the interoperability issue. To support the limited resources of tiny/constrained devices, middleware IoT protocols support several QoS features. The latter, in combination to the Things' mobility, require general evaluation techniques aiming to enable system designers to build efficiently their applications.

Accordingly, we have presented the most recent efforts for the design and evaluation of systems. Formal analysis techniques of middleware systems can provide several properties for system tuning. *Timed automata* and *Petri Nets* are some of the techniques applied to offer such properties. Additionally, *Queueing Network Models*, *Queueing Petri Nets*, *Performance Evaluation Process Algebra*, *Markov chains*, etc, provide the ability to evaluate the performance of a system for several QoS metrics (e.g., latency, reliability, throughput, etc) under multiple constraints (e.g., hand-off, best effort networks, traffic jams, etc). Finally, we presented the above techniques applied to publish/subscribe, which is an appealing communication style for mobile IoT applications.

This thesis deals with the heterogeneity and performance issues in the IoT by leveraging some of the techniques presented in this chapter. Below, we provide a brief summary describing the techniques used at each chapter of this thesis.

Middleware protocol interoperability. Access is essential for any IoT deployment, whether there is direct communication among Things or through the Cloud. In traditional SOA, standardization has been particularly effective, with SOAP and REST Web Services being the two dominant technologies. Regarding the same aspect in the IoT, i.e., public service de-

scription and middleware-level service access, there is much bigger diversity. Chapter 3 models the functional semantics of Things employing middleware IoT protocols such as CoAP, MQTT, DPWS, REST, SemiSpace and Websockets. By relying on these models we then introduce our middleware protocol interoperability solution.

End-to-end timed protocol analysis. In Chapter 4, we provide the verification of the timing behavior of multiple heterogeneous interactions using *Timed Automata*. Particularly, we rely on our interoperability solution which defines end-to-end interactions between heterogeneous Things. Then, we model the fine-grained effect of timing thresholds on both coupled and decoupled distributed systems. By leveraging the analysis of timing thresholds, designers of heterogeneous IoT applications can accurately tune parameters to ensure high success rates for interactions.

End-to-end performance evaluation. In Chapter 5, we utilize QNMs to evaluate the performance of heterogeneous interactions. By relying on the models introduced in Chapter 3 we introduce *performance modeling patterns* (PerfMP) for both unreliable/reliable middleware heterogeneous interactions. By leveraging our PerfMPs, developers have the flexibility to design their systems with the evaluation capability of these models. Moreover, they can use our models to estimate end-to-end response times by taking into account timing parameters such as, the intermittent connectivity of mobile users, the lifetime of messages, etc.

Chapter 3

Interconnecting Heterogeneous Systems in the Mobile IoT

Contents

The (mobile) IoT comprises sensors and actuators that are heterogeneous with different operating (e.g., operating platforms) and hardware (e.g., sensor chip types) characteristics, hosted on diverse Things (e.g., mobile phones, vehicles, clothing, etc.). To support the deployment of such devices, major tech industry actors have introduced their own middleware APIs and protocols, which deal with: *i)* the limited hardware (e.g., energy, memory) and network resources (e.g., low bandwidth); and *ii)* loosely coupled interactions in terms of time and space. The resulting APIs and protocols are highly heterogeneous. In particular, protocols differ significantly in

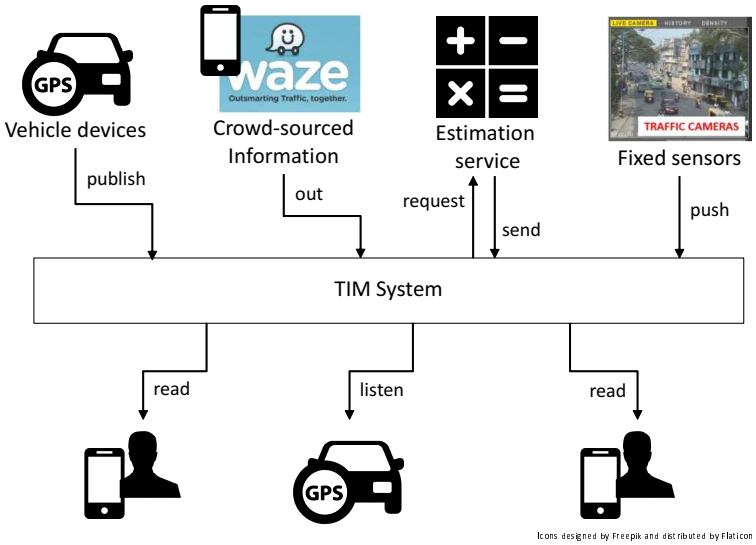


Figure 3.1: Transport Information Management (TIM) system.

terms of communication styles and data formats. For instance, protocols such as CoAP relying on CS-based interactions, MQTT based on the PS communication style, Websockets based on DS interactions, or SemiSpace offering a lightweight shared memory (TS), are among the most widely employed ones. In the following, we outline a representative application scenario, that needs to be implemented by integrating multiple IoT protocols.

The detection and management of traffic congestion in a city is a critical issue in order to avoid significant delays while driving a vehicle [168]. For this purpose, several intelligent systems have been developed. We can classify them into three categories leveraging: *i)* `fixed-sensors` (vehicle detectors, traffic cameras, doppler radars, etc) that have been installed on existing infrastructure [169, 170]; *ii)* `vehicle` (on-board) `devices` with GPS-based systems [171]; and, *iii)* `smartphones` with embedded sensors (accelerometer, gyroscope) [172]. The combination of such intelligent systems can provide us an overall *Transport Information Management* (TIM) system (depicted in Fig. 3.1) in order to estimate accurately traffic conditions. However, depending on the available system resources, each of the above sensors/applications employ a different IoT middleware protocol to exchange data efficiently. Each one of these protocols, implements different APIs and primitives (e.g., `push`, `out`, as depicted in Fig. 3.1) for sending/receiving data of different formats. In particular, the Websockets [56] protocol is deployed on fixed city-deployed sensors to enable the collection of data streams by an `estimation-service` that employs the REST [49] protocol. Data from vehicle-devices (deployed as MQTT [53] peers) are sent periodically to a broker and then to the estimation service. Similarly, users' smartphones implement the SemiSpace [55] protocol to transmit the data sensed to the estimation service through several shared data spaces. Finally, the REST estimation service processes the collected data and provides back the estimated traffic to the end-users (smartphones, vehicle end-users). To enable

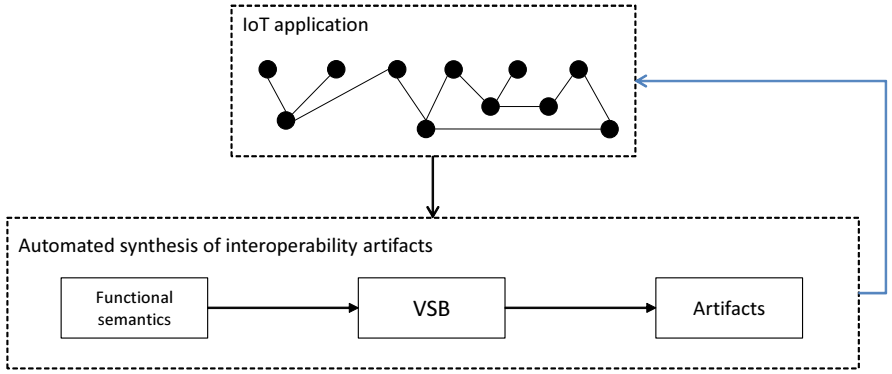


Figure 3.2: Platform for ensuring functional interoperability inside an IoT application.

such an IoT scenario, the heterogeneity between the involved peers (e.g., Websockets → REST) must be tackled.

In this chapter, we introduce the *Generic Middleware* (GM) API which supports the abstraction of functional semantics (space and synchronization) of middleware IoT protocols (e.g., REST, CoAP, MQTT, WebSockets, etc). To demonstrate how GM can represent any middleware protocol that follows one of the identified communication styles (i.e., CS, PS, DS and TS), we introduce an API model for each communication style that implements the most common functional semantics of existing middleware IoT protocols. Subsequently, we devise the *GM connector model* that comprehensively abstracts and represents the semantics of various middleware protocols that follow the four core API models.

By relying on the GM connector model we introduce our middleware protocol interoperability solution which is implemented within the *eVolution Service Bus* (VSB). VSB follows the (ESB) paradigm [76]. In this paradigm, a common intermediate bus protocol is used to facilitate interconnection between multiple peers employing heterogeneous protocols. In VSB we abstract its supported middleware protocols using the GM API. By relying on *model-driven development* techniques and the GM API, we also elicit a generic interface description (GIDL) that can be used to describe the Thing's concrete interactions in GM terms. Then, by relying on GIDL and the GM connector model, we are able to synthesize software *artifacts* (i.e., Binding Components, BCs) for connecting heterogeneous Things to the bus protocol.

With regard to the overall contribution of this thesis (see Fig. 1.1), the contribution of this chapter is positioned as depicted in Fig. 3.2. The rest of this chapter is structured as follows. In Section 3.1, we introduce our core models for each communication style (CS, PS, DS and TS) which abstract the majority of the existing middleware protocols. Section 3.2, presents our GM connector model that abstracts and represents the semantics and primitives of the above core models. In Section 3.3, we present our middleware protocol interoperability solution through the VSB framework. Then, in Section 3.4, we discuss the results of the VSB evaluation. We finally complement this chapter with a brief discussion in Section 3.5.

3.1 Models for Core Communication Styles

This section identifies the four main *communication styles* used in distributed systems (i.e., CS, PS, DS and TS), and defines their corresponding models. The proposed models are the outcome of an extensive survey of these styles as well as of related middleware platforms in the literature. Typically, middleware protocols provide an API to application developers. Each protocol provides several characteristics (supported interactions, QoS guarantees, etc) and can be classified under a communication style. In particular, for each communication style we provide its model by specifying: *i)* its *semantics*, which express the different dimensions of coupling among communicating peers and the supported interaction types; *ii)* its API (*Application Programming Interface*), which is a set of *primitives* expressed as functions supported by the middleware; and *ii) sequence diagrams* that show the detailed interactions between the peers.

By relying on [5, 7, 141], semantics of interest include *space coupling*, *time coupling*, *concurrency* and *synchronization coupling*. Space coupling determines how peers identify each other and, consequently, how interaction elements (such as messages) are routed from one peer to the other. Time coupling essentially determines if peers need to be present and available at the same time for an interaction or if, alternatively, the interaction can take place in phases occurring at different times. Concurrency characterizes the exclusive or shared access semantics of the virtual channel established between interacting peers. Finally, synchronization coupling determines whether the initiator of an end-to-end interaction blocks or not until the interaction is complete; in the former case, the interaction is executed in a synchronous way between the interacting peers. To express synchronization semantics, but also other semantics of end-to-end interactions, we define four interaction types and six role types for the interacting peers:

- one-way interaction: each peer can take either the *sender* or the *receiver* role. The sender sends a piece of data without waiting for a response; the receiver will asynchronously get notified for the arrival of the element by setting a listening & callback mechanism.
- two-way asynchronous (async) interaction: each peer can take either the *client* or the *server* role. Clients initiate a request to a server and then continue their processing (non-blocking). The server handles the client's request using a callback and returns the response at some later point, at which time the client receives the response (also with a callback) and proceeds with its processing.
- two-way synchronous (sync) interaction: each peer can take the *client* or *server* role. A synchronous interaction is blocking for the client and requires a prompt response from the server. Clients invoke a request on the server and then suspend their processing while they wait for a response for a specific `timeout` period.
- two-way stream interaction: each peer can take either the *consumer* or the *producer* role. The consumer requests to establish a dedicated session with the producer. Once estab-

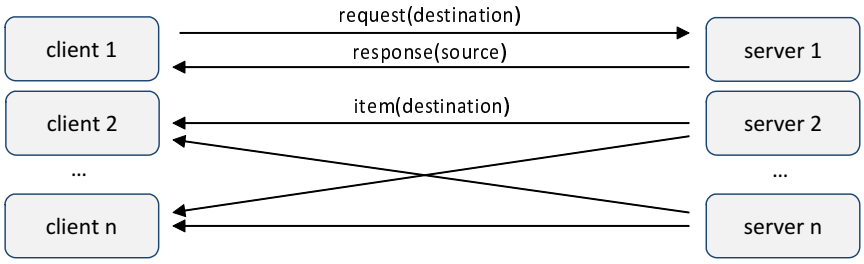


Figure 3.3: CS semantics.

lished, the producer sends multiple pieces of data that will asynchronously be received by the consumer. Depending on the middleware protocol, both peers or just the consumer can suspend, resume and terminate the session using the corresponding interaction elements.

For specifying model APIs, we use a pseudo C syntax with the following conventions: *i)* functions have no return value; they only have `I` and `O` parameters; *ii)* we identify only the parameter names but not their types; and *iii)* the pointer (`*`) represents a callback function or an output parameter. The objective for each one of these APIs is to be able to represent the supported interactions of a wide-range of middleware IoT protocols that follow the corresponding communication style. Finally, the provided sequence diagrams show the peer's interactions and the specific order for each interaction type.

3.1.1 Client/Server Model

The *Client-Server* communication style, is commonly used for Web Services. Besides Web Services, middleware protocols such as CoAP [48], XMPP [50], OPC UA [47], etc, follow the CS style. A client communicates directly with a server either by direct messaging (push notifications) or by a remote procedure call (RPC) through an `operation`. In the first case, a single `item` (which encloses data) is sent from the sending entity (`server`) to the receiving entity (`client`), while, in the second case, an exchange takes place between the two entities with a `request` message followed by a `response`; both cases are depicted in Fig. 3.3.

CS semantics. In terms of space coupling semantics between the two interacting entities, CS requires that the sending entity (`source`) must know the receiving entity (`destination`) and hold a reference of it. Thus, CS represents tight space coupling. With respect to time coupling semantics, both entities must be connected at the same time of the interaction for immediate data transmission. With respect to concurrency semantics, a dedicated virtual channel is used between a sender and a receiver. Items sent by different servers will be received (or not) by the designated clients, based on the offered QoS guarantees of the underlying infrastructure. More details regarding these QoS guarantees can be found in Chapter 5. Regarding synchronization semantics, CS supports one-way, two-way asynchronous and two-way synchronous interactions.

CS API. The above semantics are supported by the CS API primitives and their parameters

Interaction	Role	CS Primitives
one-way	Server	**send**(destination, operation, item, lifetime)
	Client	**receive**(operation, *on_receive()) **on_receive**(source, item)
two-way async	Client	**request**(destination, operation, req_item, lifetime, *on_receive()) **on_receive**(resp_item)
	Server	**receive**(operation, *on_receive()) **on_receive**(source, req_item) **send**(source, operation, resp_item, lifetime)
two-way sync	Client	**request**(destination, operation, req_item, *resp_item, timeout)
	Server	**receive**(operation, *on_receive()) **on_receive**(source, req_item) { **send**(source, operation, resp_item) }

Table 3.1: CS model API.

listed in Table 3.1. The `lifetime` parameter characterizes the item/request validity in time for asynchronous interactions. This parameter is optional; it applies, for example, in cases where IoT data become obsolete after some time and thus need to be delivered before expiration. The `timeout` parameter characterizes the maximum time interval in which the two-way synchronous interaction must be completed. We detail next the CS API primitives:

send: executes the emission of a `item`. For its parameters, it embeds the `destination`/`source` address, the corresponding `operation` name and the related `item`.

request: executes the emission of a `request` to implement two-way interactions. For asynchronous interactions, it sets the `*on_receive()` callback for receiving the response. For synchronous interactions, it blocks until it receives the response; this should be done within a `timeout` period.

receive: sets the reception of one-way items or two-way requests using the `*on_receive()` callback.

on_receive: it is executed upon the reception of one-way items or two-way requests or two-way asynchronous responses. After receiving a two-way request, it executes a `send`, either synchronously or asynchronously.

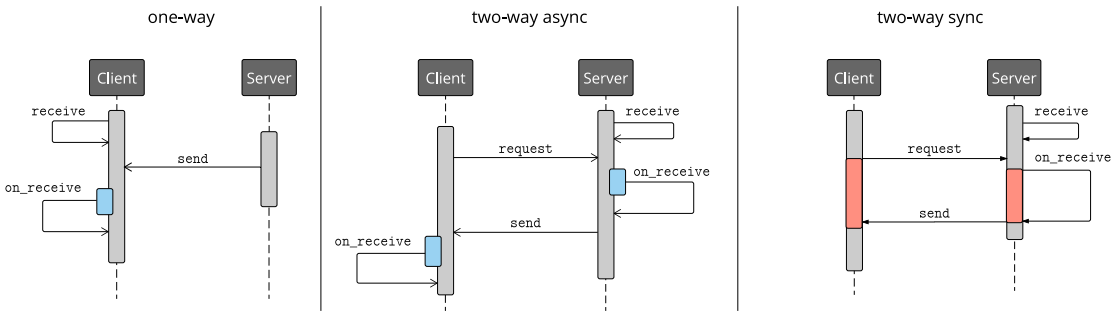


Figure 3.4: CS sequence diagram.

CS sequence diagrams. In Fig. 3.4 we provide the CS sequence diagrams that represent a more detailed view of the supported interaction types by using the above primitives. Particularly, each supported interaction type is specified as follows:

one-way: the client executes the `receive` primitive to set the `*on_receive()` callback for receiving items from any server. Independently, the server executes the `send` primitive for the transmission of an item. Each item is valid for a lifetime period and it will be received in an asynchronous way (through the `on_receive` primitive).

two-way async: the server executes a `receive` primitive to set the `*on_receive()` callback for receiving requests from any client. Independently, the client executes the `request` primitive to transmit the requested item to the server and at the same time set the `*on_receive()` callback in order to receive the response from the specific server. After the `request` primitive is emitted, the client continues its processing. Each request is valid for a `lifetime` period. On the server side, the `on_receive` primitive is executed, and depending on the server's priorities, the `send` primitive is executed with the replied item (assigned a `lifetime` period). Finally, at the client's side, the replied item is received through the `on_receive` primitive.

two-way sync: similar to async, the server initiates a `receive` primitive to set the `*on_receive()` callback for receiving requests from any client. After the client executes the `request` primitive, it blocks its processing until either the reception of the replied item from the specific server, or the expiration of the `timeout` period. On the server side, upon the reception of the request through the `on_receive` primitive, the server must process it and provide a prompt response to the client through the `send` primitive.

3.1.2 Publish/Subscribe Model

The *Publish-Subscribe* communication style, is commonly used for content broadcasting/feeds. IoT middleware protocols such as MQTT [53] and AMQP [54], as well as tools and technologies such as RabbitMQ [119], Kafka [120] and JMS [51] follow the PS style. In PS, multiple peers interact via an intermediate `broker` entity. Publishers produce `events` characterized by a specific `filter` to the broker. Subscribers `subscribe` their interest for specific filters to the broker, who maintains an up-to-date list of subscriptions. The broker matches received events with subscriptions and delivers a copy of each event to each interested subscriber. There are different types of subscription schemes, such as *topic-based*, *content-based* and *type-based* [5]. In topic-based PS, events are characterized with a topic, and subscribers subscribe to specific topics. In content-based PS, subscribers provide content filters (conditions on specific attributes of events), and receive only the events that satisfy these conditions. Finally, in type-based PS, the event structure is abstracted based on specific types and subscribers receive them based on their type. Regardless of the subscription scheme, we use the generic term `filter`, which represents the subset of events that each peer is interested to publish/receive.

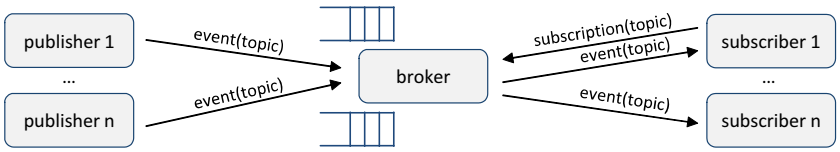


Figure 3.5: PS semantics.

PS semantics. In terms of space coupling semantics between interacting peers, in the PS style, peers do not need to know each other or how many they are. For instance, in the case of topic-based systems, events are diffused to subscribers only based on the topic (see Fig. 3.5). With respect to time coupling semantics, peers do not need to be present at the same time. Subscribers may be disconnected at the time when the events are published to the broker. Upon their re-connection to the broker they will receive the pending events. With respect to concurrency semantics, the broker maintains a dedicated buffer for each subscriber. Hence, unless an event expires, or the PS QoS features do not support event persistence, all events sent by different publishers will be eventually received by interested subscribers. Furthermore, existing PS middleware protocols support several synchronization semantics. Subscribers may choose to check for pending events synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the broker when an event arrives. We focus on the latter case that constitutes the most common practice used in PS style.

Interaction	**Role**	**PS Primitives**
one-way	Publisher	**publish**(broker, filter, event, lifetime)
	Subscriber	**listen**(broker, filter, *on_listen()) **on_listen**(event) **end_listen**(broker, filter)
two-way stream	Subscriber	**subscribe**(broker, filter, lifetime) **listen**(broker, filter, *on_listen()) **on_listen**(event) **end_listen**(broker, filter) **unsubscribe**(broker, filter)
	Broker	**listen**(filter, *on_listen()) **on_listen**(filter) { **...publish**(filter, event, lifetime) }

Table 3.2: PS model API.

PS API. The above semantics are supported by the PS API primitives and their parameters listed in Table 3.2. We represent the notions of topic, content and type with the generic `filter` parameter, which can be a value or an expression. In addition, the `lifetime` parameter stands for the availability of the `event` in time. We detail next the PS API primitives:

subscribe: executes the subscription of a peer to a `broker` for receiving events that are qualified by `filter`.

publish: at the publisher side, it publishes an `event` (to a `broker`) that is semantically qualified by `filter`. At the broker side, it forwards the already published event to the corresponding subscribers (subscribed to `filter`). In both cases, the event is available for the corresponding `lifetime` period.

listen: it is executed at the subscriber side to enable the asynchronous reception of multiple events related to the `filter` applied. Furthermore, it specifies the associated `*on_listen()` callback to handle each event received. At the broker side, it enables the asynchronous reception of subscriptions using the `*on_listen()` callback.

on_listen: it is executed upon the reception of a event at the subscriber side. Additionally, it is used at the broker side to receive a subscription (characterized by a `filter`), update its subscriptions list and enable the execution of multiple `publish` primitives which correspond to a flow of events.

end_listen: closes a session of asynchronous event reception.

unsubscribe: ends a subscription for the specific `filter`.

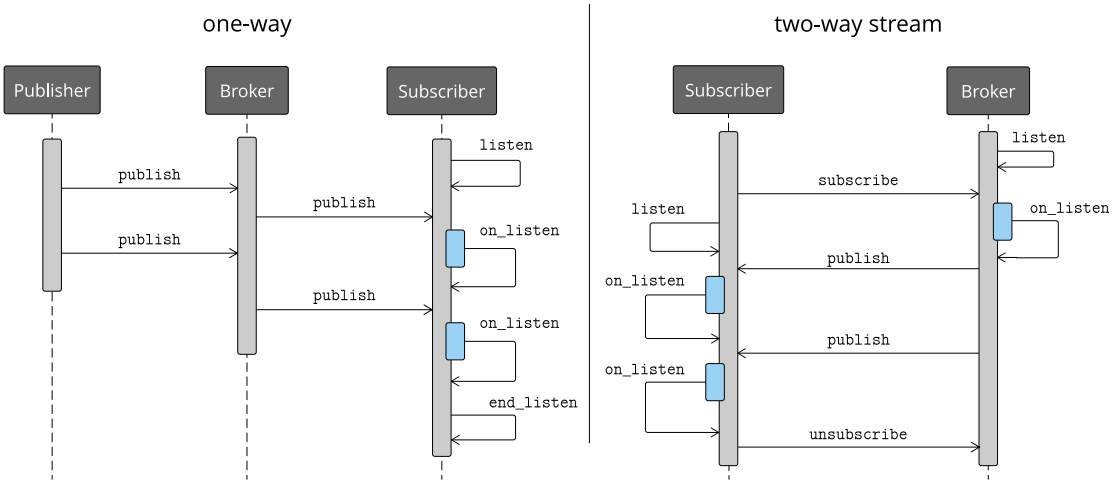


Figure 3.6: PS sequence diagram.

PS sequence diagrams. In Fig. 3.6 we provide the PS sequence diagrams that represent a more detailed view of *one-way* and *two-way stream* interaction types using the above primitives. Particularly, each interaction type is specified as follows:

one-way: to represent such an interaction, we assume that the subscriber is already subscribed to receive events using a specific `filter`. Similarly, the publisher publishes events on the same `filter`. Thus, there is an end-to-end interaction between a publisher and a subscriber through the `broker`. Since the subscriber is already subscribed, the publisher is able to `publish` events at any point in time. As soon as the subscriber executes the `listen` primitive, it connects and receives asynchronously events through the `on_listen` primitive. The subscriber is able to

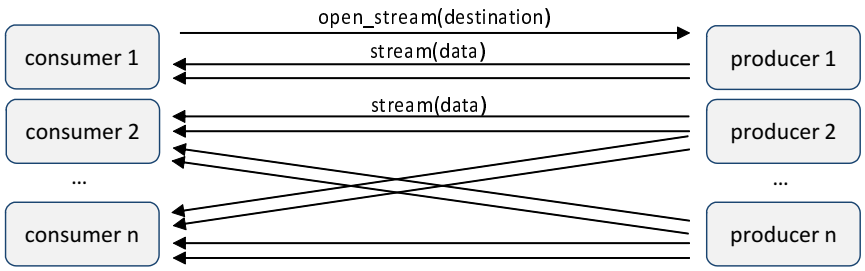


Figure 3.7: DS semantics.

disconnect with the `end_listen` primitive.

two-way stream: for such an interaction, initially the subscriber executes a `subscribe` primitive and afterwards a `listen` primitive which enables its connection to the `broker`. At the broker side, a `listen` primitive is executed to receive subscriptions. In particular, each subscription is received through the `on_listen` primitive which then enables the forwarding of multiple events (coming from multiple publishers) to the corresponding subscriber using the `publish` primitive. Finally, at the subscriber side, each event is received through the `on_listen` primitive until a disconnection (`end_listen` primitive) or a termination (`unsubscribe` primitive).

3.1.3 Data Streaming Model

The *Data Streaming* communication style, is commonly used for continuous interactions. Middleware protocols such as Websockets [56] and Dioptase [173], are based on the DS style. IoT applications (e.g., traffic management, warehouse logistic, etc) produce data coming from the physical world. Such information is produced as a flow of structured data (stream) and thus require continuous handling.

In DS, a consumer (typically) establishes a dedicated session using an `open_stream` request (see Fig. 3.7), which is sent to the producer. Upon the session's establishment, a continuous flow of `data` is pushed from the `producer` to the `consumer`. A `stream` is identified by the pair `<producer, stream_id>`, i.e., the name or address of the producer and a qualifier of the stream unique for the specific producer. Finally, each peer (but most commonly the consumer) is able to `suspend`, `resume` and `close` the stream. Our DS model, represents only the related interaction semantics of streaming protocols and middleware platforms. Other features found in data streaming, such as continuous queries, compression and windowing mechanisms, can be added on top of the stream interaction semantics of the DS model.

DS semantics. Similar to CS, DS represents tight space coupling semantics, with the consumer and producer knowing each other. There is also tight time coupling, with peers availability being crucial for immediate `data` transmission. In terms of concurrency semantics, multiple consumers can receive streams of data from multiple producers over dedicated virtual channels. Depending on the underlying communication infrastructure, data are received successfully (or not), by the

Interaction	Role	DS Primitives
one-way	Producer	**...push**(consumer, stream_id, data, lifetime)
	Consumer	**accept**(producer, stream_id, *on_accept()) **on_accept**(data)
two-way stream	Consumer	**open_stream**(producer, stream_id) **accept**(producer, stream_id, *on_accept()) **on_accept**(data) **suspend_stream**(producer, stream_id) **resume_stream**(producer, stream_id) **close_stream**(producer, stream_id)
	Producer	**open**(producer, stream_id, *on_open()) **on_open**(producer, stream_id) { **...push**(consumer, stream_id, data, lifetime) } **suspend_stream**(stream_id) **resume_stream**(stream_id) **close_stream**(stream_id)

Table 3.3: DS model API.

designated consumers. Regarding synchronization semantics, consumers receive asynchronously each arriving piece of data.

DS API. Our DS model abstracts common semantics widely found in data streaming protocols and related middleware platforms. These semantics are supported by the DS primitives and their parameters listed in Table 3.3. As already pointed out, the pair `<producer, stream_id>` is unique for each stream. The parameters `producer` and `consumer` are the physical addresses of the corresponding peers. Finally, the `lifetime` parameter stands for the availability of each piece of pushed data in time. We detail next the DS API primitives:

open_stream: it is executed by the consumer to request the establishment of a session with the producer.

open: it is executed at the producer side to handle the `open_stream` requests (characterized by the producer's address and the `stream_id`). For each request the *on_open() callback is set up.

on_open: it is executed at the producer side to establish the dedicated session in order to start pushing the `data` flow.

push: it is executed at the producer side for the transmission of a `data` piece semantically qualified by the `stream_id`. This data piece is available for max `lifetime` period.

accept: enables the asynchronous reception of a `data` flow at the consumer side related to the pair of `<producer, stream_id>`. Furthermore, it specifies the associated (`*on_accept()`) callback.

on_accept: it is executed upon the data reception at the consumer side.

suspend_stream: suspends the data flow reception. It can be executed at both the consumer and producer side.

resume_stream: resumes the already suspended data flow reception. It can be executed at both the consumer and producer side.

close_stream: terminates the data flow reception. It can be executed at both the consumer and producer side.

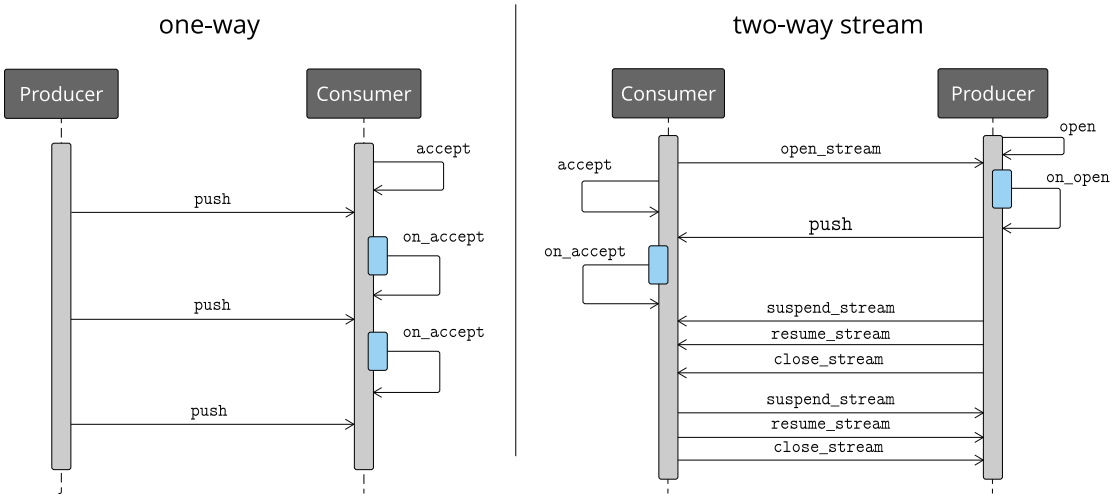


Figure 3.8: DS sequence diagram.

DS sequence diagrams. In Fig. 3.8 we provide the DS sequence diagrams that represent a more detailed view of the supported *one-way* and *two-way stream* interactions using the above primitives. Particularly, each interaction type is specified as follows:

one-way: this interaction assumes that the dedicated session between the consumer and producer is already established. Thus, the producer starts transmitting the `data` flow associated to the corresponding `stream_id` using multiple `push` primitives. On the consumer's side, the `accept` primitive enables the data flow acceptance and sets up the `on_accept` primitive.

two-way stream: to represent such an interaction, initially the consumer executes an `open_stream` primitive to request a stream of data from the consumer. Once the request is accepted, the `accept` primitive is executed to set up the `*on_accept` callback, for receiving the requested stream of data. At the producer side, an `open` primitive is executed to receive requests for the establishment of dedicated stream sessions. Once the dedicated session is established through the `on_open` primitive, the producer transmits the data flow using multiple `push` primitives. Finally, both sides are able to `suspend`, `resume` and terminate (`close`) their session.

3.1.4 Tuple Space Model

The *Tuple Space* communication style, is commonly used for shared data with multiple read-/write peers. Tuple space middleware protocols such as SemiSpace [55], GigaSpaces [121], JavaSpaces [58], etc, are based on the TS style. The definition of our TS model is based on the classic

Figure 3.9: TS semantics.

Interaction	Role	TS Primitives
one-way	Writer	**out**(tspace, template, tuple, lifetime)
	Tspace	**save**(template, *on_save()) **on_save**(tuple)
two-way sync	Reader	**read**(tspace, template, *tuple, timeout)
	Taker	**take**(tspace, template, *tuple, timeout)
	Tspace	**return**(template, *on_return()) **on_return**(reader, template) { **out**(reader, template, tuple) }
		delete(template, *on_delete()) **on_delete**(taker, template) { **out**(taker, template, tuple) }

Table 3.4: TS model API.

tuple space semantics as introduced by the Linda coordination language [123]. In TS, multiple peers interact via an intermediate node with a tuple space (`tspace`, see Fig. 3.9). Peers can write (`out`) data into the `tspace` and can also synchronously retrieve data from it, either by reading (`read`) a copy or removing (`take`) the data. Data take the form of tuples; a `tuple` is an ordered list of typed elements. Data are retrieved by matching based on a tuple `template`, which may define values or expressions for some of the elements.

TS semantics. Similarly to PS, in TS interacting peers write and read/take data from the space (see Fig. 3.9)), independently and with no knowledge of each other. As for time coupling semantics, TS peers can act without any synchronization. In comparison to PS, peers do not need to subscribe for data, they can retrieve data spontaneously and at any time. Nevertheless, the tuple space maintains a tuple until it is removed by some peer or until the tuple expires. With respect to concurrency, peers have access to a single, commonly shared copy of the tuple. Additionally, concurrent access semantics of the tuple space are non-deterministic: among a number of peers trying to access the tuples concurrently, the order is determined arbitrarily. Hence, if a peer that intends to take specific tuples is given access to the space before other peers that are interested in the same tuples, the latter will never access this tuples. This means that not all tuples added to the space by different writers eventually reach all interested readers. In addition to the above semantics, we model synchronous synchronization semantics: readers/takers can receive tuples in a synchronous way (and within a `timeout` period).

TS API. We model the TS model semantics, using the primitives and their parameters listed in Table 3.4. The `lifetime` parameter characterizes the `tuple` availability in time. Furthermore,

the `timeout` parameter characterizes the maximum duration of time in which the reader/taker must receive the requested tuple(s). We detail next the TS API primitives:

out: executes the emission of a `tuple` semantically qualified by a `template` to the `tspace` or to the reader/taker.

on_save: it is executed when a new `tuple` is inserted to the `tspace`. To enable the acceptance of tuples, the `save` primitive must be previously executed.

read/take: executes the synchronous request (`read` for not removal and `take` for removal from the `tspace`) of tuples matched to the `template` aligned.

return/delete: they are triggered at the `tspace` side and handle the incoming read/take requests for tuples matched to a `template`. For each read/take request, the corresponding `*on_read/*on_take` callback is set for providing back the corresponding tuples (using the `out` primitive).

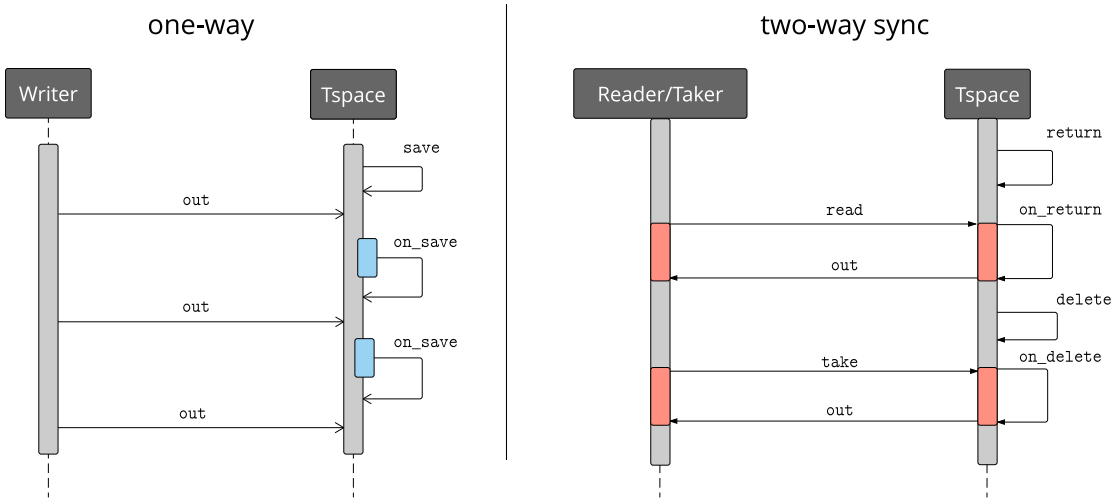


Figure 3.10: TS sequence diagram.

TS sequence diagrams. In Fig. 3.10 we provide the TS sequence diagrams that represent a more detailed view of the supported *one-way* and *two-way sync* interactions using the above primitives. Particularly, each interaction type is specified as follows:

one-way: in our model we do not support asynchronous reception of tuples. Readers and takers access the `tspace` themselves and receive the requested tuples (two-way). Thus, we model TS one-way interactions using only the necessary primitives to store tuples into the `tspace`. Thus, the writer posts tuples that match a specific template using the `out` primitive. At the `tspace` side, the `save` primitive enables the insertion of tuples and sets up the `*on_save()` callback in order to store the incoming tuples.

two-way sync: for such an interaction, a reader/taker executes the corresponding primitive (`read/take`) for requesting tuple(s) matching a specific `template`. At the `tspace` side, the re-

moval (or not, in case of `read`) of tuples can be enabled through the `delete` or `return` primitives. Then, every `read`/`take` request is received through the corresponding `on_return`/`on_delete` primitive which provides back the requested tuples through the `out` primitive.

3.2 Generic Middleware (GM) Connector Model

Given the above four core models (CS, PS, DS and TS), we now introduce the *Generic Middleware (GM) connector model*. As already pointed out, the above models represent the semantics for the majority of existing middleware protocols. Our objective is to devise a generic connector that comprehensively abstracts and represents the semantics of various middleware protocols that follow the four core models. Based on this abstraction, we will introduce later our middleware protocol interoperability solution.

To define the behavioral semantics of our GM connector, we identify two main high-level API primitives: *i)* `post` employed by a peer for sending data to one or more other peers, and *ii)* `get` employed by a peer for receiving data. For example, a PS `publish` primitive can be abstracted by a `post`. We then create a number of variations of these primitives in order to satisfy the various interaction type semantics of our CS, PS, DS and TS models. We identify space coupling semantics for the GM connector by appropriately mapping among the space coupling semantics of the core models. For instance, we define the essential interaction element for GM to be `message`, which can represent any one of CS `item`, PS `event`, DS `data` or TS `tuple`.

Below, we introduce the complete API for GM, comprising a set of primitives to be (abstractly) employed by application-level Things running on top of diverse middleware protocols abstracted by GM.

3.2.1 Generic Middleware API

Similarly to Section 3.1, our GM API is defined using a C-like syntax. For each one of the interaction types: one-way, two-way async, two-way sync, and two-way stream, the corresponding API is provided in Tables 3.5, 3.6, 3.7 and 3.8. It also distinguishes between the two roles involved in an interaction type, such as: *sender* and *receiver*, *client* and *server*, *consumer* and *producer* as described in Section 3.1. To demonstrate how GM can represent any middleware protocol that follows one of the identified communication styles (i.e., CS, PS, DS and TS) , we map the API of our core models to the GM API.

GM One-Way

The GM API that supports one-way interactions, is listed in Table 3.5. These represent CS, PS, DS and TS one-way interactions. In particular, peers that play the *sender* role, i.e., CS server, PS publisher, DS producer and TS writer, transmit messages using the primitive `post`. This is mapped to CS `send`, PS `publish`, DS `push` and TS `out` primitives. The `destination`

Interaction	Role	GM Primitives
one-way	Sender	**post**(destination, scope, post_message, lifetime)
	Receiver	**mget**(scope, *on_get()) **on_get**(source, get_message) **end_mget**(scope)
		xmget(source, scope, *on_xget()) **on_xget**(get_message) **end_xmget**(source, scope)

Table 3.5: GM one-way interaction.

Interaction	Role	GM Primitives
two-way async	Client	**post**(destination, scope, post_message, lifetime, *on_xget()) **on_xget**(get_message)
	Server	**mget**(scope, *on_get()) **on_get**(source, get_message) **post**(source, scope, post_message, lifetime)

Table 3.6: GM two-way asynchronous interaction.

parameter corresponds to the physical address of the *receiver* (i.e., client, broker, consumer and tspace). The `scope` parameter is used to unify identification for the specific CS `operation`, PS `filter`, DS `stream_id` and TS `template`. The `message` parameter embeds the corresponding `item`, `event`, `data` or `tuple`. Finally, the `lifetime` parameter is similar to the same parameter of any core model.

At the receiver's side, there are two variations of the `get` primitive to represent the different core models:

mget: executes the reception of multiple messages from multiple peers. In CS, this is mapped to, e.g., a client's `receive` primitive for multiple messages that come asynchronously from multiple clients for a specific `operation`. In PS, it corresponds to, e.g., a broker's `listen` primitive that receives events from multiple publishers. Finally in TS, it corresponds, e.g., to the `save` primitive which stores tuples coming from multiple writers to the tuple space.

xmget: executes the reception of multiple messages from an exclusive source. In DS, this is mapped to, e.g., a consumer's `accept` primitive that accepts multiple data asynchronously from a specific `<producer, stream_id>`. The same applies in the case of a PS subscriber that `listen`s events from a specific `<broker, filter>`.

Each one of the above `get` primitives sets the `*on_get()` callback function that performs the asynchronous reception. Finally, the `end_get` primitive is used to unset this callback function.

GM Two-Way Async

The GM API that supports two-way asynchronous interactions, is listed in Table 3.6. We use the *client/server* roles, since such interactions typically correspond to CS two-way async interactions. In CS, these interactions are executed using the `request`, `receive`, `on_receive` and `send` primitives (see Fig. 3.4). In GM, we map these primitives as follows: *i)* the client executes a request using the `post` primitive; *ii)* upon the emission of the `post` primitive, the

*on_xget() callback is set for receiving the response. on_xget executes the reception of a single message from an exclusive source (server); *iii)* at the server side, mget enables the reception of multiple requests from multiple clients through the *on_get() callback; *iv)* finally, the server receives the request and sends back the reply using the post primitive. It is worth noting that the client's workflow is not blocked after the emission of the post primitive.

GM Two-Way Sync

Interaction	Role	GM Primitives
two-way sync	Client	**post_xtget**(destination, scope, post_message, *get_message, timeout)
	Server	**mget**(scope, *on_get()) **on_get**(source, get_message) { **post**(source, scope, post_message) }

Table 3.7: GM two-way synchronous interaction.

GM two-way synchronous interactions are supported using the API listed in Table 3.7. Unlike two-way async interactions, the client's processing is blocked until the interaction is complete. The primitive post_xtget sends a request to a server and receives a reply from the same server within a timeout period.

With regard to our core models, the presented API supports CS and TS two-way sync interactions. In CS, the request, receive, on_receive and operation primitives and parameters are mapped to the post_xtget, mget, on_get and scope primitives and parameters in GM. In TS, based on the API of Table 3.4, each reader/taker takes the client's role and the tspace the server's role. At the reader/taker side the read primitive corresponds to the post_xtget primitive. At the server side, the return/delete and on_return/on_delete primitives correspond to the mget and on_get() primitives.

GM Two-Way Stream

GM two-way stream interactions are supported using the API listed in Table 3.8. This API can be mainly mapped to PS and DS stream interactions (Tables 3.2, 3.3). Accordingly, at the consumer side, the post primitive includes the OPEN_FLOW and flow_qualifier parameters for representing the PS subscribe and DS open_stream primitives. The flow_qualifier parameter corresponds to the PS filter and DS stream_id parameters. To initiate the callback for receiving the requested stream (or flow) of messages, the xmget primitive is executed which corresponds to PS listen or DS accept. Messages are received using the primitive on_xget which corresponds to PS on_listen and DS on_accept.

At the producer side, open_stream and subscribe requests are handled through the mget primitive which includes the OPEN_FLOW parameter. Then, multiple messages are sent to the consumer with the post primitive that corresponds to PS publish and DS push. It is worth noting that both peers are able to suspend, resume and close the flow through the correspond-

Interaction	Role	GM Primitives
two-way stream	Consumer	**post**(destination, OPEN_FLOW, flow_qualifier, 0) **xmget**(destination, flow_qualifier, *on_xget()) { **on_xget**(get_message) } **end_xmget**(destination, flow_qualifier) **suspend_flow**(destination, flow_qualifier) **resume_flow**(destination, flow_qualifier) **close_flow**(destination, flow_qualifier)

	Producer	**mget**(OPEN_FLOW, *on_get())
		on_get(source, flow_qualifier) {
		  {...**post**(source, flow_qualifier, post_message, lifetime)...}
		  end_mget(flow_qualifier) }
		  suspend_flow(flow_qualifier)
		  resume_flow(flow_qualifier)
		  close_flow(flow_qualifier) }

Table 3.8: GM two-way stream interaction.

ing primitives. While these primitives represent the majority of DS protocols, in PS, only the subscriber can handle the stream through `listen`, `end_listen` and `unsubscribe`.

Table 3.9 summarizes the mapping between GM and CS, PS, DS, TS concerning the main primitives and their parameters.

GM	CS	PS	DS	TS
post	send	publish	push	out
get	receive	listen	accept/open	save/return/delete
scope	operation	filter	stream_id	template
message	item	event	data	tuple

Table 3.9: Primitives of core models mapped to GM primitives.

3.3 eVolution Service Bus (VSB)

In this section, we introduce the *eVolution Service Bus* (VSB). Its objective is to seamlessly interconnect Things that employ heterogeneous interaction protocols at the middleware level, e.g., REST, CoAP, MQTT, WebSockets, etc. VSB follows the ESB paradigm [76]. In this paradigm, a common intermediate bus protocol is used to facilitate interconnection between multiple heterogeneous middleware protocols: instead of implementing all possible conversions between the protocols, we only need to implement the conversion of each protocol to the common bus protocol, thus considerably reducing the development effort. This conversion is done by a component associated to the Thing in question and its middleware, called a *Binding Component (BC)*, as it binds the Thing to the service bus.

Based on the above, in an IoT application every Thing whose middleware protocol is different from the common bus protocol is connected to the common bus protocol through a BC. VSB follows a fully distributed architecture implemented by a number of BCs that interact among

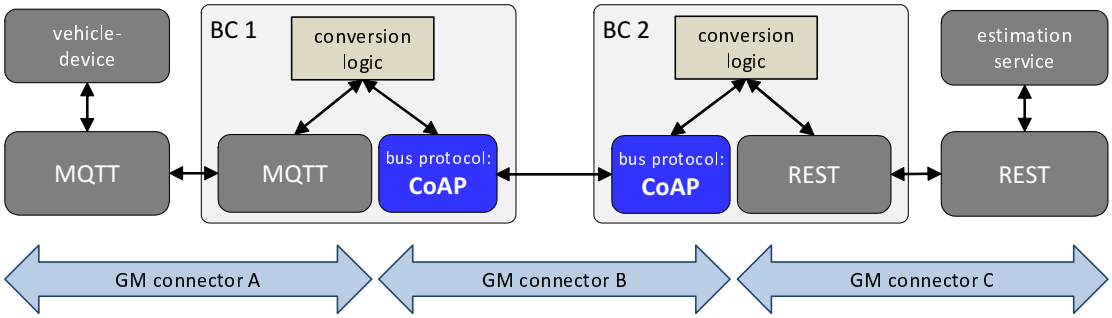


Figure 3.11: VSB end-to-end runtime architecture.

themselves through the VSB common bus protocol. A more detailed view of the VSB architecture is depicted in Fig. 3.11, showing a case of interconnection in the TIM system through the VSB. In this scenario, `vehicle-device` publishes messages through the MQTT middleware protocol and the `estimation-service` receives messages through the REST protocol. BC 1 is associated to `vehicle-device`, while BC 2 is associated to `estimation-service`. We select CoAP to be the VSB common bus protocol. Accordingly, BC 1 & 2 perform bridging between MQTT and REST, respectively, through CoAP.

To enable such a bridging, a BC employs the same (or symmetric, e.g., client vs. server) middleware protocol as its associated Thing (REST/MQTT), and all BCs use a library implementing the bus protocol (CoAP), as shown in Fig. 3.11. Furthermore, a BC contains a conversion logic which maps between the primitives of the bridged protocols. To enable such mapping, we rely on the GM connector model. More specifically, each end-to-end interaction using the same middleware-layer protocol (in our example, REST following the CS communication style, MQTT following PS and CoAP following CS) is modeled and abstracted by the GM connector.

Based on the above architecture, any heterogeneous Thing that employs a middleware protocol associated to one of the CS, PS, DS and TS communication styles, can be connected to the bus protocol. Furthermore, since the common bus protocol is abstracted based on the GM connector, in the same way as any Thing's protocol, different protocols can be introduced as VSB's common bus protocol. Finally, by relying on GM, we are able to introduce an approach for the automated synthesis of BCs. The latter possibility will enable application developers to integrate heterogeneous Things inside IoT applications in a automated manner.

In what follows, we elicit a generic interface description (GIDL) that can be used to describe a Thing's concrete interactions by relying on the GM API.

3.3.1 Generic Interface Description Language (GIDL)

As already pointed out in Chapter 2, SOA enables the interaction of software components in standard ways. Interactions are realized using well known protocols such as SOAP and REST;

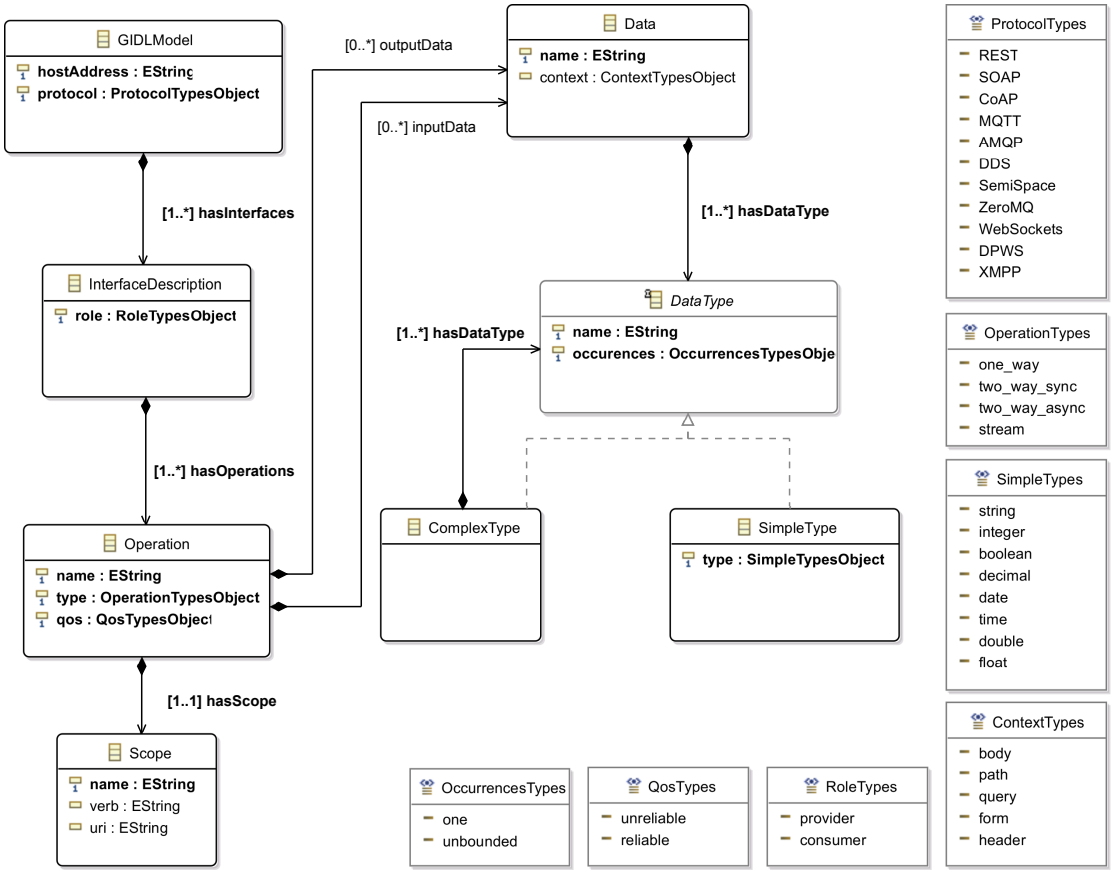


Figure 3.12: The GIDL metamodel.

and each service exposes its functionalities (operations, messages, etc.) by relying on XML-based standard interface descriptions (WSDL/WADL). The existence of standard interface descriptions facilitates the development of frameworks and the wrapping of systems for interoperability. However, with the advent of the IoT, major tech industry actors have introduced their own APIs and protocols to support the deployment of Things. Accordingly, there are very few efforts to specify standard interface descriptions that represent physical objects in the real world (see FI-WARE NGSI Context Management specifications defined by OMA[1]). This lack hampers the interconnection between heterogeneous Things in IoT applications.

As already pointed out, to enable the interconnection of heterogeneous Things, additional software artifacts (i.e., BCs) are required. To facilitate the automated synthesis of such artifacts, we propose a generic interface description which we call *Generic Interface Description Language* (GIDL). A GIDL interface corresponds to a Thing that employs any middleware protocol that can be abstracted into the GM protocol. GIDL enables the definition of operations provided or required by a Thing that follow the interaction types and roles identified in the previous

[1]http://www.openmobilealliance.org/wp

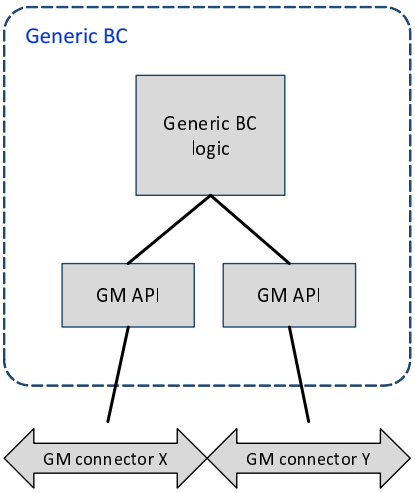


Figure 3.13: Generic BC architecture.

sections. Besides an operation's type, the names and data types of its parameters are also specified. The description is complemented by the physical address of the Thing. To specify GIDL, we create a metamodel using the *Eclipse Modeling Framework* (EMF)[2]. This metamodel allows us to generate code that builds a software artifact (i.e., a BC) for interconnecting the Thing described in GIDL to the bus protocol. Fig. 3.12 shows the GIDL metamodel. More details regarding its attributes can be found in the Appendix B.1. To facilitate the definition of a GIDL model (i.e., the GIDL description of a concrete Thing), we have developed an Eclipse Plugin[3] using the EMF tools. Application developers can follow the procedure described in the Appendix B.2, where we further provide the GIDL models of the TIM system.

In the following, we elaborate a generic architecture for BCs. Such an architecture will allow us to leverage GIDL for synthesizing concrete BCs for various Things.

3.3.2 Generic Binding Component

By relying on the GM abstraction of the protocols bridged by a BC, we design and build the architecture of a BC at an abstract level, which we call a *Generic BC (GBC)*, as shown in Fig. 3.13. A GBC performs bridging between two instances of the GM connector (X and Y in the figure), to each of which it connects through the GM API. The bridging functionality is implemented by the GBC logic, which is a set of primitive-rules of the type:

$$if\ get\ primitive\ received\ on\ GM\ connector\ X(Y),$$
$$then\ execute\ symmetric\ post\ primitive\ on\ GM\ connector\ Y(X)$$

[2]https://eclipse.org/modeling/emf/
[3]http://nexus.disim.univaq.it/content/sites/chorevolution-modeling-notations

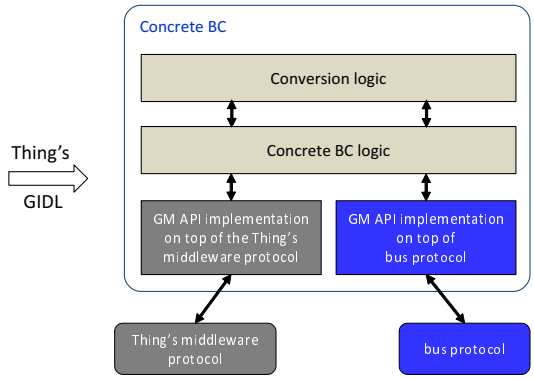


Figure 3.14: Concrete BC for bridging a Thing's middleware protocol to the bus protocol.

The association between `get` and symmetric `post` primitives is based on the GM API and the GM interaction types. In what follows, we leverage the Generic BC architecture and the GIDL metamodel to synthesize BCs for integrating heterogeneous Things inside IoT applications.

3.3.3 Binding Component Synthesis

We present in this section our approach to the automated synthesis of a concrete *Binding Component* for a specific Thing. Development of BCs is a tedious and error-prone process, which can highly benefit from automated systematic support. Furthermore, automated BC synthesis is essential for IoT applications relying on dynamic runtime composition of heterogeneous Things where there is no human intervention. Our solution to BC synthesis consists in customizing a *Generic Binding Component* (GBC) into a concrete BC according to: *i)* the Thing to which the concrete BC is associated, and *ii)* the selected VSB bus protocol, or equivalently, the selected common middleware protocol of the IoT application.

To enable GBC customization, we develop a resource pool, which we call *Protocol Pool*. This pool contains GM API implementations on top of concrete middleware protocols. Each such implementation realizes one or more of the interaction types supported by the concrete middleware protocol with the GM API in a programmatically optimal way, by mapping the concrete middleware protocol's primitives and semantics to primitives and semantics of the GM API. We develop these GM API implementations as generic code excerpts in Java.

To customize the GBC into a concrete BC (see Fig. 3.14), we select from the Protocol Pool the two GM API implementations that correspond to the Thing's middleware protocol and the VSB protocol. By attaching to them the third-party libraries that implement the two middleware protocols, we build two concrete instances of the GM connector. The concrete BC will have to bridge between these two concrete GM instances. For this, the Generic BC logic needs to be customized with the concrete data parameters of the Thing in question, as described in its related GIDL model.

To enable the automated execution of the BC synthesis actions identified above, we introduce

the VSB development framework, which can be leveraged by application developers. We present its architecture in Fig. 3.15. Bellow we provide a brief description of each component.

VSB Manager: the main component of VSB. It exposes an interface which allows the acceptance of requests for synthesis of concrete BCs for specific Things and returns the corresponding BCs. Each request consists of the Thing's GIDL model and the information about the selected bus protocol.

GIDL Parser: is responsible for the parsing of the Thing's GIDL model. Information about the Thing's operations, input/output messages, middleware protocol, etc, are extracted through this component.

GBC Logic: the Generic BC Logic contains a set of primitive-rules. Each primitive-rule is a composition of `get` and symmetric `post` primitives that make part of a GM interaction type. This component returns the *concrete BC logic* for the identified interaction type(s) and the specific Thing.

GM API: except for the defined GM API comprising `post` and `get` actions, this component defines generic methods for message conversion between protocols.

Protocol Pool: this component refines the GM API and implements the supported GM interactions of several concrete middleware protocols. Moreover, concrete methods regarding message conversion between protocols and the BCs' operation (startup/shutdown) are implemented.

BC Synthesizer: based on the GIDL's parsing, this component selects the appropriate primitive-rules and obtains the concrete BC logic via the *GBC Logic* component. Then, it synthesizes the concrete BC using appropriate GM implementations from the *Protocol Pool*.

BC: each BC implements the mapping between the Thing's middleware protocol and the common bus protocol. To allow its configuration (i.e., IP addresses, port numbers, etc) and handling, each BC exposes an interface named *BC Manager*.

As depicted in Fig. 3.15, the VSB manager operates as a service that accepts suitable requests and returns the synthesized BCs. Accordingly, the VSB framework can be used in the following two ways by developers: *i)* an application developer incorporates a *new Thing* into an IoT application; and *ii)* the middleware developer maintaining the VSB framework instance introduces a *new protocol* into the *Protocol Pool*. We detail these two cases in the following.

New Thing

Consider the scenario where an application developer wishes to add the Thing `traffic-light` to the TIM system that employs CoAP as its common middleware protocol. `traffic-light` exposes a REST interface for requesting its light status and thus, a new BC must be synthesized. The BC synthesis process is realized by taking the following steps:

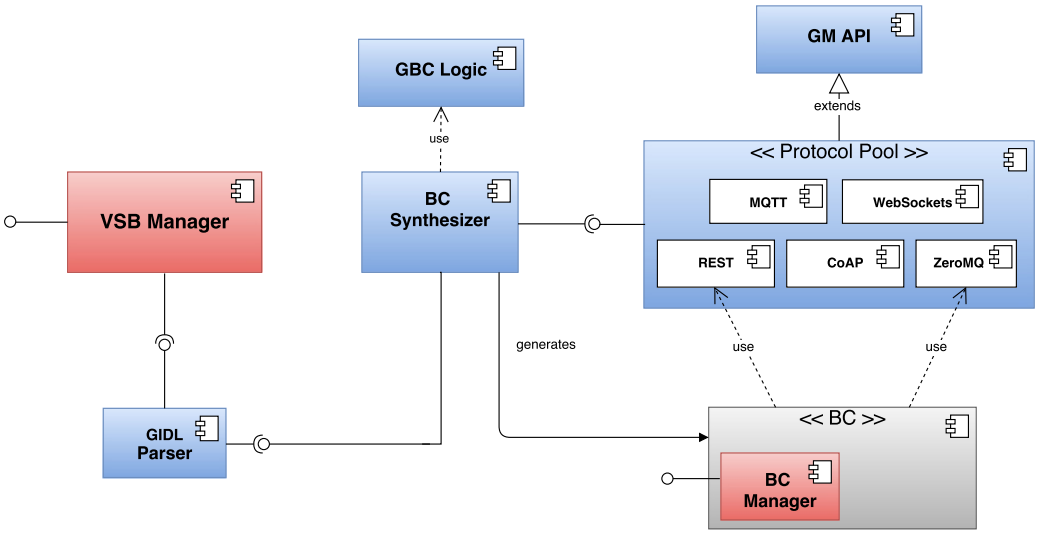


Figure 3.15: VSB development framework architecture.

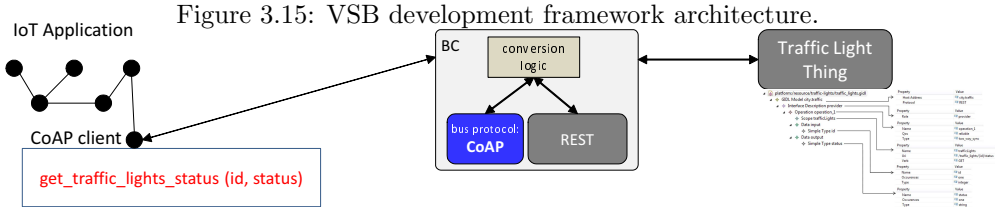


Figure 3.16: REST `traffic-light` interacting with the TIM system.

1. Using our Eclipse plugin, the application developer defines the GIDL model for `traffic-light`. More details regarding the specification of a Thing's GIDL model (including the example of `traffic-light`) can be found in the Appendix B.2.
2. The application developer makes a request to the *VSB Manager* by providing `traffic-light`'s GIDL model and the information about the common protocol of the TIM system (CoAP).
3. If `traffic-light`'s middleware protocol was not included in the *Protocol Pool*, the *VSB Manager* would not be able to synthesize the corresponding BC. In such case, the middleware developer must enrich the *Protocol Pool* with the new middleware protocol.
4. If the protocol is already supported (in our case REST), the *VSB Manager* requests the corresponding BC from the *BC Synthesizer*.
5. The new BC has two main subcomponents (see Fig. 3.16): a REST client (that invokes the REST `traffic-light`) and a CoAP server (that accepts requests from the TIM system). Accordingly, the *BC Synthesizer* synthesizes the BC as follows:
   a. from the information derived from the GIDL parser it identifies the GM interaction type (two way sync), `traffic-light`'s role (server), the supported operations, input/output messages, etc.
   b. it requests the corresponding *concrete BC logic* from the *GBC Logic* component.

More development-oriented information concerning the BC synthesis process for a new Thing can be found in the Appendix B.3.

New protocol

Consider now the scenario where an application developer wishes to add the Thing `smart-bridge` to the TIM system. `smart-bridge` exposes an XMPP interface for providing its status (open/-closed), and thus, a new BC must be synthesized. However, XMPP is not supported by the VSB framework. Hence, the middleware developer must enrich the *Protocol Pool* with this protocol. To add support for a new protocol the developer should follow the steps below:

1. Identify the protocol's primitives with respect to the GM API.
2. Develop a GM API implementation for this protocol.
3. Implement the methods for the deployment (startup, shutdown, etc) of each protocol role (client, server, etc).
4. Incorporate the protocol into the *Protocol Pool*.

3.4 Implementation and Assessment of VSB

VSB has been implemented using `Java 8` and the `Maven` software project management tool. BCs are synthesized using the `JCodeModel` API (code generator). Regarding the *Protocol Pool*, REST has been implemented using the `Restlet` API, MQTT has been implemented using the Paho project[4], CoAP using the Californium framework[5], DPWS using the JMEDS framework[6], JMS using the ActiveMQ messaging server[7], SemiSpace using its framework[8] and WebSockets using its API. The first version of VSB, which is released in: *https:gitlab.ow2.orgchorevolutionevolution-service-bus.git*, which is a prototype version. VSB is utilized as core component of the H2020 CHOReVOLUTION project[9] and enables heterogeneous interactions in IoT choreographies. Currently, VSB supports the following middleware protocols: REST, CoAP, MQTT, WebSockets and DPWS, providing the functionalities mentioned in the previous sections (interconnection through BCs, lightweight BCs, automated generation, etc). Nevertheless, prior to its utilization in real life scenarios, it is essential to evaluate the developers' support and the performance of the synthesized BCs.

More specifically, we evaluate the VSB framework and runtime environment with respect to two criteria: *i)* the support that the VSB framework offers to developers when developing a new IoT application – which may contain a set of heterogeneous Things; and *ii)* the performance of

[4]https://eclipse.org/paho/
[5]http://www.eclipse.org/californium/
[6]http://ws4d.org/jmeds
[7]http://activemq.apache.org/
[8]http://www.semispace.org/
[9]www.chorevolution.eu

KPI name	Development effort for the integration of heterogeneous Things to an IoT application
Metric	Average % of person-hours reduction using VSB
Measurement procedure	1. Define a test case 2. Ask n developers to develop the test case until correct execution 3. Measure the person-hours / time needed with and without VSB and average the results
Test case with VSB	Define a GIDL model for each heterogeneous Thing through the VSB eclipse plugin. (VSB Manager accepts each GIDL file and automatically synthesizes the related artifacts).
Test case without VSB	Develop the adaption and binding logic for each heterogeneous Thing.
VSB Benefits	Less development effort, less effort for bugs fixing and re-works

Table 3.10: BC synthesis Key Performance Indicator (KPI).

the synthesized runtime BCs in terms of response time and throughput, under both *low traffic* and *stress conditions*. To interconnect two heterogeneous Things, developers can leverage VSB to synthesize a BC – otherwise they have to develop their own software artifacts to map between interconnected Things specific primitives and data. Hence, the former evaluation shows the development reduction in person-hours when leveraging VSB. Subsequently, the latter evaluation aims to show that the time overhead introduced by the necessary BCs in the heterogeneous IoT application does not raise a performance issue – hence, it remains reasonable with respect to the overall performance requirements.

For our evaluations, we rely upon the TIM system, which is described in the introduction of this chapter. This scenario prescribes interconnections between Things employing heterogeneous protocols classified to all four communication styles, i.e., CS, PS, DS and TS. The *VSB Manager* is utilized to synthesize BCs that integrate each heterogeneous Thing into the IoT application. We present our evaluation results in the following.

3.4.1 Support to Developers

The BC synthesis process is described in subsection 3.3.3. Based on this process, the *VSB Manager* requires the Thing's GIDL model and the information about the common bus protocol selected for the specific IoT application (CoAP in our scenario). Then, the VSB Manager returns the synthesized BC artifact to be deployed and executed.

The TIM system includes non-CoAP Things. More specifically, `fixed-sensors` employ the WebSocket protocol, the `estimation-service` employs the REST protocol, `vehicle-devices` employ the MQTT protocol and finally, `smartphones` employ the SemiSpace protocol. To incorporate such Things and allow their interconnection with the remaining (CoAP) participants (if any), an application developer is able to use our eclipse plugin, define the GIDL models and synthesize the corresponding BCs. The specified GIDL models of the above heterogeneous services can be found in the Appendix B.2.

heterogeneous Thing	man-hours with VSB	man-hours without VSB	man-hours reduction using VSB (%)
`fixed-sensors`	0.25	3	78.3
`vehicle-devices`	0.25	3	78.3
`smartphones`	0.1	2.5	96
`estimation-service`	0.4	4	90

Table 3.11: Development effort of the application developer.

The above process requires the VSB platform for integrating heterogeneous Things to the TIM system. To evaluate the effectiveness of the provided development process by VSB, we define a *Key Performance Indicator* (KPI). KPIs evaluate the success of an organization or of a particular activity in which it engages. VSB aims to reduce the development effort for IoT applications employing multiple heterogeneous protocols. Accordingly, Table 3.10 presents the KPI used to measure the percentage of the person-hours reduction for synthesizing a software artifact with VSB. By following the KPI measurement procedure of Table 3.10, we define our test cases as follows:

1. Test case with VSB: application developers use VSB Eclipse plugin to integrate heterogeneous Things by specifying their GIDL models, synthesize their BCs and finally deploy them for execution.
2. Test case without VSB: application developers integrate heterogeneous Things by developing software components that adapt different protocols' primitives/data, and finally deploy them for execution.

As a first step, we have asked a software engineer of our team at Inria to perform the above test cases. However, in our future work we intend to run similar test cases with more participants. Table 3.11 summarizes our measurements of the development effort required for our scenario.

Based on the Table 3.11, the VSB framework reduces the application development effort considerably. Particularly, application developers are able to save 78.3% - 96% of person-hours for building software artifacts that interconnect heterogeneous Things in our scenario. The `estimation-service` is the most complex one for defining its GIDL model, since there are multiple operations, as well as input and output parameters to be defined. We note here that a developer does not require to have any special knowledge for defining a GIDL model using our Eclipse plugin. On the other hand, building interoperability components without VSB requires from a developer to be aware of the corresponding APIs and protocols.

3.4.2 End-to-End Performance Evaluation

The VSB runtime (i.e., the synthesized BCs) introduces runtime transformations enabling cross-connection and data conversion among heterogeneous middleware protocols. Hence, we need to evaluate the performance of our solution given the time overhead introduced by such transformations. We evaluate the performance of the VSB runtime with the following experimental setup.

We interconnect heterogeneous Things through a specific bus protocol and measure end-to-end response times and throughput, under both low traffic and stress conditions. At the same time, we measure the introduced latency inside the BCs. Then, we substitute the bus protocol with other middleware protocols aiming to observe trade-offs in the resulting end-to-end performance.

We evaluate the performance of the VSB Binding Components under stress conditions by relying on [174] and [175]. Our approach enables setting a lightweight testing environment around the VSB BCs with practical hardware resources and making sure that BCs employ their maximum capacity. In particular, to evaluate the performance capacity of VSB, we have to saturate each BC system to determine its maximum performance. To this end, Things have to take the role of senders and receivers and need to send and receive high message rate through BCs.

Test Scenario

We set up our test environment with heterogeneous mock Things (senders and receivers) and we synthesize corresponding BCs. We utilize the supported middleware protocols of the VSB Framework (Wesockets, REST, CoAP, MQTT and DPWS). Our purpose is to remove any bottlenecks from the Things (senders/receivers) and create potential bottlenecks in the BCs for testing their maximum performance. Regarding performance, we measure throughput and one-way end-to-end response times. More specifically, we develop a sender (using Websockets) and threads for creating many mock producer applications. Then, the synthesized BCs interconnect the producers with a single receiver and handle the traffic sent through the bus protocol. As bus protocol, we leverage and test REST, CoAP and DPWS middleware protocols. In order to overload the BCs, our receiver must be able to receive thousands of messages per second. The CPU usage of the machines hosting the BCs should be close to 100% to reach the maximum performance of the BCs. On the other hand, it is important that the senders and the receiver are not highly loaded.

Test Setup

We used the following software and hardware for our experiments. The setup consisted of five machines, connected via a local switch (GS900/8, Allied Telesis) creating a private 1000 Mb/s Ethernet local network. The first machine (M1) has as Intel Core i7-3520M CPU 2.90Ghz x 4 (8 GB RAM), the second (M2) an Intel Xeon(R) CPU W3540 2.93GHz x 4 (4 GB RAM), the third (M3) an Intel Core i7-4600U CPU 2.10GHz x 4 (8 GB), the forth (M4) an Intel(R) Xeon(R) CPU W3550 3.07GHz (8GB RAM), and the last machine (M5) has an Intel Core i7-4790 CPU 3.60GHz x 8 (8GB RAM). M5 is used as monitor that collects information related to the exact arrival time of messages at each machine, for estimating the end-to-end response times and throughput. Running tests on powerful machines allows simulating a large number of

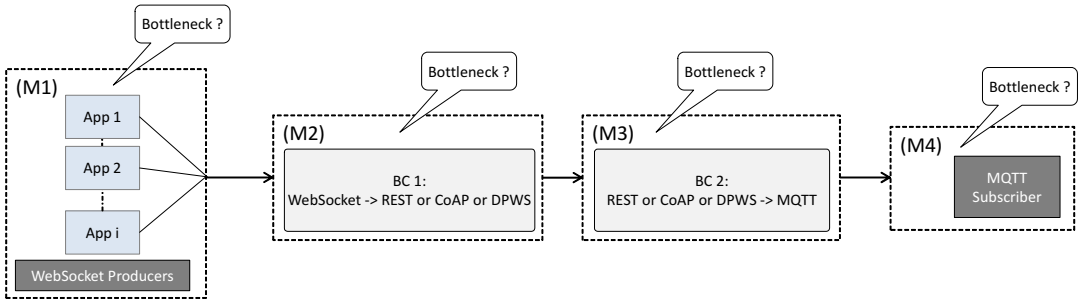


Figure 3.17: Components of the mock environment for the VSB runtime capacity testing.

Scenario (bus protocol)	**BC 1 Latency** (ms)	**BC 2 Latency** (ms)	**End-to-end Response Time** (ms)	**Throughput** (msg/sec)	**Avg. Senders**
Scenario 1 (REST)	0.1	0.1	2.44	299	300
Scenario 2 (CoAP)	0.16	0.11	453	300	300
Scenario 3 (DPWS)	0.16	0.05	1.6	299	300

Table 3.12: Results for one-way interaction in the three scenarios with 300 concurrent senders.

senders more accurately, as opposed to simple core machines.

As depicted in Fig. 3.17, we provide three test scenarios:

1. Producers - BCs (REST) - Subscriber: Using the Websockets middleware we create mock producers running on M1; BCs are synthesized employing the REST protocol running on M2 and M3; and finally we create a subscriber using the MQTT (with the "fire-and-forget" QoS mode [53]) middleware running on M4.
2. Producers - BCs (CoAP) - Subscriber: This is the same scenario as the previous one with only one difference: BCs are synthesized employing the CoAP (with the "non-confirmable" QoS mode [48]) protocol running on M2 and M3;
3. Producers - BCs (DPWS) - Subscriber: This is also the same scenario with BCs employing the DPWS [46] protocol running on M2 and M3;

The first scenario leverages BCs performing a transformation from Websockets to REST and then to MQTT primitives by relying on GM. Then, in the second and third scenario we substitute the REST bus protocol with CoAP and DPWS, respectively. Thus, BCs perform a transformation from WebSockets to CoAP/DPWS and from CoAP/DPWS to MQTT primitives. Note that, in all cases senders produce messages every second. In our measurements, we discard the first 4000 messages allowing machines to reach a steady state. Things generally do not exchange very large quantities of data, so messages usually tend to be of average size. Hence, we set the size of messages to 284 bytes (such message payloads are usually encountered in IoT applications).

Results

Table 3.12 presents some detailed data for a test run with 300 concurrent senders. To run a test, we create and send a sufficient number of messages by running each experiment for at least 2 hours. Accordingly, for 300 concurrent senders we send approximately 2160000 messages and then we calculate the average response times and throughput. In total we have performed 49 tests for REST and DPWS and 25 for CoAP. Showing detailed results for each performed test it would be impractical, as it would require several pages; nevertheless, we plot these tests in the following figures. Based on the Table 3.12, the end-to-end response time is 2.44 ms when employing REST as the bus protocol, 453 ms for CoAP and 1.6 ms for DPWS. In case of CoAP, the throughput is 300 messages per second, which corresponds to the maximum limit of this protocol (based on its RFC[10], it enables up to about 250 messages per second from one endpoint to another with default protocol parameters). Thus, as the incoming load of messages (more then 250 msg/sec) increases, this results to high end-to-end response time where the synthesized BCs have reached their maximum resources (CPU, Memory) since they employ CoAP as the bus protocol. It is worth noting that the latency inside the BCs is negligible (0.05 - 0.16 ms, for this particular message size), with regard to the end-to-end response time.

Fig. 3.18 shows the measured throughput when sending one-way messages to the MQTT subscriber, in function of the number of concurrent senders for each of the above scenarios (employing different bus protocols). The procedure we applied to execute this experiment is the following: *i)* in all cases, after BCs reach the steady state, the MQTT subscriber counts incoming messages for a duration of time; and *ii)* we repeat the same experiment by increasing the WebSocket application senders. Thus, the MQTT subscriber receives an increasing number of messages according to the concurrent senders. We observed that the number of messages passing via BCs per second (throughput) for high input loads depend on the employed bus protocol. In scenario 1, (REST bus protocol), the maximum throughput is 1865 messages per second, in scenario 2 (CoAP bus protocol) is 324 messages per second and in scenario 3 (DPWS bus protocol) is 1801 messages per second. We verified at the same time that the CPU usage of the machine hosting BC 1 had reached its maximum, while the CPU for senders/receivers was about 30% - 50%. To achieve this, we have deployed BCs to less powerful machines (M2, M3), in comparison to the ones that host the senders and the receiver (M1, M4). In this way we are able to: *i)* reach the maximum resources of machines hosting BCs; and *ii)* compare the three bus protocols with the same criteria.

Fig. 3.19 shows the measured one-way response times when sending messages to the MQTT subscriber, in function of the number of concurrent senders for each of the above scenarios. The procedure we applied to execute this experiment is the same as the previous one for the throughput. We observed that for low traffic (up to 200 senders) DPWS presents the lowest end-

[10]https://tools.ietf.org/html/rfc7252

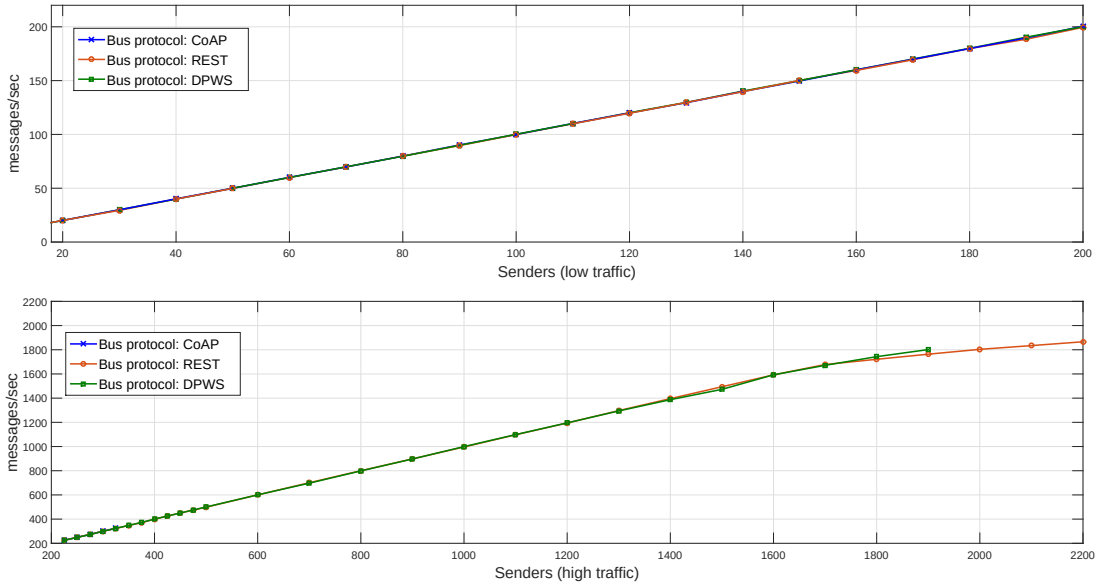


Figure 3.18: Throughput for one-way interactions through REST, CoAP and DPWS bus protocols in scenarios 1, 2 and 3.

to-end response times (∼1.6 ms), while REST presents quite low values (∼2.5 ms). Regarding CoAP, for low traffic it presents similar values in comparison to DPWS; nevertheless, after having 60 senders end-to-end response times it reaches quite high values. For high traffic, DPWS maintains its low response time values (∼1.6 ms) up to 400 senders; while afterwards it presents quite low values (∼10 ms) up to 1300 senders. DPWS is scalable enough, since it presents high values of response times and low values of throughput (see Fig. 3.18) after having 1700 senders. Regarding REST, it presents the lowest end-to-end response times (∼4.5 ms) for high traffic and is much more scalable than both CoAP and DPWS protocols. It is worth noting that after having 1600 senders, REST maintains its end-to-end response times, however its throughput values become lower (see Fig. 3.18) due to a considerable number of losses. Hence, for heavy traffic load REST presents lower response times and higher message losses while DPWS presents higher response times and lower message losses. For all the above cases we verify that the bottleneck is present at the machines hosting BCs.

3.5 Discussion

Integrating Things that employ heterogeneous middleware protocols is challenging. Specifically, an IoT application may integrate sensors, actuators, mobile devices, etc, which interact with each other through push notifications, synchronous/asynchronous and streaming types of interactions. In this chapter, we have introduced models for the core communications styles (CS, PS, DS and TS) including their interaction types, semantics, APIs and sequence diagrams. Based on the basic interaction types we enable cross-protocol interconnection at the middleware layer by

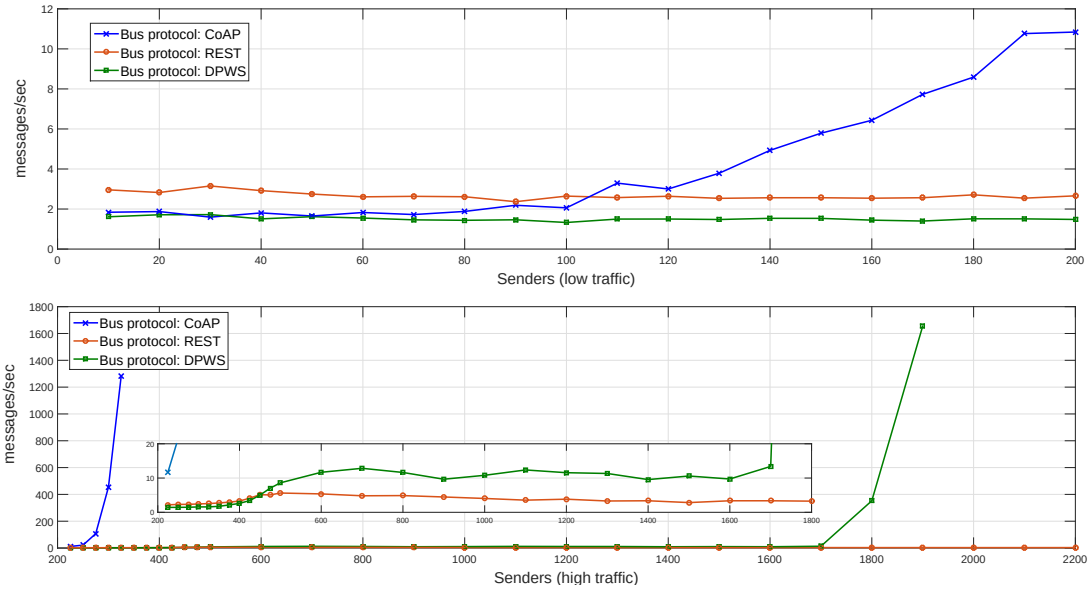


Figure 3.19: Response times for one-way interactions through REST, CoAP and DPWS bus protocols in scenarios 1, 2 and 3.

introducing a *Generic Middleware* (GM) API which can abstract any IoT protocol that employs one or more of these interaction types.

We apply our modeling abstraction as a lightweight and fully distributed ESB, which enables interconnections among Things in a peer-to-peer way. Our development and runtime platform, *eVolution Service Bus (VSB)*, can be leveraged by application developers. Using the platform's GIDL metamodel, they can easily describe their Things and synthesize BCs that enable the interconnection with the VSB's common bus protocol (which corresponds to the IoT application protocol). Different protocols can be introduced as VSB's common bus protocol with the same easiness as for integrating support for a new middleware protocol of a Thing. Additionally, VSB BCs are built and deployed as necessary; hence, no BC is needed when a Thing employs the same middleware protocol as the one used as the VSB protocol.

The evaluation of the VSB framework and runtime demonstrated good results in terms of both developer support and performance. We note that our software engineering support evaluation is based on the person-hours measurement of a single developer. A more comprehensive empirical evaluation would require an subjective evaluation of the framework facilities by a number of developers. Furthermore, our end-to-end performance evaluation shows that the time overhead introduced by BCs is negligible. However, we show that the selected bus protocol (or the Things' middleware protocols) may raise a performance or scalability issue. Hence, we have to perform further experiments under low traffic and stress conditions in order to evaluate: *i)* more middleware protocols employed as bus protocols; *ii)* several message sizes under low and stress conditions. Such a comprehensive performance evaluation will enable us to improve the

performance at runtime, by employing the proper bus protocol.

While this chapter deals with the mapping of functional end-to-end semantics, Things present additional heterogeneous characteristics in terms of non-functional semantics. Particularly, IoT applications introduce messages which may be valid for a specific time period. Additionally, typical synchronous interactions must be completed within a `timeout` period. Such behavior is supported through our GM API where messages and requests can be valid for `lifetime` or `timeout` periods. However, mobile IoT devices acting as message senders and recipients may be intermittently available introducing additional timing parameters, which affect end-to-end response times and delivery success rates. In the next chapter, we provide a formal approach of our interoperability solution focusing on the above timing behavior. The resulting formal properties aim to support application designers to tune an IoT application and achieve a favorable balance between delivery success rates and response times.

Chapter 4

Timed Protocol Analysis of Interconnected Mobile Systems

Contents

As already pointed out in Chapter 2, SOA allow heterogeneous components to interact via standard interfaces and by employing standard protocols. These are principally based on the client/server communication style, where typically a client sends a request to a server and gets the response within a `timeout` period. The successful completion of such an interaction depends on the: *i)* server's reachability; and the *ii)* time needed to process the request – in comparison to the `timeout` period applied by the client application designer. On the other hand, the advent of paradigms such as the IoT [65] involves not only conventional services but also sensor-actuator networks and data feeds. Such feeds may contain data records which are valid or available for a limited `lifetime` (time-to-live) period. Additionally, a considerable portion of IoT is mobile which results to intermittently available data recipients. The latter, in conjunction to the data availability/validity, may affect the successful delivery of data in IoT applications.

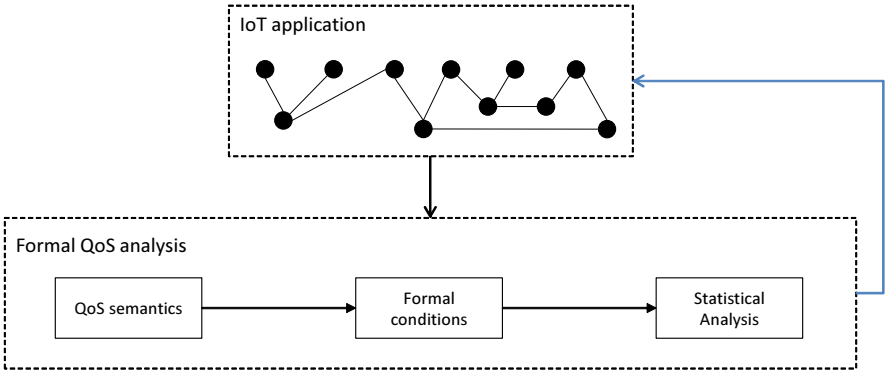


Figure 4.1: Platform for ensuring successful interactions into an IoT application.

Such timing constraints can be illustrated in the *Transport Information Management* (TIM) system (Fig. 3.1). As already presented in Chapter 3, the TIM system operates based on both authoritative (`fixed-sensors`) and mobile crowd-sourced information (`vehicles-devices` and `smartphones`) from multiple heterogeneous sources. Timing constraints may be applied to such a scenario as follows: to guarantee the freshness of provided information, notifications are maintained by the system for a (limited) `lifetime` period. Mobile Things access the system periodically and receive up-to-date transport information on their devices, but also publish traffic-related information themselves. They stay connected for a certain period (`time_on`) and then disconnect for resource saving purposes. Under these constraints, an application designer should be able to analyze and configure certain system aspects (user connectivity, message lifetime period, etc) in order to guarantee the appropriate system response time and delivery success rate.

To investigate such features, the primary purpose of this chapter is to model and analyze the aforestated QoS semantics in mobile IoT interactions. To deal with the Things' heterogeneity, we leverage the GM connector model (defined in Chapter 3), which maps end-to-end functional semantics of Things employing heterogeneous middleware protocols. Besides functional semantics, GM introduces additional *QoS semantics* (i.e., `lifetime` and `timeout` timing parameters) through its API representing data availability/validity. In this chapter, we propose a timing model that takes into account the above interaction types and represents a system relying on not only any of the CS, PS, DS and TS styles, but also any interconnection between them. We represent the behavior of our model by relying on *Timed Automata* [148]. This provides us with formal conditions for successful GM interactions and their reliance on the applied QoS semantics as well as on the stochastic behavior of interacting Things. We further perform statistical analysis through the simulation of GM interactions over multiple runs, and study the delivery success rate and response time trade-off with varying timing parameters. The model of this chapter can be used to compare between communication styles, select among them, tune the QoS semantics (or timing parameters) of the overlying application, and also do the previous

Parameter(s)	Definition/Description
$t_{\texttt{post}}$, $t_{\texttt{get}}$	at each timestamp (t) one `post` or `get` occur
$\delta_{\texttt{post}}$, $\delta_{\texttt{get}}$	the time period between two successive `post` or `get` operations
`lifetime`	`message` availability and validity in time
`time_on`, `time_off`	connected (`ON`) and disconnected (`OFF`) periods for receiving messages
$T_{\texttt{ON}}$, $T_{\texttt{OFF}}$	averages periods of `time_on`, `time_off` during the study period
`serve_time`	time needed for a request to be processed at the server side
`timeout`	required time period to complete a request-response interaction
$t_{\texttt{post\_req}}$, $t_{\texttt{get\_req}}$	at each timestamp (t) one request is sent and received, respectively
$t_{\texttt{post\_res}}$, $t_{\texttt{get\_res}}$	at each timestamp (t) one response is sent and received, respectively

Table 4.1: Analysis parameters' and shorthand notation.

when interconnection is involved. Hence, our model allows us to study, in a unified manner, time coupling and decoupling among interacting Things.

With regard to the overall contribution of this thesis (see Fig. 1.1), the contribution of this chapter is positioned as depicted in Fig. 4.1. The rest of the chapter is organized as follows. The model for timing analysis of GM interactions is introduced in Section 4.1. This is further refined with timed automata models and verification of properties in Section 4.2. The results of our analysis through simulation experiments are presented in Section 4.3, which includes comparison with experiments on the VSB testbed. This is followed by conclusions in the Section 4.4.

4.1 Time Modeling of GM Interactions

In this section, we model GM interaction types (in particular, focusing on one-way and two-way synchronous) with specific emphasis on their timing behavior [9]. We propose timing models that can represent end-to-end interactions of CS, PS, DS and TS systems, but also any interconnection between them through VSB, by relying on the GM connector.

The parameters of our timing models are depicted in Table 4.1. Among them, `lifetime` refers to emitted CS `items`/`requests`, PS `events`, DS `data` or TS `tuples`, and characterizes both data availability in time, e.g., thanks to storing by a broker, and data validity, e.g., for data that become obsolete as part of a data feed, for asynchronous interactions. In case of synchronous interactions, the `timeout` period is applied to request-response (two-way sync) interactions and represents their validity in time. Finally, `time_on` characterizes the interval during which a receiving peer is connected and available to receive one or more of the produced CS `items`, PS `events`, DS `data` or TS `tuples`, either synchronously or asynchronously. Between active intervals, the peer is disconnected (e.g., for energy-saving or other application-related reason), for period `time_off`.

In the next subsections, we detail our modeling for one-way and two-way synchronous GM interactions.

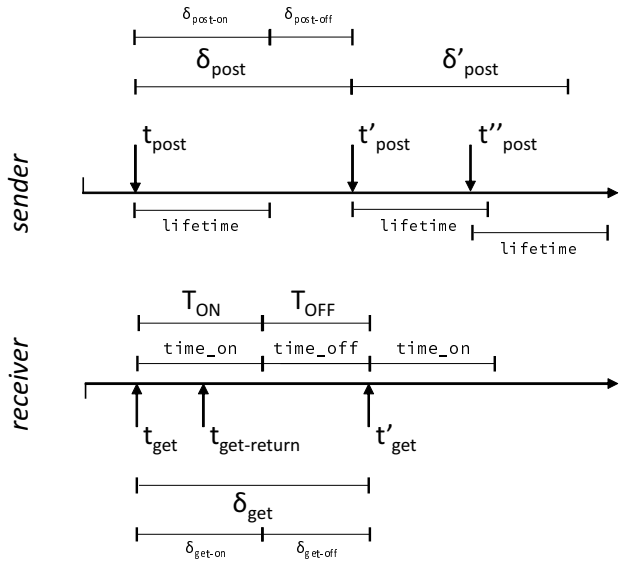


Figure 4.2: Analysis of `post` and `get` δ increments for GM one-way interactions.

One-way Interactions

We focus here on *one-way interactions* for CS, PS, DS and TS communication styles represented by GM. In particular, our analysis considers the "steady state" behavior of PS, DS and TS interactions. More specifically in PS, subscribers have been already subscribed to receive specific events when published and they do not unsubscribe during the study period. In DS, consumers have already established a session, and in TS readers/takers accessing the tuple space properly coordinate for preventing early removal of tuples by one of the peers before all interested peers have accessed these tuples.

In a GM one-way interaction a *sender* entity posts messages with a validity period `lifetime`; this message can be procured using `get` within the `time_on` period at the *receiver* side. Fig. 4.2 depicts a GM interaction as a correlation in time between a `post` operation and a `get` operation. The `post` and `get` operations are independent and have individual time-stamps. We assume that application entities (undertaking the sender and receiver roles) enforce their semantics independently (no coordination).

The `post` operation is initiated at $t_{\texttt{post}}$. A timer is started also at $t_{\texttt{post}}$, constraining the message availability to the `lifetime` period, also denoted by $\delta_{\texttt{post-on}}$. The period when the `lifetime` period elapses and the next `post` operation is yet to begin is denoted as $\delta_{\texttt{post-off}}$. Similarly at the receiver side, the `get` operation is initiated at $t_{\texttt{get}}$, together with a timer controlling the active period limited by the `time_on` (also denoted by $\delta_{\texttt{get-on}}$) interval. If `get` returns within the `time_on` period with valid data (not exceeding the `lifetime`), then the interaction is successful. We consider this instance also as the end of the `post` operation. Let T_{ON} be the average period of `time_on` periods.

`post` operations are initiated repeatedly, with an interval rate $\delta_{\texttt{post}}$ (set as a random valued variable) between two successive `post` operations. Similarly, `get` operations are initiated repeatedly, with a random valued interval equal to $\delta_{\texttt{get}}$ between the start of two successive `time_on` periods; the interval between `time_on` and the next $t_{\texttt{get}}$ qualifies the disconnection period of receivers (`time_off` or $\delta_{\texttt{get-off}}$). Let T_{OFF} be the average period of `time_off` periods. While `lifetime` and `time_on` are in general set by application/middleware designers, inter-arrival delays $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ are stochastic random variables dependent on multiple factors such as concurrent number of peers, network availability, user (dis)connections and so on.

Note that this model allows concurrent `post` messages; buffers of active receiving entities (including the broker and tuple space) are assumed to be infinite, hence there is no message loss due to limited buffering capacity. The message processing, transmission and queueing (due to processing and transmission of preceding messages) times inside the interaction are assumed to be negligible compared to durations of $\delta_{\texttt{post}}$ and `time_on` periods.

In particular regarding queueing, we assume that we have no heavy load effects. This means that: all posts arriving during an active period are immediately served; all posts arriving during an inactive period are immediately served at the next `time_on` period, unless they have expired before. This corresponds to a G/G/∞/∞ queueing model, where there are an infinite number of on-demand servers, hence there is no queueing. We assume that the general distribution characterizing service times incorporates the disconnections of receivers. We extend this model with actual queueing in Chapter 5.

Accordingly, successful one-way interactions depend on either of the disjunctive conditions:

$$t_{\texttt{get}} < t_{\texttt{post}} < t_{\texttt{get}} + \texttt{time\_on} \tag{4.1}$$

$$t_{\texttt{post}} < t_{\texttt{get}} < t_{\texttt{post}} + \texttt{lifetime} \tag{4.2}$$

meaning that a successful interaction occurs as long as a `post` and a `get` operation overlap in time. Otherwise, there is no overlapping in time between the two operations: only one of them takes place, and goes up to its maximum duration, i.e., `lifetime` for `post` and `time_on` for `get`. Precisely:

1. If `get` occurs first, and then `post` occurs before `time_on`: the interaction is *successful*. Else, `time_on` is reached, and the `get` operation yields no interaction.
2. If `post` occur first, and then `get` occurs before `lifetime`: the interaction is *successful*. Else, `lifetime` is reached, and the interaction is a *failure*.

Two-way Synchronous Interactions

In two-way synchronous interactions, a *client* entity posts requests (`post_req`) and waits to get the response (`get_res`) during `timeout`; such a request can be procured using `get_req` within the `time_on` period at the *server* side. Subsequently, the request is processed for `serve_time`

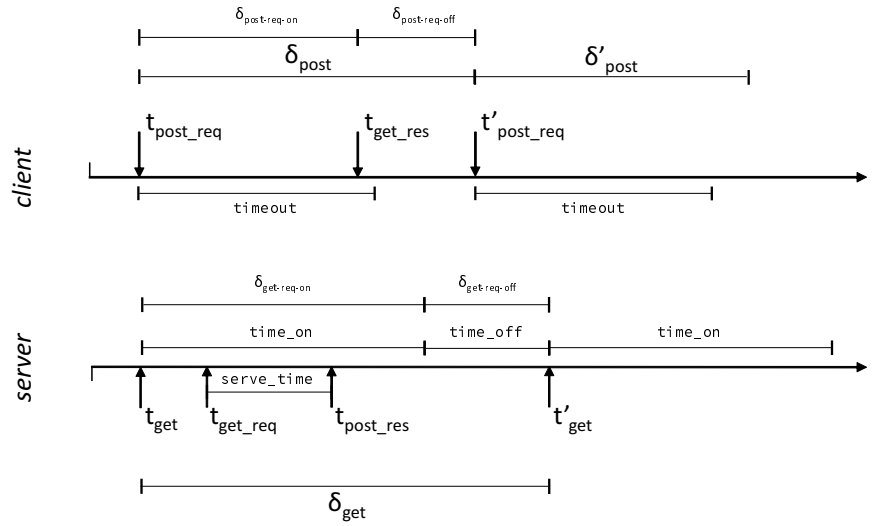


Figure 4.3: Analysis of `post` and `get` δ increments for two-way synchronous interactions.

and at the end of this period the *server* entity posts (`post_res`) the response (unless the timeout period is reached). Finally, the message is delivered using `get_res` within the `timeout` period at the *client* side. Fig. 4.3 depicts a GM interaction as a correlation in time between a `post` operation and a `get` operation. In comparison to the one-way timing model, the client/server application entities do not enforce their `post` and `get` semantics independently. In particular, the `get_req` semantic is enabled after posting a request (`post_req`). On the other hand, requests and connection periods (`time_on`) are initiated independently from each other.

The `post_req` operation is initiated at $t_{\texttt{post\_req}}$. A timer is started also at $t_{\texttt{post\_req}}$, constraining the request-response availability to the `timeout` period, also denoted by $\delta_{\texttt{post-req-on}}$. The period when the `timeout` period elapses and the next `post_req` operation is yet to begin is denoted as $\delta_{\texttt{post-req-off}}$. Similarly at the server side, the interval that allows to receive requests is initiated at $t_{\texttt{get}}$, together with a timer controlling the active period limited by the `time_on` (also denoted by $\delta_{\texttt{get-req-on}}$) interval. If `get_req` returns within the `time_on` period with valid request (not exceeding the `timeout`), then after a `serve_time` interval (and only if is still valid) the `post_res` returns the response to the client and the interaction is successful.

`post_req` operations are initiated repeatedly, with an interval rate $\delta_{\texttt{post}}$ (set as a random valued variable) between two successive `post-req` operations. Similarly, `get` operations are initiated repeatedly, with a random valued interval equal to $\delta_{\texttt{get}}$ between the start of two successive `time_on` periods; the interval between `time_on` and the next $t_{\texttt{get}}$ qualifies the disconnection period of receivers (`time_off` or $\delta_{\texttt{get-req-off}}$). `timeout` and `time_on` are in general set by application/middleware designers and inter-arrival delays $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ are stochastic random variables (dependent on network availability, user (dis)connections, etc).

Similar to the one-way timing model, this model allows concurrent `post` of requests; buffers

of active receiving entities are assumed to be infinite, hence there is no message loss due to limited buffering capacity. However, once a `post_req` is active, the client blocks its operation and waits for the response during `timeout`. The request-response transmission and queueing (due to processing and transmission of preceding messages) times inside the interaction are assumed to be negligible compared to durations of $\delta_{\texttt{post}}$ and `time_on` periods. On the other hand, the processing of requests on he server side is defined using the `serve_time` interval.
Successful interactions depend on the following condition:

$$t_{\texttt{post\_req}} < t_{\texttt{get}} + \texttt{serve\_time} < t_{\texttt{post\_req}} + \texttt{timeout} \tag{4.3}$$

meaning that a successful interaction occurs as long as: *i)* a `post_req` and a `get` operation overlap in time; and *ii)* when there is an overlap between the `post_req` and the `get` operations, the request must be served before the timeout period is reached. Otherwise, there is no overlapping in time between the two operations: only one of them takes place, and goes up to its maximum duration, i.e., `timeout` for `post_req` and `time_on` for `get`.
Failed interactions occur in the following cases:

1. When `post_req` occurs, and then `get` occurs before `timeout`: the `get_req` is enforced. Else, `timeout` is reached, and the interaction is a *failure*.
2. After enforcing the `get_req` operation, if the `get_res` occurs before `timeout`: the interaction is *successful*. Else, the `timeout` is reached due to the processing at the server side, and the interaction is a *failure*.

The above time modeling focuses on one-way and two-way synchronous GM interactions; similar models can be derived for the other GM interaction types. In this way, we can cover the various interaction types found in the IoT and represent the individual CS, PS, DS, TS styles, but also any heterogeneous interconnection between them, e.g., a PS publisher interacting with a TS reader. Interconnection is performed through the VSB framework. We assume here that the effect of the VSB bus on the timings of the end-to-end interactions is negligible. We extend this model in Chapter 5, where we actually consider the timing effect of the VSB.

4.2 Timed Automata-based Analysis

A timed automaton [148] is essentially a finite automaton extended with real-valued clock variables. These variables model the logical clocks in the system, which are initialized with zero when the system is started, and then increase synchronously at the same rate. Clock constraints are used to restrict the behavior of the automaton. A transition represented by an edge can be taken only when the clock values satisfy the *guard* labeled on the edge. Clocks may be reset to zero when a transition is taken. Clock constraints are also used as *invariants* at locations represented by vertices: they must be satisfied at all times the location is reached or maintained.

In order to study GM interactions with timed automata, we make use of UPPAAL [149]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. In such networks, automata synchronize via *binary synchronization channels*. For instance, with a channel declared as `chan c`, a transition of an automaton labeled with `c!` (sending action) synchronizes with the transition of another automaton labeled with `c?` (receiving action). UPPAAL makes use of computation tree logic (CTL) [176] to specify and verify temporal logic properties. We employ the *committed location* qualifier (marked with a 'C') for some of the locations. In UPPAAL, time is not allowed to pass when the system is in a committed location; additionally, outgoing transitions from a committed location have absolute priority over normal transitions. The *urgent location* qualifier (marked with a 'U') is also used: time is not allowed to pass when the system is in an urgent location (without the priority clause of committed locations, though).

In this section, we build timed automata models which represent the typical behavior of the GM connector for performing the timed one-way and two-way synchronous interactions described in the previous section. By relying on the expressive power of timed automata, we are able not only to model the timing conditions of such interactions, but also to introduce basic stochastic semantics regarding the behavior of peers. Using the UPPAAL model checker, we provide and verify essential properties of our timed automata model, including formal conditions for successful GM interactions.

4.2.1 Analysis of One-Way Interactions

We represent one-way GM interactions with the connector roles *GM sender*, *GM receiver*, and with the corresponding *GM one-way glue*. The two roles model the behavior expected from application components employing the connector, while the glue represents the internal logic of the connector coordinating the two roles. We detail in the following the modeling of these components.

Fig. 4.4 shows the *sender* behavior. Typically, a sender entity repeatedly emits a `post!` action (`message`) to the *glue* without receiving any feedback about the end (successful or not) of the `post` operation. We have enhanced (and at the same time constrained) the sender's behavior with a number of features. The committed locations `post_event` (`post!` sent to the glue) and `post_end_event` (`post_end?` received from the glue) have been introduced to detect the corresponding events. Upon these events, the automaton oscillates between the `post_on` and `post_off` locations, which correspond to the $\delta_{\texttt{post-on}}$ and $\delta_{\texttt{post-off}}$ intervals presented in Fig. 4.2. `delta_post` is a clock that controls the $\delta_{\texttt{post}}$ interval between two successive `post` operations. `delta_post` is reset upon a new `post` operation and set to `lifetime` at the end of this operation (note that the `post_init` location and its outgoing transition serve initializing `delta_post` at the beginning of the sender's execution – this unifies verification also for the very first `post`

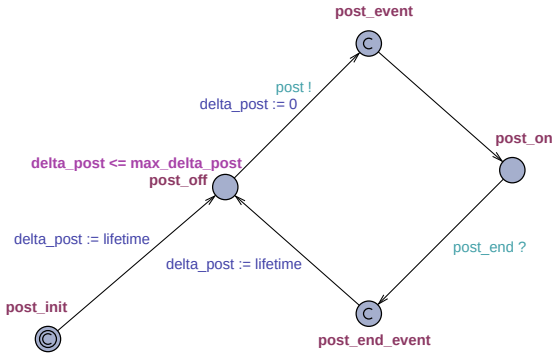


Figure 4.4: GM sender automaton.

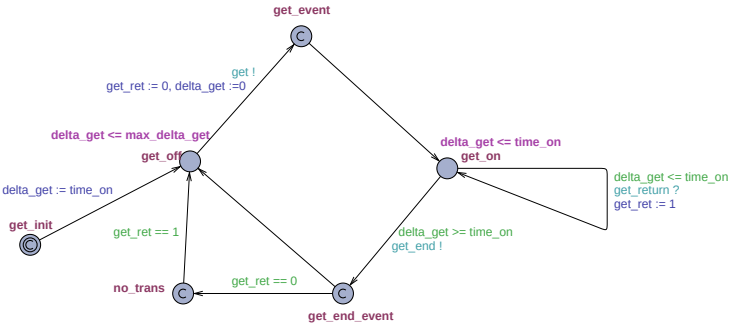


Figure 4.5: GM receiver automaton.

operation). The invariant condition `delta_post<=max_delta_post` (where `max_delta_post` is a constant) at the `post_off` location ensures that a new `post` operation will be initiated before the identified boundary.

This setup results in at most one `post` operation active at a time. This `post` remains active ($\delta_{\texttt{post-on}}$ interval) for `lifetime` interval (and then it expires) or less than `lifetime` interval (in case of successful interaction). In both cases, we set `delta_post` to `lifetime` at the end of the `post` operation (this enables verification, since we can not capture absolute times in UPPAAL). Hence, the immediately following $\delta_{\texttt{post-off}}$ interval will last a stochastic time uniformly distributed in the interval [`lifetime`, `max_delta_post`]. With regard to the one-way timing model of Section 4.1, we opted here for restraining concurrency of `post` operations for simplifying the architecture of the glue. The present model (sender, receiver and one-way glue) can be compared to one of the infinite on-demand servers of the G/G/∞/∞ model of Section 4.1. Nevertheless, this model is sufficient for verifying Conditions (4.1) and (4.2) for successful GM interactions. These conditions relate any `post` operation with an overlapping `get` operation; possible concurrency of `post` operations has no effect on this. Moreover, in the following sections we prove that these conditions are independent of the probability distributions characterizing the sender and receiver's stochastic behavior.

Fig. 4.5 shows the *receiver* behavior. Typically, a receiver entity repeatedly emits a `get!` action to the glue, with at most one `get` operation active at a time. The duration of the `get` operation is controlled by the receiver with a local `time_on`; upon the `time_on`, a `get_end!` action is sent to the glue. Before reaching the `time_on`, multiple `messages` (posted by senders) may be delivered to the receiver by the glue, each with a `get_return?` action. We have enhanced the receiver's behavior with similar features as for the sender. Hence, we capture the events and time intervals presented in Fig. 4.2 with the `get_event`, `get_end_event`, `get_on`, `get_off` locations, as well as with the `delta_get` clock and the invariant conditions `delta_get<=time_on` (at `get_on`) and `delta_get<=max_delta_get` (at `get_off`). This setup results in a succession of $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals, with the former lasting `time_on` time and the latter lasting a stochastic time uniformly distributed in the interval [`time_on`, `max_delta_get`]. We have additionally introduced

the committed location `no_trans`, which, together with the Boolean variable `get_ret`, helps detecting whether the whole `time_on` period elapsed with no interaction performed or at least one `message` was received.

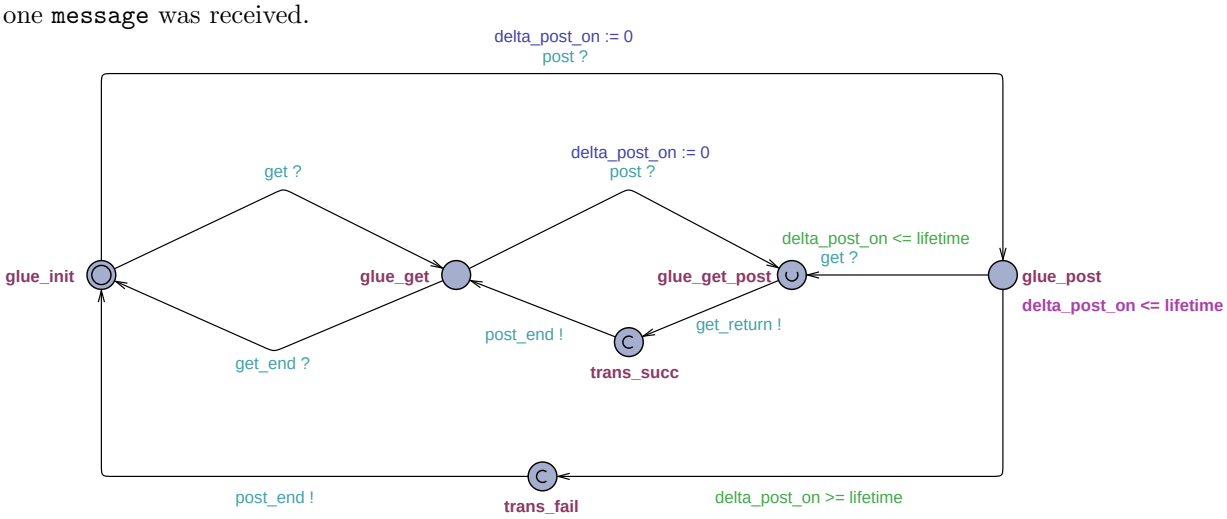


Figure 4.6: GM glue one-way automaton.

The *glue* one-way automaton is shown in Fig. 4.6. It determines the synchronization of the incoming `post?` and `get?` operations. A successful synchronization between such operations leads to a successful interaction, which is represented in the automaton by the `trans_succ` location. Note that the timing constraints specified in Section 4.1 regarding the lifetime of posted `messages` have been applied here with the additional clock `delta_post_on` employed to guard transitions dependent on the `lifetime` period. Two ways for reaching the `trans_succ` location are considered:

- If the `get?` operation occurs from the initial location (leading to location `glue_get`), a consequent `post?` operation results in a `get_return! message` and eventually the successful interaction location `trans_succ` (Eq. 4.1). At the same time, the sender is notified of the end of the `post` operation with `post_end!`. Note that we employ the *urgent location* qualifier for `glue_get_post`; thus, the glue completes instantly the successful interaction and is ready for a new one. At the `glue_get` location, if the `get_end?` action is received from the receiver automaton (suggesting `delta_get >= timeout`), the glue is reset to the initial location `glue_init`.
- If the `post?` operation occurs initially (leading to location `glue_post`), a `get?` operation before the constraint `delta_post_on <= lifetime` results again in a successful interaction (Eq. 4.2). Exceeding the `lifetime` period without any `get?` results in location `trans_fail`, and the automaton returns to its initial location `glue_init`, notifying at the same time the sender with `post_end!`. This is done without any delay, thanks to the invariant `delta_post_on <= lifetime` at the `glue_post` location.

Verification of Properties

We verify reachability and safety properties of the combined automata *GM sender*, *GM receiver* and *GM glue one-way*, by using the model checker of UPPAAL. A reachability property, specified in UPPAAL as `E<>`φ, expresses that, starting at the initial state, a path exists such that the condition φ is eventually satisfied along that path. A safety property, specified in UPPAAL as `A[]`φ, expresses that the condition φ invariantly holds in all reachable states.

Sender Automaton. We verify a set of reachability and safety properties that characterize the timings of the sender's stochastic behavior.

```
A[] sender.post_event imply delta_post==0                                   (4.4)
A[] sender.post_on imply delta_post<=lifetime                               (4.5)
A[] sender.post_off imply (delta_post>=lifetime and
                           delta_post<=max_delta_post)                      (4.6)
E<> sender.post_end_event and delta_post<lifetime                           (4.7)
```

Eq. 4.4 states that `post` events occur at time 0 captured by the `delta_post` clock. Eq. 4.5 and 4.7 together state that $[0, \texttt{lifetime}]$ is the maximum interval in which a `post` operation is active, nevertheless, the operation can end before `lifetime` is reached. Eq. 4.6 states that $[\texttt{lifetime}, \texttt{max\_delta\_post}]$ is the maximum interval in which there is no active `post` operation. This confirms the fact that we artificially "advance time" to `lifetime` at the end of the `post` operation.

Receiver Automaton. We verify similar properties that characterize the timings of the receiver's stochastic behavior.

```
A[] receiver.get_event imply delta_get==0                                   (4.8)
A[] receiver.get_on imply delta_get<=time_on                                (4.9)
A[] receiver.get_off imply (delta_get>=time_on and
                            delta_get<=max_delta_get)                       (4.10)
A[] receiver.get_end_event imply delta_get==time_on                         (4.11)
```

Hence, Eq. 4.8 states that `get` events occur at time 0 captured by the `delta_get` clock. Eq. 4.9 and 4.11 together state that a `get` operation precisely and invariantly terminates at the end of the $[0, \texttt{time\_on}]$ interval. Eq. 4.10 states that $[\texttt{time\_on}, \texttt{max\_delta\_get}]$ is the maximum interval in which there is no active `get` operation.

Glue one-way Automaton. Finally, we verify conditions for successful interactions using the glue automaton.

```
A[] glue.trans_succ imply (sender.post_on and receiver.get_on
                           and (delta_post==0 or delta_get==0))              (4.12)
```

In addition to the reachability property (`E<> glue.trans_succ`), we verify the safety property in Eq. 4.12. According to this, a successful interaction event implies that while a `post` operation is active a `get` event occurs, or while a `get` operation is active a `post` event occurs.

```
A[] glue.trans_fail imply (sender.post_on and receiver.get_off
     and delta_post==lifetime and delta_get-time_on>=lifetime)
```
(4.13)

In addition to the reachability property (`E<> glue.trans_fail`), we verify the safety property in Eq. 4.13. A failed interaction event means that `lifetime` is reached for an active `post` operation and no `get` operation is active. Additionally, the ongoing inactive `get` interval entirely includes the terminating active `post` interval. With regard to the stochastic `post` and `get` processes of our specific setting, we explicitly checked that if the condition `max_delta_get-time_on>=lifetime` does not hold for the given values of the included constants, then the reachability property `E<> glue.trans_fail` is indeed not satisfied.

```
A[] receiver.no_trans imply (receiver.get_on and sender.post_off
        and delta_get==time_on and delta_post-lifetime>=time_on)
```
(4.14)

In addition to the reachability property (`E<> receiver.no_trans`), we verify the safety property in Eq. 4.14. Symmetrically to Eq. 4.13, a no-interaction event implies that `time_on` is reached for an active `get` operation and no `post` operation is active. Additionally, the ongoing inactive `post` interval entirely includes the terminating active `get` interval. Similarly to Eq. 4.13, we check that if this safety property is not satisfied, then the state `receiver.no_trans` is indeed not reachable.

Checking Eqs. 4.12, 4.13, 4.14, successful interactions are determined by the durations and relative positions in time of the $\delta_{\texttt{post-on}}$, $\delta_{\texttt{post-off}}$, $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals. These depend on the deterministic parameter constants `lifetime`, `time_on` and on the stochastic parameters $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$. Nevertheless, Eqs. 4.12, 4.13, 4.14 are expressed in a general way, independently of the specific `post` and `get` processes. Hence, the analysis results of this section provide us with general formal conditions for successful GM interactions and their reliance on observable and potentially tunable system and environment parameters. Using these results, we perform experiments to quantify the effect of varying these parameters for successful interactions in Section 4.3.

4.2.2 Analysis of Two-Way Synchronous Interactions

We represent two-way synchronous GM interactions with the connector roles *GM client*, *GM server*, and *GM two-way sync glue*. The two roles model the behavior expected from application components employing the connector, while the glue represents the internal logic of connecting the two roles. We detail in the following the modeling of these components.

Fig. 4.7 shows the *client* behavior. Typically, a client emits a `post_req` (`request`) to the *glue* and waits for `timeout` to receive the `get_res` (`response`). The committed location

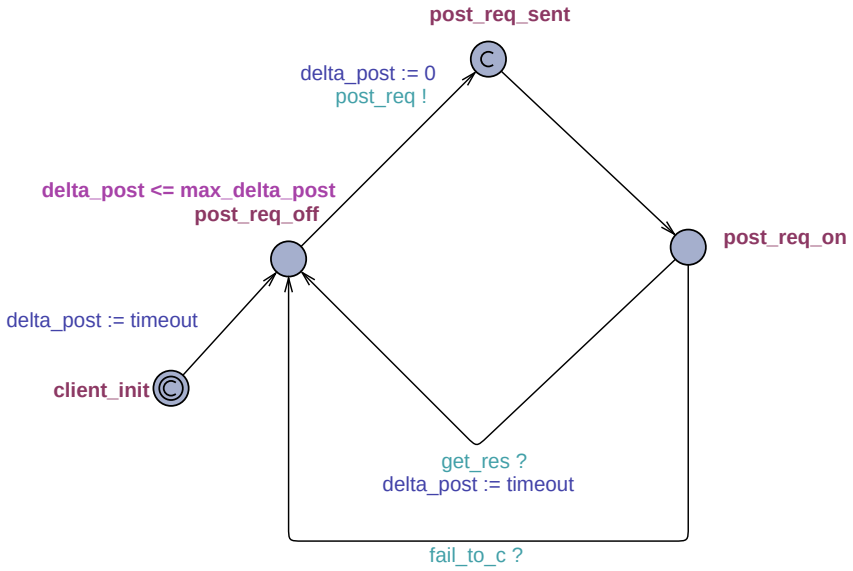


Figure 4.7: GM client automaton.

`post_req_sent` is introduced to detect the event of sending a request (`post_req!`) to the glue. Upon such an event, the automaton stays on the `post_req_on` location to either receive the response or until the `timeout` expires, which corresponds to the $\delta_{\texttt{post-req-on}}$ interval presented in Fig. 4.3. Upon the timeout expiration or the `get_res` reception, the automaton stays to the `post_req_off` for $\delta_{\texttt{post-req-off}}$ time period.

`delta_post` is a clock that controls the $\delta_{\texttt{post}}$ interval between two successive `post_req` operations. `delta_post` is reset upon a new `post_req` operation and set to `timeout` upon a `get_res?` (prior to the `timeout` expiration). On the other hand, when the timeout period is reached, the `delta_post` clock is already set to timeout on the `post_req_off` location. Similar to one-way interactions, we initialize `delta_post` at the beginning of the client's execution (`post_init` location) to unify our verification also for the very first `post_req` operation. The invariant condition `delta_post<=max_delta_post` (where `max_delta_post` is a constant) at the `post_req_off` location ensures that a new `post_req` operation will be initiated before the identified boundary.

Based on the above setup, the client sends at most one `post_req` operation active at a time. This request remains active ($\delta_{\texttt{post-req-on}}$ interval) for `timeout` period (and then it expires) or less than `timeout` period (in case of successful interaction). In both cases, `delta_post` equals to `timeout` at the end of the `post_req` operation. Hence, the immediately following $\delta_{\texttt{post-off}}$ interval will last a stochastic time uniformly distributed in the interval [`timeout`, `max_delta_post`]. Such a model is sufficient for verifying the condition (Eq. 4.3) for successful GM two-way sync interactions. This condition relate any `post_req` operation with an overlapping `get` operation, by taking also into account the `serve_time` at the server side.

Fig. 4.8 shows the *server* behavior. Typically, a server entity repeatedly becomes online (location `get_on`) to receive requests from the glue. Thus, the server automaton oscillates between

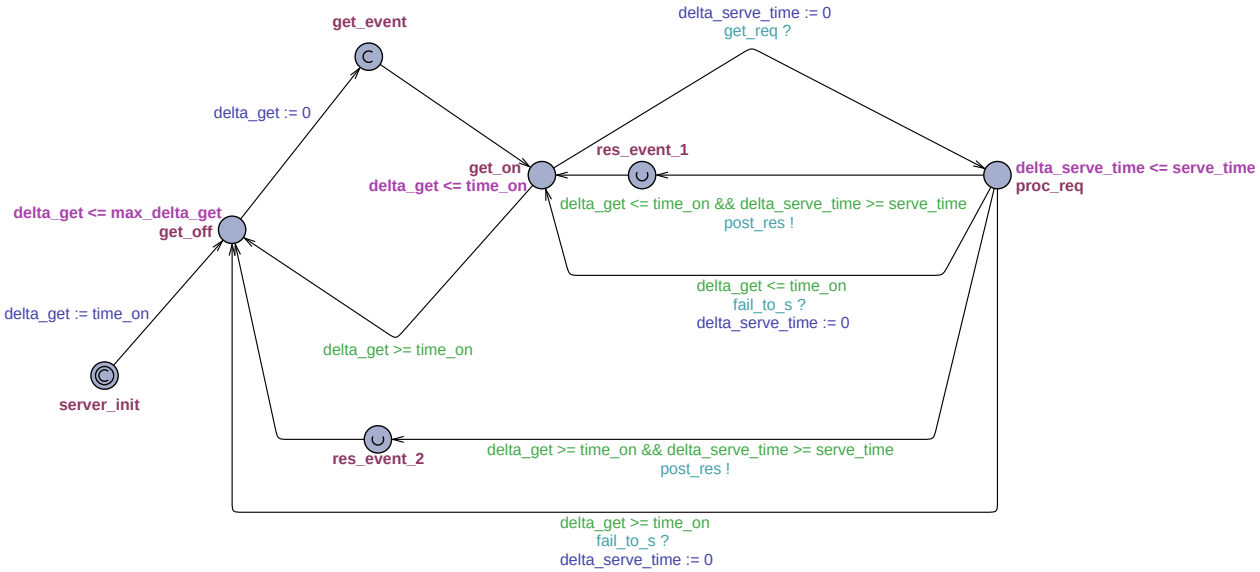


Figure 4.8: GM server automaton.

the locations `get_off` and `get_on`. The `get_event` committed location is used to detect the online status of the server. It is worth noting that the server entity operates independently from the glue – i.e., it does not notify the glue when changing between the `get_on` and `get_off` locations. The automaton stays on the `get_on` location for a specific interval, which is controlled by the server with a local `time_on`; upon the `time_on`, the automaton returns to the `get_off` location. Similar to the client entity, the `delta_get` clock is used to measure the `time_on` interval and switch between the two locations (`get_on` and `get_off`). Furthermore, the invariant conditions `delta_get<=time_on` (at `get_on`) and `delta_get<=max_delta_get` (at `get_off`) guarantee the correct operation of our automaton.

Before reaching the `time_on`, multiple `requests` (posted by clients) may be delivered to the clients by the glue, each with a `get_req?` action. Upon a `get_req?`, the automaton stays in the `proc_req` location for `serve_time` interval, which corresponds to the necessary time period for processing a request. We use the *urgent* `res_event_1` and `res_event_2` locations to detect successful responses through the `post_res!` action. Particularly, the `res_event_1` location is reached only if the server is still online (`delta_get<=time_on`). However, while being in location `proc_req`, the server entity may become offline. For such case, the `res_event_2` location is reached after serving the request (because of the invariant `delta_serve_time<=serve_time`), and then the automaton returns to the `get_off` location. Finally, the automaton returns to `get_on` or `get_off` locations upon a `fail_to_s?` action received by the glue, which corresponds to the request (`post_req`) expiration due to the `timeout` period.

This setup results in a succession of $\delta_{\texttt{get-req-on}}$ and $\delta_{\texttt{get-req-off}}$ intervals (see Fig. 4.3), with the former lasting `time_on` time and the latter lasting a stochastic time uniformly distributed in the interval [`time_on`, `max_delta_get`].

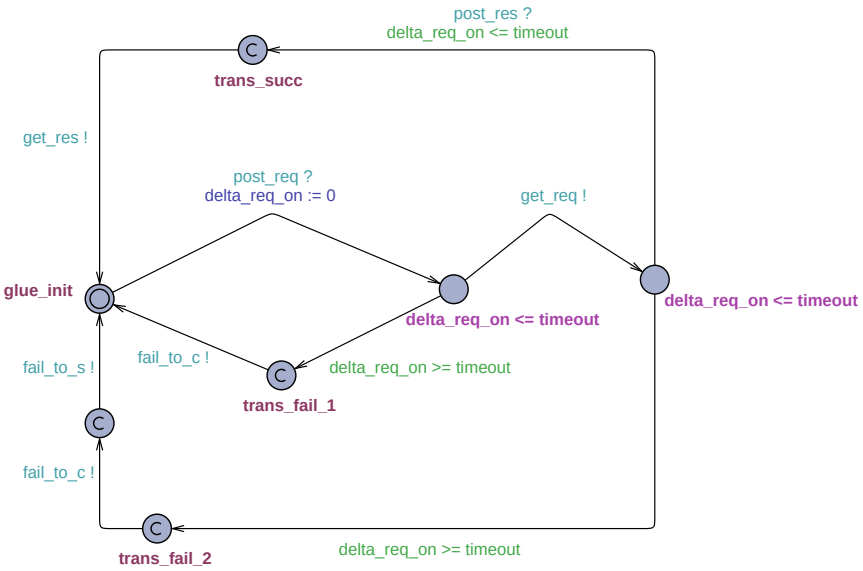


Figure 4.9: GM glue two-way synchronous automaton.

The *glue* two-way sync automaton is shown in Fig. 4.9. It determines the synchronization of the incoming (`post_req?` and `post_res?`) and outgoing (`get_req!`) operations. A successful synchronization between such operations leads to a successful interaction, which is represented in the automaton by the `trans_succ` location. Note that the timing constraints specified in Section 4.1 regarding the `timeout` of sent requests have been applied here with the additional clock `delta_req_on` employed to guard transitions dependent on the `timeout` period.

The `trans_succ` location is reached through the following operations: if the `post_req?` operation occurs from the initial location and the invariant `delta_re1_on<=timeout` is satisfied, a consequent `get_req!` request is sent to the server (if the server automaton is on the location `get_on`). While the request is processed on the server side, the glue automaton waits for the reply. After the specified `server_time` a `post_res?` operation occurs to the glue and eventually the successful interaction location `trans_succ` (the Eq. 4.3 is satisfied). At the same time, the client is notified of the end of the `post_req` operation with `get_ges!`. Note that we employ the `get_req` channel as `urgent`. In this way, upon a `post_req?` and if the server is online, the `get_req!` action occurs instantly, without any delay as indicated by the invariant `delta_re1_on<=timeout`.

With regard to the `timeout`, `time_on` and `serve_time` parameters, we identify failed interactions in the glue trough the `trans_fail_1` and `trans_fail_2`. Two ways for reaching the fail locations are considered:

- If the `post_req?` operation occurs from the initial location and the server automaton is offline (stays on the `get_off` location) for time period that leads to the timeout expiration (`delta_req_on>=timeout`), the `trans_fail_1` location is reached. At the same time, the client is notified with `fail_to_c!` in order to move at the `post_req_off` location.

- If the post_req? operation occurs from the initial location and the server automaton is online (stays on the get_on location), a consequent get_req! request is sent to the server. While the request is processed for serve_time, the timeout period may expire (delta_req_on>=timeout) and the trans_fail_2 location is reached. At the same time, the client is notified with fail_to_c! to move at the post_req_off location, and the server with fail_to_s! to move either on get_on or on get_off locations, depending of the delta_get clock.

Verification of Properties

Similar to one-way interactions, we verify reachability (E<>φ) and safety (A[]φ) properties of the combined automata *GM client*, *GM server* and *GM two-way sync glue*, by using the model checker of UPPAAL.

Client Automaton. We verify a set of safety properties that characterize the timings of the client's stochastic behavior.

```
A[] client.post_req_sent imply delta_post==0                          (4.15)

A[] client.post_req_sent imply delta_post<=timeout                    (4.16)

A[] client.post_req_off imply (delta_post>=timeout and
                               delta_post<=max_delta_post)            (4.17)
```

Eq. 4.15 states that post_req events occur at time 0 captured by the delta_post clock. Eq. 4.16 states that $[0, \texttt{timeout}]$ is the maximum interval in which a post_req operation is active, nevertheless, the operation can end before timeout is reached. Eq. 4.17 states that $[\texttt{timeout}, \texttt{max\_delta\_post}]$ is the maximum interval in which there is no active post_req operation. Similar to the *GM sender* automaton, we artificially "advance time" to timeout at the end of the post_req operation.

Server Automaton. We verify similar properties that characterize the timings of the server's stochastic behavior.

```
A[] server.get_event imply delta_get==0                               (4.18)

A[] server.get_on imply delta_get<=time_on                            (4.19)

A[] server.get_off imply (delta_get>=time_on and
                          delta_get<=max_delta_get)                   (4.20)
```

Eq. 4.19 states that at the beginning of the server's online period, the automaton passes from the location get_event at time 0 captured by the delta_get clock. Eq. 4.21 states that the server stays online (at the location get_on) at least for time_on interval. Eq. 4.22 states that $[\texttt{time\_on}, \texttt{max\_delta\_get}]$ is the maximum interval in which the server is offline (at the location get_off).

Glue two-way sync Automaton. Finally, we verify conditions for successful interactions using the glue automaton.

```
A[] glue.trans_succ imply (client.post_req_on and delta_post<=timeout
                        and (server.res_event_1 or server.res_event_2)
                                and delta_get<=time_on+serve_time)
```
(4.21)

In addition to the reachability property (`E<> glue.trans_succ`), we verify the safety property in Eq. 4.21. According to this, a successful interaction event implies that while a `post_req` operation is active the `timeout` period is not reached. Additionally on the server side, one of the committed locations `res_event_1` or `res_event_2` is active and the condition `delta_get<=time_on+serve_time` holds.

```
A[] glue.trans_fail_1 imply (client.post_req_on and delta_post==timeout
                    and ((server.get_off and delta_get-time_on>=timeout)
                                or (server.get_on and delta_get==0)))
```
(4.22)

In addition to the reachability property (`E<> glue.trans_fail_1`), we verify the safety property in Eq. 4.22. A failed interaction event means that `timeout` is reached for an active `post_req` operation. Additionally, the request can not reach the server either because is offline (`get_off` location) for time period greater of timeout (`delta_get-time_on>=timeout`), or due the server automaton moved on the location `get_on` and at the same time the `timeout` period is reached.

```
A[] glue.trans_fail_2 imply (client.post_req_on and delta_post==timeout
                        and server.proc_req and delta_get<=serve_time)
```
(4.23)

Upon an interaction if the above condition is not verified, it means that the request is processed at the server side. However, an additional failure can occur in location `trans_fail_2` while the request is processed. In addition to the reachability property (`E<> glue.trans_fail_2`), we verify the safety property in Eq. 4.23. Such a failed interaction event means that `timeout` is reached for an active `post_req` operation. Additionally, the request is processed in location `proc_req` since the condition `delta_get<=serve_time` is valid.

Similar to one-way interactions, Eqs. 4.21, 4.22 and 4.23 provide us with general formal conditions which can be utilized by system designers to tune timing parameters such as `timeout`, `time_on` and `serve_time` and achieve successful interactions.

4.3 Simulation-based Analysis

In this section, we provide results of simulations of GM one-way interactions with varied `lifetime` and `time_on` periods. We demonstrate that varying these periods has a significant effect on the rate of successful interactions. Furthermore, the trade-off involved between delivery success rates

and response times (depending on `lifetime`/`time_on` periods) is evaluated. Finally, we validate our simulation-based analysis by using the VSB framework which provides implementations of the GM interactions through real middleware protocols.

4.3.1 Delivery Success Rates

In order to test the effect of varying `lifetime` and `time_on` periods on interaction success rates, we perform simulations over the timing analysis one-way model described in Section 4.1. *Poisson* arrival rates are assumed for subsequent t_{post} instances (hence, $\delta_{\texttt{post}}$ follows the corresponding exponential distribution). Each message is valid for a deterministic `lifetime` period and then discarded. Similarly, there are *exponential* intervals between subsequent $t_{\texttt{get}}$ periods ($\delta_{\texttt{get}}$ follows this distribution). The receiver entity is active for a deterministic `time_on` period and can disconnect for random valued intervals. Applying the one-way timing model in Section 4.1, the simulation enables concurrent posts with no-queueing. As the arrivals follow a *Poisson* process, this simulates an M/G/∞/∞ queueing model.

The simulations done in *Scilab*[1] analyze the effect of varying `lifetime` and `time_on` periods on VSB interactions. We set $\delta_{\texttt{post}}$ between subsequent `post` messages to have a mean of 10 sec. The `get` messages are simulated with varying exponential active periods ($\delta_{\texttt{get}}$). This procedure was run for $10,000$ $t_{\texttt{get}}$ periods to collect interaction statistics, by applying the formal conditions of subsection 4.2.1.

The rates of successful interactions are shown in Fig. 4.10 for various values of `lifetime`, `time_on` and δ_{get} periods. As expected, increasing `time_on` periods for individual `lifetime` values improves the success rate. However, notice that the success rate is severely bounded by `lifetime` periods. For instance, when the `lifetime` period is very low (0 sec), the success rate, even at higher `time_on` intervals, remains bound at around 70% for $\delta_{\texttt{get}}$ with mean 40 sec. Such behavior represents time/space coupled CS interactions, where each message is received immediately by the receiver (we assume that the transmission delay of the underlying network delay is negligible). Reducing `get` disconnection intervals (by properly setting $\delta_{\texttt{get}}$ and `time_on`) produces a significant improvement in the success rate, especially for the CS case. For the other communication styles (PS/TS employing an intermediate middleware node), where the `lifetime` period can be varied: a higher `lifetime` period combined with higher `time_on` or lower $\delta_{\texttt{get}}$ intervals would guarantee better success rates.

4.3.2 Response Time vs. Delivery Success Rate

In order to study the trade-off between end-to-end response time and delivery success rate, we present cumulative response time distributions for interactions in Fig. 4.11. Note that we assume that all posts arriving during an active `get` period are immediately served; all posts arriving

[1] `http://www.scilab.org`

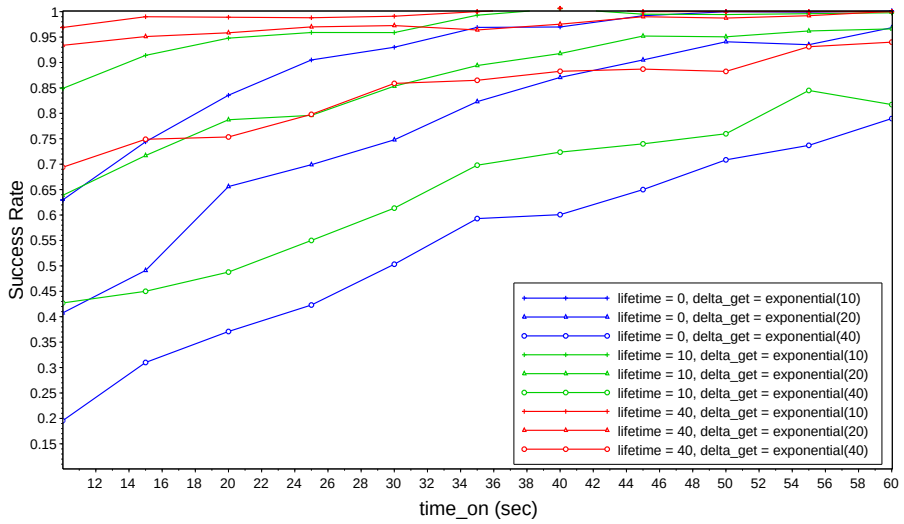


Figure 4.10: Delivery success rates with varying `time_on` and `lifetime` periods.

during an inactive `get` period are immediately served at the next active period, unless they have expired before. All failed transactions are pegged to the value: `lifetime`.

We set $\delta_{\texttt{post}} = Poisson(10)$ sec and $\delta_{\texttt{get}} = Exponential(20)$ sec for all simulated cases. From Fig. 4.11, lower `lifetime` periods produce markedly improved response times. For instance, with `lifetime` = 10 sec, `time_on` = 20 sec, all interactions complete within 10 sec. Comparing this to Fig. 4.10, the success rate with these settings is 78%. Changing to `lifetime` = 40 sec, `time_on` = 20 sec, we get a success rate of 95%, but with increased response time. So, with higher levels of `lifetime` periods (typically PS/TS), we notice high success rates, but also higher response time. While individual success rates and response time values depend also on the network/middleware efficiency, our analysis provides general guidelines for setting the `lifetime` and `time_on` periods to ensure successful interactions.

Our fine-grained timing analysis can be employed to properly configure the TIM system. Accordingly, `vehicle-devices` and `fixed-sensors` emit `post`s carrying traffic-related messages with a mean arrival rate of 1 event every 10 min. To guarantee the freshness of provided information, notifications are maintained by the system for a `lifetime` period of 10 min. We assume that `mobile-users` access the system every 20 min on average to receive up-to-date transport information on their hand-held devices. They stay connected for a `time_on` period and then disconnect, also for resource saving purposes. Actual connection/disconnection behavior is based on the user's profile. By relying on our statistical analysis, an application designer may configure the `time_on` period of user access to 10 min. Using scaled values from Figs. 4.10 and 4.11, this guarantees that the user will receive on average 65% of the posted notifications, within at most 8 min of response time with a probability of 0.63. If these values are insufficient

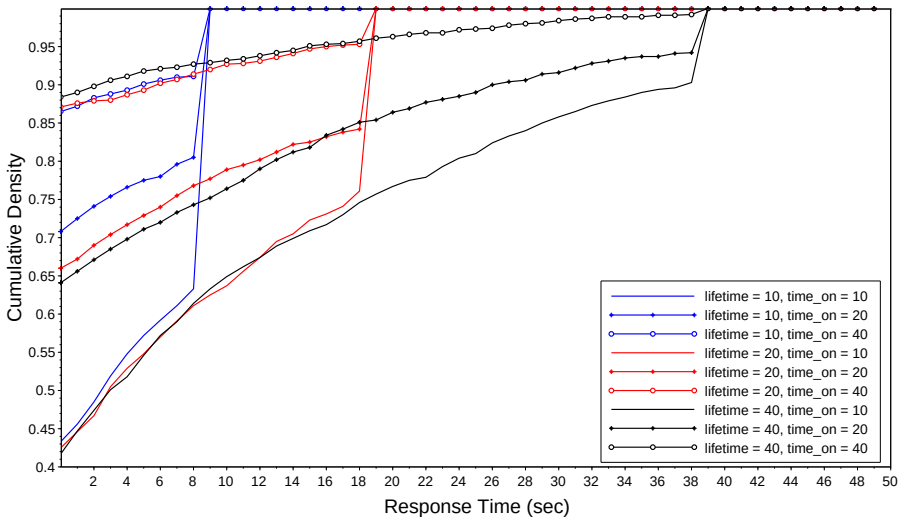


Figure 4.11: Response time distributions for interactions with varying `time_on` and `lifetime` periods.

and the designer re-configures the `time_on` to 20 min, this guarantees that now the user will receive on average 80% of the posted notifications, within at most 4 min of response time with a probability of 0.77.

4.3.3 Comparison with VSB Implementation

In order to validate the simulations performed in Section 4.3.1, we implement realistic interactions using the VSB framework. Specifically, we use two middleware implementations: i) for `lifetime` = 0 transactions, the DPWS[2] CS middleware provides an API to set a sender and a receiver interacting with each other directly; and ii) for (`lifetime` > 0) transactions, the JMS[3] PS middleware provides an API to set a sender, a receiver, and the intermediate entity through which they interact. Applying the same settings as in Section 4.3.1, senders and receivers perform operations based on probability distributions (exponential $\delta_{\texttt{post}}$ with mean of 10 sec and $\delta_{\texttt{get}}$ with various mean periods). At the intermediate entity we set various `lifetime` periods, using the JMS API. Note that in these VSB implementation settings, we have concurrent `post`s and queueing. This corresponds to an M/G/1/∞ queueing model; however, the queueing time of data due to processing of preceding data is negligible in our specific settings. All the interactions are performed using an Intel Xeon W3550e 3.08 GHz $\times$ 4 (7.8GB RAM) under a *Linux Mint* OS. For getting reliable results, the mean values of $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ intervals are expected to be close to the expected mean values. To do so, we create sufficient number of `post` operations and `get` connections/disconnections by running each experiment for at least 2 hours. In Table 4.2,

[2] `http://ws4d.e-technik.uni-rostock.de/jmeds`
[3] `http://activemq.apache.org`

`lifetime` (s)	$\delta_{\texttt{get}}$ (s)	**Simulation**	**Measurement**
0	`exponential(20)`	0.65	0.717
0	`exponential(40)`	0.35	0.42
10	`exponential(20)`	0.75	0.778
10	`exponential(40)`	0.48	0.554
40	`exponential(20)`	0.93	0.91
40	`exponential(40)`	0.75	0.81

Table 4.2: Simulated vs. measured delivery success rates.

we compare the results of simulated and measured success rates for `time_on` = 20 sec, $\delta_{\texttt{post}} = Poisson(10)$ sec, `lifetime` $= 0, 10, 40$ sec and various $\delta_{\texttt{get}}$. The absolute deviation between the two is no more than 10%. This deviation may be attributed to implementation factors such as network delays and buffering at each entity (sender, receiver, intermediate entity) which may affect the success rates. As this deviation is not too high, it allows developers to rely on our simulation model to tune the system.

4.4 Discussion

Timing constraints have typically been used for time-sensitive systems to ensure properties such as deadlock freeness and time-bounded liveness. In this chapter, we leverage the GM connector model to analyze the timing behavior of middleware IoT interactions. By including additional QoS semantics, we model such behavior by relying on timed automata. Verification of conditions for successful GM interactions is done in Uppaal in conjunction with the timing guards specified. We demonstrate that accurate setting of `lifetime`, `timeout`, `time_on` and `time_off` periods significantly affects the delivery success rate. By providing a fine-grained analysis of the related timing thresholds for application designers, increased probability of successful interactions can be ensured. This is crucial for accurate runtime behavior, especially in the case of heterogeneous space-time coupled/decoupled interactions with variable peer connectivity. Furthermore, we demonstrate that the delivery success rate vs. response time tradeoff can be suitably configured for heterogeneous middleware interactions in IoT applications.

Despite the fact that our timing analysis provides formal conditions to application designers to tune and ensure successful interactions, we assume that all posts arriving during an active period are immediately served. The latter means that when a message is transmitted, it does not meet any transmission delay due to its underlying protocol infrastructure. Furthermore, the delay of messages passing through the synthesized BCs is negligible. This end-to-end model corresponds to a G/G/∞/∞ queueing model where there is an infinite number of on-demand servers (hence there is no queueing) and the general distribution characterizing service times incorporates the disconnections of Things. Furthermore, actual disconnections occur only due to the user's behavior and not due to the wireless network coverage. To extend this work and consider the effect of additional QoS semantics such as: *i)* transmission delays; *ii)* network

disconnections; *iii)* application-layer service times; etc, it is essential to consider the delays introduced due to the queueing effect. Extending this model with actual queueing is part of our work in Chapter 5.

Chapter 5

Performance Evaluation of Interconnected Mobile Systems

Contents

In the previous chapter, we evaluated the end-to-end response times and delivery success rates in heterogeneous IoT interactions where Things have different functional and QoS semantics. The considered QoS semantics concern data validity/availability as well as the intermittent availability of data recipients due to energy saving purposes – assuming that the effect of the end-to-end protocol infrastructure is negligible. This chapter includes further QoS semantics by modeling the end-to-end infrastructure of mobile IoT applications. In particular, the performance of such applications may be constrained by queueing delays and message losses due to heavy traffic load and also due to low network bandwidth or even network disconnections occurring in the mobile IoT. Existing (IP-based) IoT protocols, such as CoAP, MQTT, DPWS, XMPP and ZeroMQ [59, 60], are being used today to face these constraints.

Each such protocol inherits the QoS characteristics of the underlying transport mechanism (reliable TCP or unreliable UDP) and implements its own modes of message delivery for satisfying application QoS requirements concerning response times and message delivery success rates. For instance, CoAP offers a choice between "confirmable" and "non-confirmable" message delivery, whereas MQTT supports three choices ("fire-and-forget", "delivered-at-least-once" and "delivered-exactly-once") [128, 177]. Depending on the selected mode, response times and delivery success rates differ significantly: the employment of reliable delivery results in higher response times, while the use of unreliable delivery results in message losses.

To illustrate the above QoS semantics, we discuss their application in the TIM system. The TIM system guarantees the freshness of provided information by applying a `lifetime` period to each message. However, messages may be delivered with delay because of disconnections. Apart from intermittent connectivity due to resource saving purposes, the actual connection/disconnection status of mobile Things depends also on the network coverage of the specific area. Among the heterogeneous Things, the `estimation-service`, `fixed-sensors` and `smartphones` employ the reliable (built atop TCP) REST, WebSockets and SemiSpace middleware protocols, respectively. On the other hand, `vehicle-devices` produce notifications via the MQTT protocol using the "fire-and-forget" delivery mode, which ensures the fastest but not reliable message delivery. Nevertheless, to receive up-to-date transport information on their devices they use the "delivered-exactly-once" mode, which ensures the reliable delivery of information. Finally, the deployed BCs employ CoAP as their common protocol utilizing the "non-confirmable" mode for collecting traffic notifications and the "confirmable" one to provide the estimated traffic to the end-users.

Under the aforestated constraints, an application designer should be able to analyze and possibly configure certain QoS semantics (protocol QoS features, network and user connectivity, message lifetime periods, allocated system resources) in order to provide appropriate response times and delivery success rates in IoT applications. To investigate such possibilities, it is essential to model the performance of the underlying middleware protocols. In [60, 177, 178] the trade-off between response times and delivery success rates by using key IoT protocols is evaluated (e.g., for CoAP). However, the employed methods are protocol-specific and do not cover the introduction of a new IoT protocol. Several existing efforts concerning the design and evaluation of mobile systems aim at satisfying QoS requirements under several constraints (e.g., intermittent availability, limited resources, etc). The evaluation methods used are derived mainly from the field of *Queueing Theory*. For instance, 2-Dimensional (2-D) Markov chains and quasi-birth-death (QBD) processes have been used in [131, 134, 136, 137] to model the changing connectivity of mobile users when offloading computation or data to the Cloud through either 3G or WiFi connections. The above efforts have already been discussed in more detail in Chapter 2.

In this chapter, we model the performance of middleware protocols by relying on *Queue-*

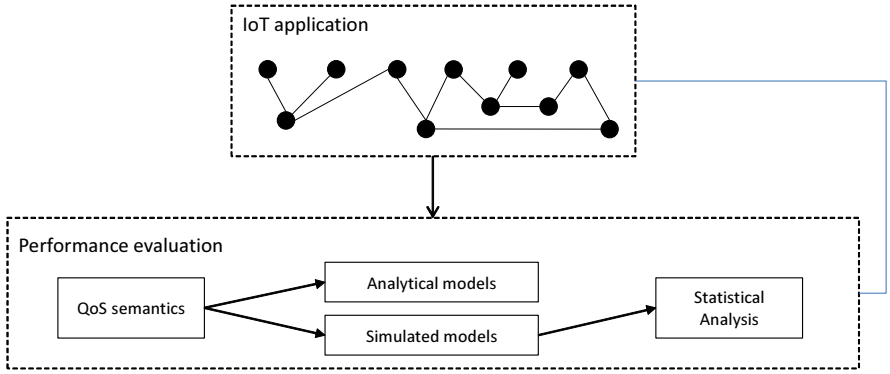


Figure 5.1: Platform for evaluating the performance of an IoT application.

ing Network Models (QNMs) [179, 180]. Based on QNMs, middleware nodes (clients, servers, brokers, etc) are represented as queues, called *service centers* or *queueing centers*, and the exchanged messages as *jobs* served. QNMs have been extensively applied to represent and analyze communication and computer systems and have proved to be simple and at the same time powerful tools for system designers with regard to system performance evaluation and prediction. In particular, queueing networks have simple analytical solutions and can achieve a favorable balance between accuracy and efficiency. They enable the isolation and analysis of each service center from the rest of the network. The solution of the entire network can be formed by combining these separate solutions. In this chapter, we analyze a service center that represents an intermittently connected mobile Thing. This is modeled explicitly as an "ON/OFF queueing center". We further extend it by introducing parameters for message availability in time and queue capacity. The key contributions of this chapter are:

- An extensive analysis and related performance metrics of the "ON/OFF queueing center", which can then be used as separate component inside queueing networks. Hence, we enrich the existing bibliography on QNMs and their solutions.
- Performance modeling patterns for our CS, PS, DS and TS core communication styles, which rely on QNMs (including the ON/OFF queueing center) and include modeling of reliable/unreliable mobile IoT protocols.
- Combining our performance modeling patterns to further model the performance of end-to-end interconnections between different communication styles via the VSB.
- Validating our models by using simulations based on both probability distributions and real-world workload traces.

With regard to the overall contribution of this thesis (see Fig. 1.1), the contribution of this chapter is positioned as depicted in Fig. 5.1. The rest of this chapter is organized as follows: In Section 5.1, we define our queueing models and we provide the theoretical analysis of the "ON/OFF queueing center". Performance modeling patterns for reliable/unreliable middleware IoT protocols that follow our core communication styles are provided in Section 5.2. In Sec-

Variable(s)	Definition/Description
λ_{in}, λ_{out}, $\lambda_{\text{out}}^{\text{on/off}}$	input and output rates of messages at the M/M/1 and ON/OFF queueing centers
D	service demand for the processing of messages at any queueing center
ON, OFF	mobile peers states: ON for connected, OFF for disconnected
T_{ON}, T_{OFF}	average duration of ON and OFF periods
in, off	classes: in for incoming messages, off for virtual messages
λ_{in}, λ_{off}	arrival rates for each class
$R_{\text{in}}^{\text{m/m/1}}$, $R_{\text{in}}^{\text{on/off}}$	overall response time at the *m/m/1* and *ON/OFF* queueing centers
D, D_{off}	service demands for *in* and *off* classes at the *ON/OFF queueing center*
R_{in}, R_{off}	response time for *in* and *off* classes at the *ON/OFF queueing center*
Q_{in}, Q_{off}	queue size for *in* and *off* classes at the *ON/OFF queueing center*
$Q_{\text{off}}^{\text{pre}}$, $Q_{\text{off}}^{\text{post}}$	number of *off* messages found at the *ON/OFF queueing center* when (pre) and after (post) the *in* message arrived

Table 5.1: Queueing models' variables and shorthand notation.

tion 5.3, we model and evaluate the end-to-end message delivery for: *i)* heterogeneous mobile Things and *ii)* Things that operate in large scale. Finally, comparison with simulations is considered in Section 5.4 followed by conclusions in Section 5.5.

5.1 Queueing Models for Mobile IoT Interactions

To model end-to-end mobile IoT interactions for performance analysis, we rely on queueing theory. In particular, we use simple input and output queues to estimate response times and message delivery success rates, as part of analytical or simulation models. Considering an end-to-end interaction between a sender and a receiver, an *input queue* can be used to receive and process messages (at the receiver's side) and an *output queue* to transmit them (at the sender's side). Each *queue* or *queueing center* serves messages through a dedicated *server*. Each server supports a specific *service demand* (time needed to process or transmit one message) denoted as D. All queueing centers apply a first-come-first-served (FCFS) queueing policy. In this section, we define the individual queueing models that are used as part of the analytical or simulation queueing networks of Section 5.2. We introduce a set of variables in Table 5.1 related to our queueing models and their analysis.

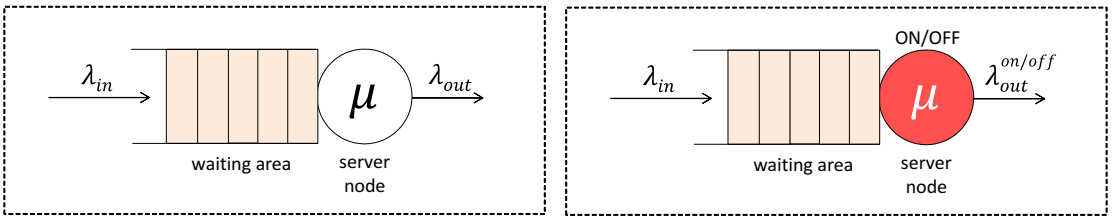
5.1.1 Continuous Queueing Center

This queueing center models uninterrupted serving (transmission, reception or processing) of messages as part of an end-to-end IoT interaction. It corresponds to the most common M/M/1 queue (see Fig. 5.2a), featuring Poisson arrivals and exponential service times.

An M/M/1 queueing center ($q_{\text{m/m/1}}$) is defined by the tuple:

$$q_{\text{m/m/1}} = (\lambda_{\text{in}}, \lambda_{\text{out}}, D) \tag{5.1}$$

where λ_{in} is the input rate of messages to the queueing center, λ_{out} is the output rate of messages, and D is the service demand for the processing of messages. Based on standard solutions for the



(a) Continuous queue.

(b) Intermittent (ON/OFF) queue.

Figure 5.2: Continuous and intermittent queues.

M/M/1 queue [179], the time that a message remains in the system (corresponding to queueing time + service time; we also call it response time) is given by:

$$R_{\text{in}}^{\text{m/m/1}} = \frac{D}{1 - \lambda_{\text{in}} D} \tag{5.2}$$

5.1.2 ON/OFF Queueing Center

To deal with the mobile peer's connections and disconnections we introduce the *Intermittent (ON/OFF)* queue (see Fig. 5.2b). The ON/OFF queueing center is depicted in Fig. 5.2b. Messages arrive according to a Poisson process with rate $\lambda_{\text{in}} > 0$, and are placed in a queue waiting to be "served" (waiting area in Fig. 5.2b). Messages are served with rate $\mu > 0$, which is exponentially distributed.

We assume that the server is subject to an on-off procedure. That said, it remains in the *ON*-state for exponential time with parameter θ_{ON} ($\theta_{\text{ON}} = 1/T_{\text{ON}}$), during which it serves messages (if any). Upon the expiration of this time the server enters the *OFF*-state during which it stops working (stops serving relevant messages) for an exponential distributed time period with rate θ_{OFF} ($\theta_{\text{OFF}} = 1/T_{\text{OFF}}$). Accordingly, an ON/OFF queueing center $q_{\text{on/off}}$ queueing center is defined by the tuple:

$$q_{\text{on/off}} = (\lambda_{\text{in}}, \lambda_{\text{out}}^{\text{on/off}}, D, T_{\text{ON}}, T_{\text{OFF}}) \tag{5.3}$$

where λ_{in} is the input rate of messages to the queueing center, $\lambda_{\text{out}}^{\text{on/off}}$ is the output rate of messages, and D is the service demand for the processing of messages (if any) during T_{ON}. The output process $\lambda_{\text{out}}^{\text{on/off}}$ is intermittent, because no message exits the queue during T_{OFF} intervals. Without loss of generality, we make the following assumption: if T_{ON} expires and there is a message currently being served, the server interrupts its processing and will continue in the next T_{ON} period.

Let $R_{\text{in}}^{on/off}$ be the the average response time (the time that a message remains in the system) for the $q_{\text{on/off}}$ queueing center. In the following, we elaborate an analytical solution for estimating $R_{\text{in}}^{on/off}$ based on the *mean value approach* [181]. This approach relies on common assumptions, such as: *i) the PASTA property*, where Poisson messages encounter the mean queue upon arrival [181]; *ii) the Memoryless property of the exponential distribution*, where the

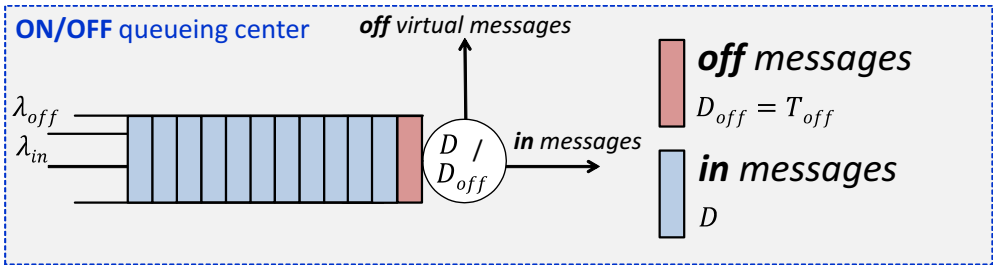


Figure 5.3: Two-class ON/OFF queueing center with preemptive priority.

expected time until completion of a message in service upon the arrival of a new message is equal to the service demand of the message in service; and *iii) Little's Law*. Additionally, we rely on queueing centers with multiple classes of customers and priority queueing. This means that the queuing center receives multiple arrival flows (classes), with each flow having its own distinct arrival rate and average service time. Moreover, these classes may have different priorities in getting served by the queueing center.

Let *in* be the class name of messages sent (*incoming*) to the ON/OFF queueing center, with λ_{in} input rate. We introduce – besides the incoming messages – virtual (additional) *off* messages that follow precisely the $T_{\text{ON}}/T_{\text{OFF}}$ timing of the server: an *off* (message) arrives at the server exactly when the latter goes into its T_{OFF} interval. Moreover, we represent the server's inactivity during the T_{OFF} interval by the service demand of the *off* message. More precisely, we assume that an *off* message arrives in the system exactly at the beginning of a T_{OFF} interval and has preemptive priority over regular messages. Hence, it also reaches the server at the beginning of the T_{OFF} interval.

Hence, we set the *off* messages to have mean virtual service demand equal to T_{OFF} (let service demand D_{off} equal to T_{OFF}). Additionally, we assume that incoming messages have mean actual service demand D. This modeling allows mapping our ON/OFF queueing center to one with continuous service. Hence, as depicted in Fig. 5.3, we specify our model as a two-class model (*in* and *off* messages) with preemptive priority [182]. To specify the arrival rate λ_{off} of messages of class off, we formulate the following theorem.

Theorem 1. *The average* λ_{off} *rate for the virtual* off *messages is given by:*

$$\lambda_{\text{off}} = \frac{1}{T_{\text{ON}} + T_{\text{OFF}}} \tag{5.4}$$

Proof. For an outside observer looking at the system at an arbitrary point in time, a new *off* message arrives at the beginning of a T_{OFF} interval and is served for T_{OFF}. During T_{OFF} there is no other *off* message arrival. At the end of the T_{OFF} interval, a new *off* message will arrive after T_{ON} time period. Based on the above:

$$\lambda_{\text{off}} = \begin{cases} 0, & \text{during } T_{\text{OFF}} \text{ intervals} \\ \frac{1}{T_{\text{ON}}}, & \text{during } T_{\text{ON}} \text{ intervals} \end{cases} \tag{5.5}$$

Hence, during T_{ON}, the λ_{off} flow is Poisson with exponentially distributed parameter T_{ON}. The average λ_{off} rate for both intervals is given by:

$$\begin{aligned}\lambda_{\mathrm{off}} &= \frac{T_{\mathrm{OFF}}}{T_{\mathrm{ON}}+T_{\mathrm{OFF}}}\ 0 + \frac{T_{\mathrm{ON}}}{T_{\mathrm{ON}}+T_{\mathrm{OFF}}}\frac{1}{T_{\mathrm{ON}}} \\ &= \frac{1}{T_{\mathrm{ON}}+T_{\mathrm{OFF}}}\end{aligned} \tag{5.6}$$

Note that the overall λ_{off} flow is not Poisson: during the T_{OFF} interval no new *off* message is allowed to arrive. □

The following theorem exploits the PASTA property, priority queueing, and Little's law in order to calculate response times for the $q_{\mathrm{on/off}}$ queueing center.

Theorem 2. *The average response time $R_{\mathrm{in}}^{on/off}$ for the $q_{\mathrm{on/off}}$ queueing center is given by:*

$$R_{\mathrm{in}}^{on/off} \;=\; \frac{\frac{T_{\mathrm{OFF}}^2}{T_{\mathrm{ON}}\ +\ T_{\mathrm{OFF}}}\ +\ D\frac{T_{\mathrm{ON}}\ +\ T_{\mathrm{OFF}}}{T_{\mathrm{ON}}}}{1\ -\ \lambda_{\mathrm{in}}D\frac{T_{\mathrm{ON}}\ +\ T_{\mathrm{OFF}}}{T_{\mathrm{ON}}}} \tag{5.7}$$

Proof. In our queueing center, the *off* class has *preemptive priority* over the class *in*. For such a model, a new arriving *off* message has to wait and be served for time:

$$R_{\mathrm{off}} \;=\; D_{\mathrm{off}} \;+\; Q_{\mathrm{off}}D_{\mathrm{off}} \tag{5.8}$$

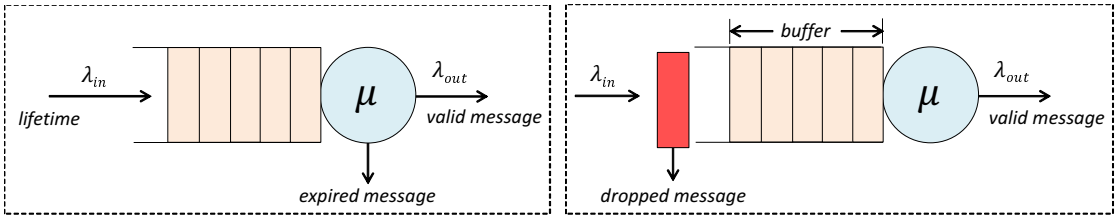
where Q_{off} is the number of the *off* messages present in the system. The *off* message has priority over the *in* messages and thus, it has to wait only for preceding *off* messages (if any). On the other hand, a new arriving *in* message has to wait and be served for time:

$$R_{\mathrm{in}} \;=\; D \;+\; Q_{\mathrm{in}}D \;+\; Q_{\mathrm{off}}^{\mathrm{pre}}D_{\mathrm{off}} \;+\; Q_{\mathrm{off}}^{\mathrm{post}}D_{\mathrm{off}} \tag{5.9}$$

In this case, despite the fact that a new *in* message has arrived, there is always the possibility that an *off* message can arrive. Thus, a message *in* must wait for any *off* and *in* class messages that are already in the system when it arrives, and any *off* class messages that arrive during the time that the *in* message is in the system. Let $Q_{\mathrm{off}}^{\mathrm{pre}}$ be the average number of *off* messages found in the system when the *in* message arrives and $Q_{\mathrm{off}}^{\mathrm{post}}$ be the average number of *off* messages that arrive in the system after the arrival of *in* and while it is present in the system.

Our model has some singularities we should take into account. More specifically, according to the Theorem 1 λ_{off} is not Poisson. Thus, the PASTA property does not hold and Q_{off} in eq. 5.8, encountered by a new arriving off message, is not the average Q_{off}. Nevertheless, we have already defined that there can be only one *off* message in the system. Thus, a new arriving *off* message sees $Q_{\mathrm{off}} = 0$, and based on the eq. 5.8 it has to wait for time:

$$R_{\mathrm{off}} \;=\; D_{\mathrm{off}} \tag{5.10}$$



(a) Applying lifetime periods to a queueing center. (b) Applying a buffer size to a queueing center.

Figure 5.4: Queues with message expirations and finite capacity.

On the other hand, *in* class arrivals are Poisson. Thus, in case of a new *in* arriving message, the PASTA property holds. Hence, by taking into account that during T_{OFF} there exists only one *off* message in the system and during T_{ON} none, the average $Q_{\text{off}}^{\text{pre}}$ number of *off* messages is given by:

$$\begin{aligned} Q_{\text{off}}^{\text{pre}} &= \frac{T_{\text{OFF}}}{T_{\text{ON}}+T_{\text{OFF}}}1 \;+\; \frac{T_{\text{ON}}}{T_{\text{ON}}+T_{\text{OFF}}}0 \\ &= \frac{T_{\text{OFF}}}{T_{\text{ON}}+T_{\text{OFF}}} \end{aligned} \tag{5.11}$$

Furthermore, since λ_{off} is different during T_{ON} and T_{OFF} (see eq. 5.5), we must express the average $Q_{\text{off}}^{\text{post}}$ number of *off* messages arriving during R_{in}, separately at each interval:

$$Q_{\text{off}}^{\text{post}} \;=\; R_{\text{in}}^{\text{T}_{\text{ON}}}\frac{1}{T_{\text{ON}}} \;+\; R_{\text{in}}^{\text{T}_{\text{OFF}}}0 \tag{5.12}$$

$R_{\text{in}}^{\text{T}_{\text{ON}}}$ is the portion of R_{in} that corresponds to T_{ON} and $R_{\text{in}}^{\text{T}_{\text{OFF}}}$ is the portion of R_{in} that corresponds to T_{OFF}. Based on the eq. 5.9:

$$R_{\text{in}}^{\text{T}_{\text{ON}}} \;=\; D \;+\; Q_{\text{in}}D \tag{5.13}$$

Finally, based on Little's law:

$$Q_{\text{in}} \;=\; \lambda_{\text{in}}R_{\text{in}} \tag{5.14}$$

Thus, based on equations 5.11, 5.12, 5.9 and 5.14, we use equation 5.9 to derive the response time R_{in}, which is denoted as $R_{in}^{on/off}$ for the *ON/OFF queueing center*. □

We note here that, besides the above solution for the response time of the ON/OFF queueing center, where we applied the mean value approach, we have also elaborated a second solution, in [10], where we represent the ON/OFF queueing center as a 2-D Markov chain and solve its global balance equations. Both solutions confirm Theorem 2.

5.1.3 Queueing Centers with Message Expirations and Finite Capacity

Up to now, we have defined queueing models having buffers with infinite capacity and arriving messages with infinite lifetime. This certainly affects the response time but also the rate of messages successfully served over the total number of arriving messages. However, modeling IoT

protocols with such characteristics may not be realistic. For instance, upon a long disconnection period (e.g., 30 mins) of an IoT sensor, the produced data/messages may exceed the sensor's buffer capacity and/or some of the oldest data may become obsolete for the receiving application/user. Accordingly, in this subsection we introduce the corresponding queueing model features that take into account the above constraints. These features can be applied on both continuous (M/M/1) and ON/OFF queues.

Queueing Center with Message Expirations

As depicted in Fig. 5.4a, messages arrive in the queue with λ_{in} input rate to be processed. An arriving message carries a `lifetime` period attributed to it upon its creation and properly updated at each processing stage, which represents the message validity inside the queueing center. The validity of a message is checked at the queue's *head* prior to its processing, using a related condition. Let t_0 be the timestamp at which the message arrives in the queue and t_1 be the timestamp at which the message reaches the head of the queue. Then, each message is characterized based on the following condition:

$$if \begin{cases} t_1 - t_0 > residual\_lifetime : message \leftarrow expired \\ t_1 - t_0 \leq residual\_lifetime : message \leftarrow valid \end{cases} \tag{5.15}$$

We assume that `lifetime` checking makes part of the regular service demand of a valid message and introduces only a negligible overhead. Expired messages exit the queueing center without regular processing.

Queueing Center with Finite Capacity

This is a well known queueing model feature, where a specific `buffer` size is applied to the queueing center that ensures having max. `buffer` messages in the queue, including the one in service. This prevents from storing too many messages for too long in devices with limited hardware capacity (memory, hard disk). In particular, as depicted in Fig. 5.4b messages arrive in the queue with λ_{in}. Before a message enters the queue the following condition is checked: $new\_queue\_size + message\_in\_service > buffer$. If the condition is true, the message is *dropped*. Otherwise, the message enters the queue to be processed.

Based on the literature, an M/M/1 queue with finite capacity is notated as M/M/1/k, where `buffer` = k - 1. In our models, we represent both M/M/1 and ON/OFF queues with finite capacity by adding `buffer` size to the corresponding definition (Eq. 5.1 and 5.3).

We use the above queueing centers as part of larger queueing networks in order to represent and evaluate the application and middleware layer of mobile peers. Estimated response times can be derived by using both our analytical and simulated models after composing larger queueing networks. Nevertheless, we do not provide analytical solutions for queueing centers with messages expiration and finite capacity. Though, the above queueing models can be applied to our

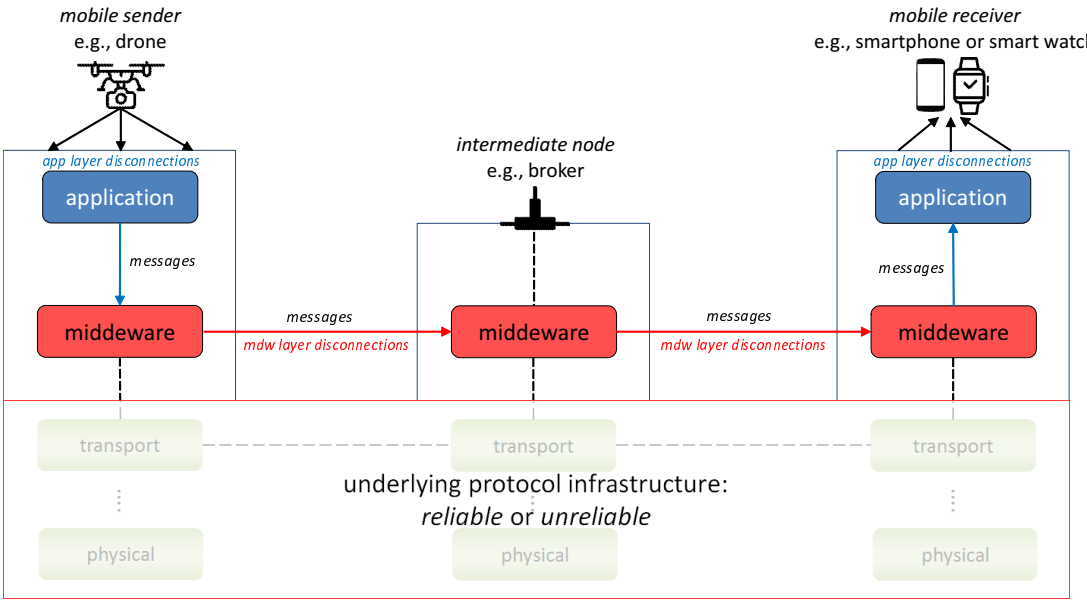


Figure 5.5: Middleware interaction model with the underlying infrastructure.

simulation models and the trade-off between delivery success rates and response times can be evaluated.

5.2 Performance Models for the Core Communication Styles

In Chapter 3, we defined four communication styles: *i)* CS with peers: client, server; *ii)* PS with peers: publisher, subscriber; and intermediate node: broker; *iii)* DS style with peers: consumer, producer; and *iv)* TS with peers: reader/taker, writer; and intermediate node: tuplespace (tspace). Furthermore, by using the above styles we identified four possible interaction types: *i)* one-way; *ii)* two-way sync; *iii)* two-way async; and *iv)* two-way stream. We defined in detail the above interactions through the provision of their functional semantics, APIs and sequence diagrams.

Middleware protocols usually follow one of the above communication styles and implement one or more types of interactions (see Chapter 3). Depending on the communication style they follow, response times and delivery success rates may differ, as showed our simulation-based analysis of GM interactions in Chapter 4. However, in that analysis we ignored the effect of the end-to-end protocol infrastructure on these QoS metrics, and in particular the end-to-end communication reliability in the face of disconnections; essentially, we assumed reliable communication. In this section, we enrich our models and analysis with such concerns. For instance, let us consider the interaction of two mobile peers, as depicted in Fig. 5.5; we demonstrate in the following the effect of disconnections to QoS. In this interaction, a *mobile sender* (e.g., a drone) produces messages through multiple applications (apps). Each app disconnects from the network from time to time (e.g., for energy saving purposes), and the produced messages are

buffered until the next connection, upon which they are forwarded to the middleware layer (mdw layer). At the other side, a *mobile receiver* (e.g., a smartphone or smart watch) is able to receive messages from multiple senders; the messages are distributed to multiple apps, whenever each app's connectivity (app layer connection) allows it.

The mdw layer is responsible to handle the incoming messages and transmits them via the underlying network. Inside the network, additional disconnections may occur (defined as *middleware disconnections*) due to several reasons: *i)* broken session of the underlying protocol; *ii)* router crash/reboot; *ii)* wireless devices moving out of range; etc. Based on the above, message transmission may fail at the app or mdw layer. To enhance reliability, middleware protocols either rely on the underlying protocol mechanisms, or they introduce additional mechanisms, or they even incorporate middleware *intermediate nodes* (e.g., broker), mainly to decouple the mobile peers. Thus, existing protocols can be categorized into: *i)* **unreliable** protocols, where guarantees for the delivery of messages are missing; and *ii)* **reliable** protocols, where the delivery of each message is verified. In the following, we introduce our assumptions for modeling unreliable and reliable middleware protocols by using our queueing models of Section 5.1.

Unreliable Protocols

Unreliable middleware protocols typically build on top of the UDP unreliable transport protocol (e.g., CoAP non-confirmable). In UDP, two middleware nodes do not set up an end-to-end connection (sender → intermediate-node, or intermediate-node → receiver, or sender → receiver without intermediate-node), as it is the case in TCP. Additionally, there is no confirmation for message delivery, and hence no message re-transmission in case of unsuccessful delivery. Thus, a **sent message may not be received**.

To model the message transmission of such protocols, we use an intermittent queue (ON/OFF, see Fig. 5.2b) at the app-layer of the mobile sender representing its connectivity (e.g., voluntary disconnection). At the mdw layer we use continuous queueing centers (e.g., M/M/1, see Fig. 5.2a) with losses at the exit, for the processing of messages regardless of the middleware/-mobile receiver's connection or disconnection. Either the message is successfully transmitted (in the former case) or it is lost (in the latter case).

Reliable Protocols

Reliable middleware protocols usually build on top of the TCP reliable transport protocol [183]. Over TCP, two middleware nodes set-up or shut-down a reliable end-to-end connection (sender → intermediate-node, or intermediate-node → receiver, or sender → receiver without intermediate-node) via 3-way or 4-way handshake, respectively. Based on TCP's mechanisms, **the delivery of each message is verified** using ACKs, timeouts and retransmissions. After the initial 3-way handshake, a session between the peers starts. During the session, intermediate routers

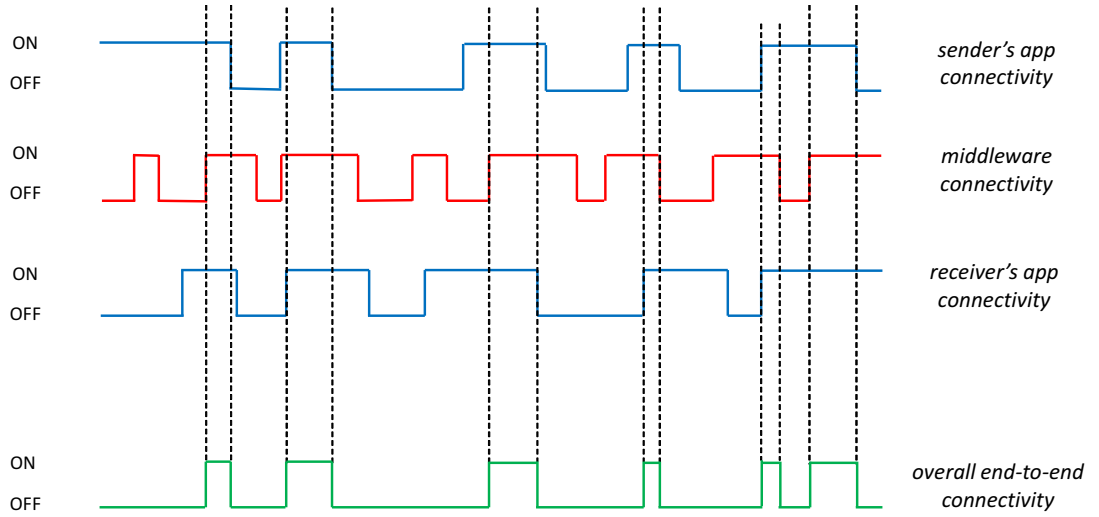


Figure 5.6: Overall end-to-end connectivity pattern.

can crash and reboot, wireless disconnections can occur, servers may shut down, etc, and thus the session may break. There are several ways to detect such dropped connections in order to re-establish a TCP session. For instance, when sending out data without receiving ACKs; after several seconds the sender considers the receiver is down and terminates the connection. In a different approach, some reliable protocols build on top of UDP (e.g., CoAP confirmable) and add their own reliability mechanisms through additional acknowledgments (ACKs) or negative-acknowledgments (NACKs). Whether on top of TCP or on top of UDP, different levels of QoS may be provided (e.g., with MQTT, which adds its own reliability mechanisms on top of TCP).

To model the message transmission of reliable protocols, we use an intermittent queue at the app layer of the mobile sender. To represent the end-to-end established session, we apply at the ON/OFF queue an overall connectivity pattern between the sender and the receiver. Determining such an overall pattern requires to take the intersection of the connectivity patterns of: *i)* the sender's app; *ii)* the receiver's app; and *iii)* the middleware. We illustrate this in Fig. 5.6. At the mdw layer we use continuous queueing centers to represent simply the processing/transmission times of messages (end-to-end connectivity is represented by the above ON/OFF queueing center and there are no message losses).

By following the above assumptions, we introduce in this section performance models using QNMs for both unreliable/reliable middleware interactions. In particular, we call these models *performance modeling patterns* (PerfMP), as they can be reused inside bigger compositions modeling end-to-end performance of systems. We list our patterns in Table 5.2. They provide solutions for the interactions types found in the core CS, PS, DS, and TS communication styles.

Moreover, to enable simulation-based analysis of these patterns (see Section 5.4), we have developed in Java the MobileJINQS open source simulator, which builds on top of the JINQS

Communication style	**Interaction type**	**Description**	**Pattern**
CS	`one-way`	Models the interaction of messages sent from a server to a client using QNMs	Fig. 5.7
	`two-way sync`	Models the synchronous request-response interaction between a client and a server using QNMs	Fig. 5.8
	`two-way async`	Models the asynchronous request-response interaction between a client and a server using QNMs	Fig. 5.9
PS	`one-way`	Models the interaction of messages sent from a publisher to a subscriber through a broker using QNMs	Fig. 5.10
	`two-way stream`	Models the subscription-response interaction between a subscriber and a broker using QNMs	Fig. 5.11
DS	`one-way`	Models the stream of messages sent from a producer to a consumer using QNMs	Fig. 5.7
	`two-way stream`	Models the streaming interaction between a consumer and a producer using QNMs	Fig. 5.9
TS	`one-way`	Models the messages sent from a writer to a tuple space using QNMs	Fig. 5.12
	`two-way sync`	Models the synchronous template-response interaction between a reader/taker and a tuple space using QNMs	Fig. 5.13

Table 5.2: Performance modeling patterns.

simulation library for multi-class queueing networks [184]. The full description, a user guide and the code scripts for some of the patterns can be found in the Appendix C.

Pattern 1: CS/DS one-way Interaction

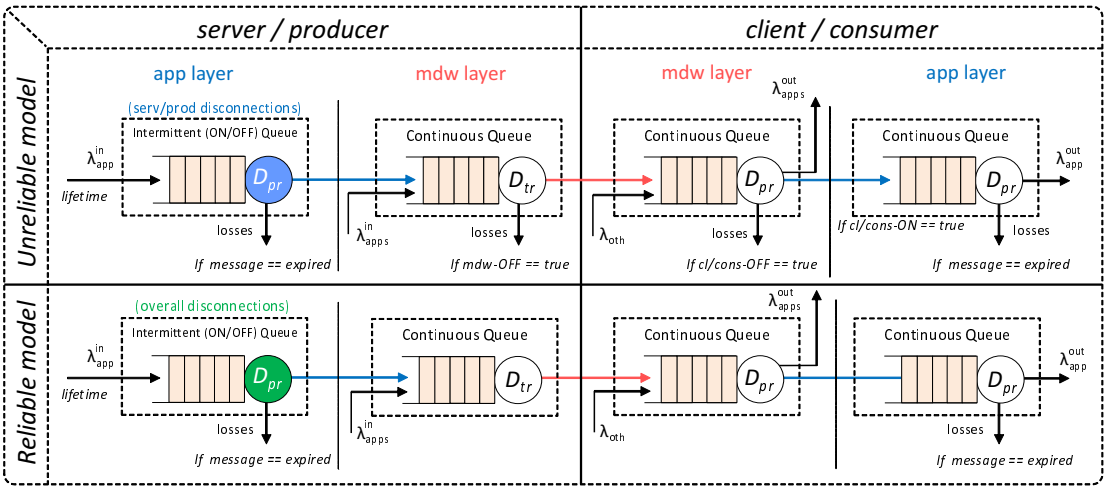


Figure 5.7: PerfMP for CS/DS one-way interactions.

Characteristics

- Push-based notification
- Transmission of streaming data

Example. A server sends a notification message to a client (e.g., a push notification to a smartphone). In another example, a producer sends a stream of data to a consumer. Regarding the latter, we assume that the required stream session is already established between the consumer and the producer.

Description. The *CS/DS one-way* pattern is depicted in Fig. 5.7; it is used to model a reliable/unreliable Client/Server or Data Streaming one-way interaction. Such a model evaluates the end-to-end response time and message delivery success rate for this interaction. For both the reliable/unreliable models, multiple apps produce messages at the server's/producer's side (app layer). Each app may be disconnected (e.g., for energy saving purposes) and until its next connection the produced messages are buffered in an ON/OFF queueing center. For any produced message a `lifetime` period is applied, which represents the message validity inside the queueing network. Let $\lambda_{\text{app}}^{in}$ be the input rate of messages to the app's ON/OFF queueing center. The server's/producer's mdw layer accepts messages from the specific app and from multiple other apps. Let $\lambda_{\text{apps}}^{in}$ be the input rate of messages from other apps to the mdw layer.

Regarding the unreliable model, the server's/producer's mdw layer does not verify the successful transmission of messages to the client/consumer. Messages are sent continuously without any knowledge of the disconnections of the mdw or the client/consumer, may result in losses. Hence, a message transmission is modeled using a continuous queueing center at the

server's/producer's mdw layer, where the applied service demand D_{tr} represents the delay inside the network. Finally, additional message losses may occur due to message expirations at the app layer.

On the other hand, in the reliable model, the app's ON/OFF queueing center applies the overall end-to-end connectivity pattern (see Fig. 5.6). Thus, the app layer transmits messages to the mdw as soon as an end-to-end connection (between the server/producer and the client/-consumer) is established. Similarly to the unreliable model, a message transmission is modeled using a continuous queueing center at the server's/producer's mdw layer. Nevertheless, in the reliable model the message reception is verified and, hence, message losses occur only in case of message expirations.

Finally, at the client's/consumer's side, messages arrive to the mdw layer through a continuous queueing center. These messages may arrive from several servers/producers (see the additional flow λ_{oth}) and be destined to multiple apps. Finally, let $\lambda_{\text{app}}^{out}$ be the flow destined to the client/consumer of interest. For the client's/consumer's app, a continuous queueing center is used to process messages and detect possible message expirations; its service demand (D_{pr}) represents the app's processing time.

IoT protocols. The CS/DS one-way pattern can be used to model several IoT protocols. DPWS, OPC UA, REST, XMPP, Websockets support CS/DS one-way interactions [59, 60]. As already defined in Chapter 2, in these protocols there are not any built-in reliability features since they rely on TCP's delivery mechanisms. Such protocols can be modeled using the reliable pattern where the overall end-to-end connectivity follows TCP's sessions. Moreover DPWS, OPC UA and XMPP use some additional mechanisms at the server's mdw layer, which are used to observe the availability of the client resource. Thus, the detection of end-to-end disconnections is more accurate, and TCP message re-transmissions leading to the detection of broken sessions are avoided by using several techniques applied to the TCP protocol (e.g., keep-alive messages) [183]. Also in these cases, the reliable pattern can be used.

On the other hand, CoAP transmits messages over the unreliable UDP protocol. However it supports two built-in features: "non-confirmable" and "confirmable". The "non-confirmable" can be modeled using the unreliable pattern since does not guarantee any message delivery. The "confirmable" feature supports message re-transmissions using ACKs and NACKs and thus, it can be modeled using the reliable pattern.

Pattern 2: CS two-way sync Interaction

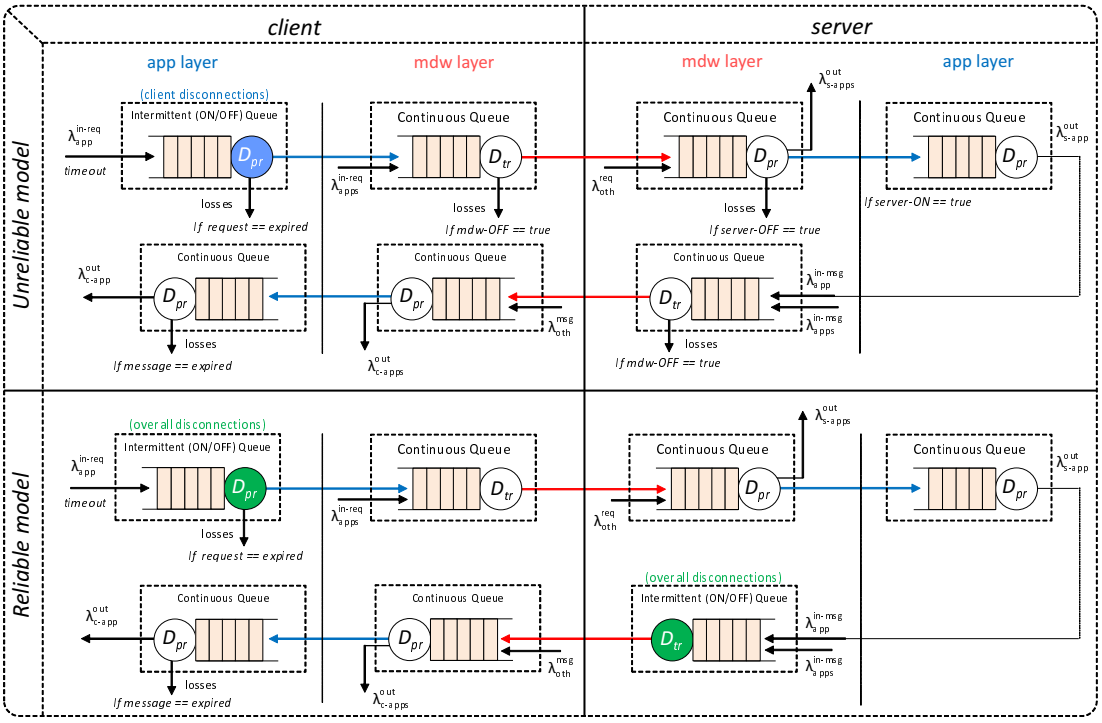


Figure 5.8: PerfMP for CS two-way synchronous interactions.

Characteristics

- Request-response interaction
- Synchronous interaction

Example. A client sends a request to a server and the server has to reply back within a `timeout` period (e.g., a smartphone that requests values regarding the air quality from a mobile air sensor).

Description. The *CS two-way sync* pattern is depicted in Fig. 5.8; it is used to model a reliable/unreliable Client/Server synchronous interaction. Such a model evaluates the end-to-end response time and message delivery success rate of a request since is sent from an app, until the app receives the response from the requested server. A typical synchronous interaction must completed within a `timeout` period and the client's process is blocked till the completion.

For both the reliable/unreliable models, multiple apps produce requests on the client's side (app layer). Based on the applied intermittent connectivity behavior, requests are buffered in an ON/OFF queueing center; during the connected state (ON) requests are forwarded to the mdw layer. Let $\lambda_{\text{app}}^{in-req}$ be the input rate of requests to the app's ON/OFF queueing center. Since the interaction is synchronous, after sending a request the client's app must remain connected to receive the response for `timeout` period. Hence, for any produced request a `timeout` period

is applied. Furthermore, the client's mdw layer accepts requests with rate $\lambda_{\text{apps}}^{in-req}$ from multiple other apps. Regarding the unreliable model, similar to *Pattern 1*, a request transmission is modeled using a continuous queueing center at the client's mdw layer. Request losses occur due to middleware disconnections. On the other hand, in the reliable model, the app's ON/OFF queueing center applies the overall end-to-end connectivity behavior and the mdw layer transmits requests via a continuous queueing center.

At the server's side, requests arrive to the mdw layer through a continuous queueing center along with requests from other clients (flow $\lambda_{\text{oth}}^{req}$) and be destined to multiple apps. Let $\lambda_{\text{s-app}}^{out}$ be the flow destined to the server of interest. This flow is processed at the app layer via an additional continuous queueing center. Since the interaction is synchronous, the server's app must remain connected and process the request promptly in order to provide the response (message) to its mdw layer. Let $\lambda_{\text{app}}^{in-msg}$ be the rate of responses from the corresponding app. Similarly, other apps provide messages to be transmitted with rate $\lambda_{\text{apps}}^{in-msg}$ to the server's mdw layer. The mdw layer transmits back the messages through a continuous queueing center (unreliable model) or an ON/OFF (reliable model).

Finally at the client's side, the app is blocked waiting its responses from the corresponding server and from multiple other servers with rate $\lambda_{\text{oth}}^{msg}$. To distribute the incoming messages to multiple apps, a continuous queueing center is used. Furthermore, an additional continuous queueing center is used to detect possible message expirations for the client of interest (see flow $\lambda_{\text{c-app}}^{out}$).

IoT protocols. The CS two-way sync pattern can be used to model several IoT protocols. DPWS, OPC UA, REST and CoAP support CS two-way sync interactions [59, 60]. As already defined in *Pattern 1*, DPWS, OPC UA, REST, Websockets and CoAP confirmable can be modeled using the reliable model and CoAP non-confirmable with the unreliable.

Pattern 3: CS two-way async/stream Interactions

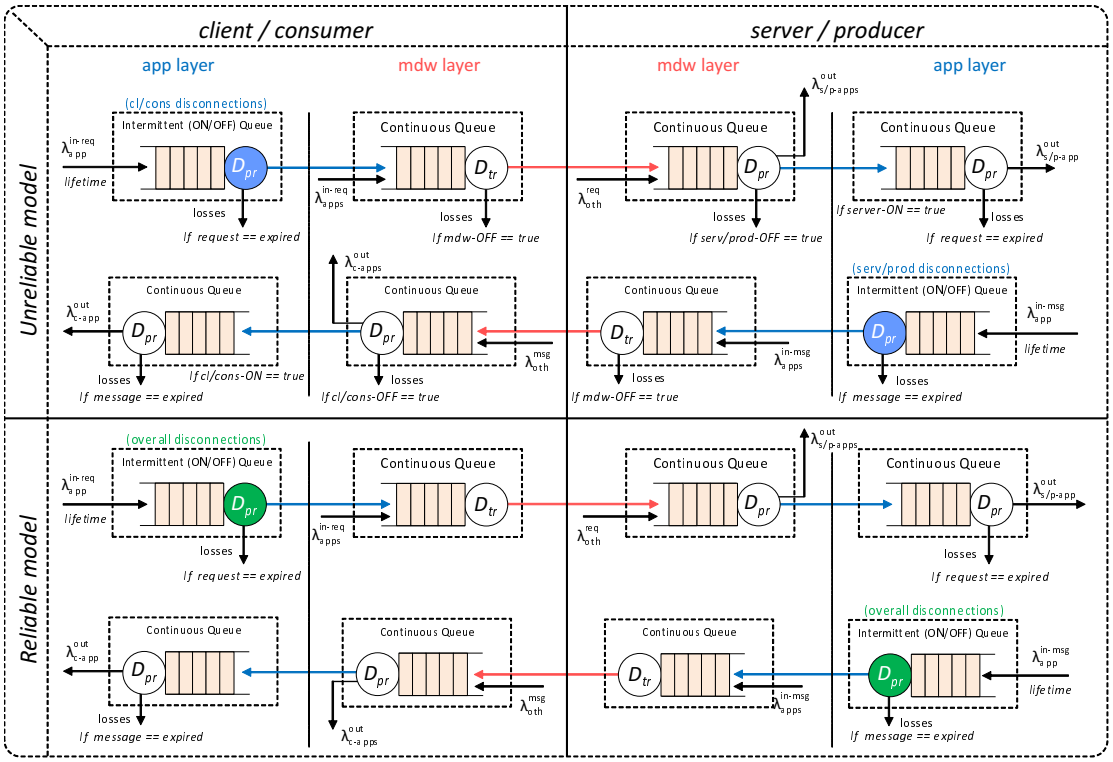


Figure 5.9: PerfMP for CS two-way async/stream interactions.

Characteristics

- Request-response interaction
- Asynchronous or Streaming interaction

Example. A client sends a request to a server and resume its processing without waiting for a response. The server handles the request and returns the response at some point later where the client receives it and proceeds with its processing. In another example, a consumer requests to open a stream session with a producer. As long as the session is established, the producer transmits a flow of messages in arbitrary moments to the consumer. For instance, a smartphone that requests video streaming data.

Description. The *CS two-way async/stream* pattern is depicted in Fig. 5.9; it is used to model a reliable/unreliable Client/Server asynchronous or Data Streaming interaction. Such a model evaluates the end-to-end response time and message delivery success rate of: *i)* requests sent from the client's/consumer's app to the server's/produce's app; and *ii)* messages sent from the server's/produce's app to the client's/consumer's app. We model this interaction using approximately the same queueing network as in *Pattern 2*.

In particular, in pattern 3, the interaction is asynchronous. Hence, the client's/consumer's

app can disconnect after producing a request. Similarly, at the server's/producer's side, each app does not block its processing for providing the response(s) (i.e. it can receive the request, disconnect and provide the response(s) at some point later while being connected or disconnected). For any produced request or response a `lifetime` is applied. In comparison to *Pattern 2*, an ON/OFF queueing center is used at the server's/producer's side of each app to buffer the response(s) and finally forward them to the mdw layer. In the unreliable model, the ON/OFF queueing center adopts the server's/producer's connectivity behavior and in the reliable model the overall end-to-end behavior. Finally, the mdw layer transmits requests and messages by using continuous queueing centers for both reliable and unreliable models.

IoT protocols. The CS two-way async/stream pattern can be used to model several IoT protocols. DPWS, OPC UA, REST, and CoAP support CS two-way async interactions [59, 60]. Furthermore, Websockets support streaming interactions [59]. As already defined in *Pattern 1*, DPWS, OPC UA, REST, Websockets and CoAP confirmable can be modeled using the reliable model and CoAP non-confirmable with the unreliable.

Pattern 4: PS one-way Interaction

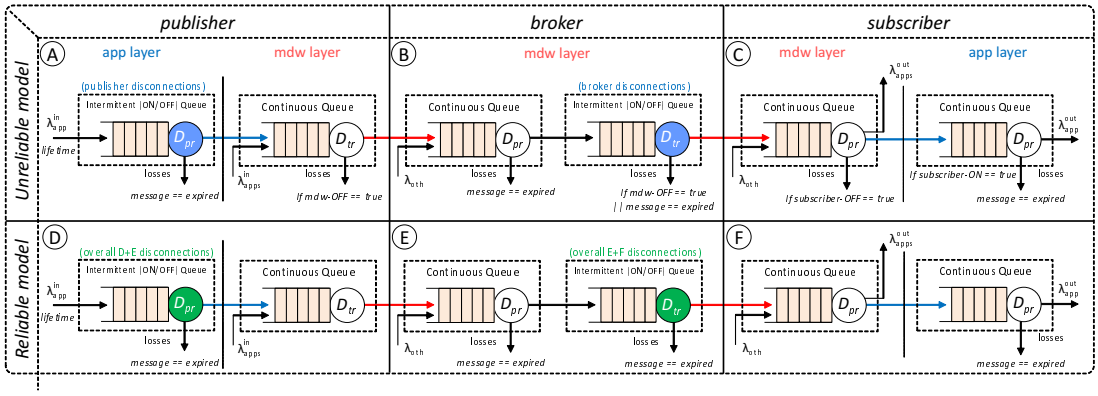


Figure 5.10: PerfMP for PS one-way interactions.

Characteristics

– Publish/subscribe end-to-end interaction

Example. A publisher publishes messages to the broker and the subscriber receives them (through the broker). We assume that the subscriber is already subscribed to a specific `filter` in a broker and the publisher publishes messages characterized by the same `filter`. For instance, a reporter posting news for a football team, which are received by another user.

Description. The *PS one-way pattern* is depicted in Fig. 5.10; it is used to model a reliable/unreliable Publish/Subscribe or Data Streaming one-way interaction. Such a model evaluates the end-to-end response time and message delivery success rate of messages, since they are sent from

the publisher's app, then they are received by the broker and are forwarded to the subscriber, and finally they are received by the subscriber.

At the publisher's side (this is similar to the server's/producer's side in *Pattern 1*), for the reliable or the unreliable models, the app produces messages with rate $\lambda^{in}_{\text{app}}$. For any produced message a `lifetime` period is applied. Messages are buffered in an ON/OFF queueing center which represents, respectively, the publisher's or the overall end-to-end connectivity between the publisher and the broker (publisher's app, middleware, and broker's connectivity). During the connected state (ON) messages are forwarded to the mdw layer which may receive messages from multiple other apps with rate $\lambda^{in}_{\text{apps}}$. For both the reliable/unreliable models, the mdw layer transmits messages through a continuous queueing center. In the unreliable model losses occur due to middleware disconnections or message expirations. In the reliable model losses may occur only due to message expirations.

At the broker's side, for both the reliable/unreliable models, messages arrive to the mdw layer through a continuous queueing center. These messages may arrive from multiple publishers (see the additional flow λ_{oth}). Dropping of messages occur at the exit of the broker's input queue, depending on the subscriptions or due to message expirations (based on the `lifetime` period). In case a message is not dropped, it is forwarded to an output queue for its transmission to the corresponding subscriber. In the unreliable model, the transmission of messages to the subscriber is done through an ON/OFF queueing center, which represents the broker's app-layer disconnections. This is the case where the broker is deployed in a mobile device and the transmission of messages must done based on the device's disconnections. Nevertheless, losses may occur due to middleware disconnections or message expirations. In the reliable model, the transmission is done through an ON/OFF queueing center, which represents the overall end-to-end connectivity between the broker and the subscriber (broker's, middleware, and subscriber's app connectivity), and losses may occur only due to message expirations.

Finally, at the subscriber's side, messages arrive to the mdw layer through a continuous queueing center. These messages may arrive from multiple other publishers (see the additional flow λ_{oth}) and be destined to multiple apps. Let $\lambda^{out}_{\text{app}}$ be the flow destined to the subscriber of interest. For the subscriber's app, a continuous queueing center is used to process messages and detect possible message expirations.

IoT protocols. The *PS one-way* pattern can be used to model several IoT protocols and messaging technologies. AMQP [54] and MQTT [53, 59] support PS one-way interactions. AMQP and MQTT rely on TCP's delivery mechanisms and introduce additional built-in features for the end-to-end (from the publisher to the subscriber) message delivery such as "fire and forget" or "at most once" (*QoS level 0*), "at least once" (*QoS level 1*) and "exactly once" (*QoS level 2*). We model AMQP, MQTT and their built-in QoS features using the reliable model. Tools such as RabbitMQ [119] and Kafka [120] are implementations of these protocols.

JMS [51] has been one of most successful asynchronous messaging technology available. JMS defines the API for building messaging systems, but it is not a messaging protocol like MQTT and AMQP. Nevertheless, it uses several underlying messaging protocols in order to be language independent. Using the JMS API a subscriber can be defined as "non-durable" or "durable". For a non-durable subscriber, message losses occur upon its disconnections (mdw or app layer disconnections). Thus, the *broker-subscriber* link is unreliable and it can be evaluated using (B) - (C) queueing centers in Fig. 5.10. On the other hand, when a durable subscriber is disconnected, messages are kept at the broker. Thus, the *broker-subscriber* link is reliable and it can be evaluated using (E) - (F) queueing centers in Fig. 5.10.

Pattern 5: PS two-way stream Interaction

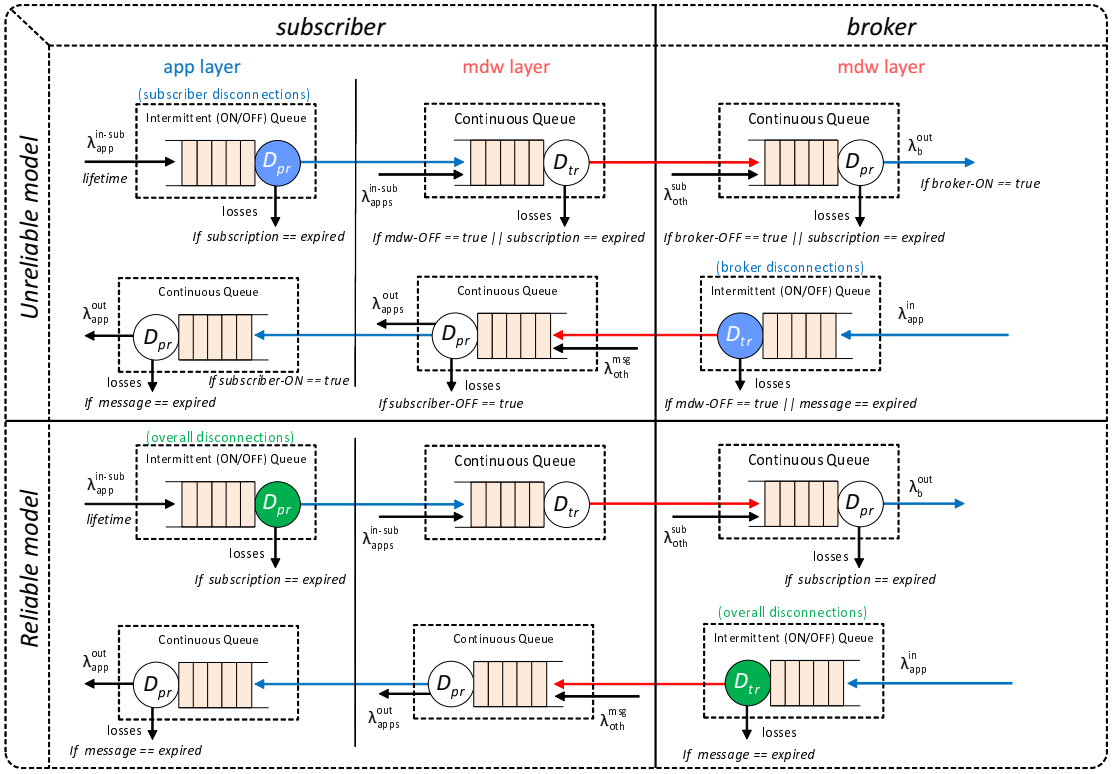


Figure 5.11: PerfMP for PS two-way stream interactions.

Characteristics

- Publish/subscribe two-way asynchronous interaction
- Streaming interaction

Example. A subscriber sends a subscription to the broker to receive messages that match a specific `filter`. As soon as the messages arrive to the broker and match the `filter`, are forwarded to the subscriber. For instance, a mobile user subscribes to receive notifications regarding her favorite football team.

Description. The *PS two-way stream* pattern is depicted in Fig. 5.11; it is used to model a reliable/unreliable Publish/Subscribe two-way streaming interaction. Such a model evaluates the end-to-end response time and message delivery success rate of: *i)* subscriptions sent from the subscribers's app to the broker; and *ii)* messages sent from the broker to the subscriber's app.

At the subscriber's side, the app produces subscriptions with rate $\lambda_{\text{app}}^{in-sub}$ which are buffered in an ON/OFF queueing center that represents the subscriber's app connectivity in the unreliable model and the overall end-to-end (subscriber - broker) connectivity in the reliable. For any produced subscription a `lifetime` period is applied, which represents the message validity inside the queueing network. The mdw layer may receive subscriptions from multiple other apps with rate $\lambda_{\text{apps}}^{in-sub}$ and transmits them through a continuous queueing center. Subscription losses may occur due to middleware disconnections and expirations in the unreliable model, and only due to subscription expirations in the reliable model.

The broker receives subscriptions in an input continuous queueing center and are maintained in an up-to-date list. Additional subscriptions arrive from other subscribers with rate $\lambda_{\text{oth}}^{sub}$. The broker matches messages with the corresponding subscription and forwards a copy of each message to its mdw layer to be delivered to the subscriber of interest with rate $\lambda_{\text{app}}^{in}$. Subsequently, the mdw layer transmits the messages through an ON/OFF queueing center which represents the broker's disconnections (unreliable); or through an ON/OFF queueing center which represents the overall end-to-end connectivity between the broker and the subscriber (reliable).

Finally, at the subscriber's side, messages arrive to the mdw layer through a continuous queueing center, along with messages from other brokers (see flow λ_{oth}) and be destined to multiple apps. Let $\lambda_{\text{app}}^{out}$ be the flow destined to the subscriber of interest. For the subscriber's app, a continuous queueing center is used to process messages and detect possible message expirations.

IoT protocols. The PS two-way stream pattern can be used to model several IoT protocols. MQTT and AMQP support PS two-way stream interactions [59]. As already defined in *Pattern 4*, these protocols support several QoS features which can be modeled using the reliable models. Tools and APIs such as RabbitMQ [119], Kafka [120] and JMS [51] are implementations of such protocols.

Pattern 6: TS one-way Interaction

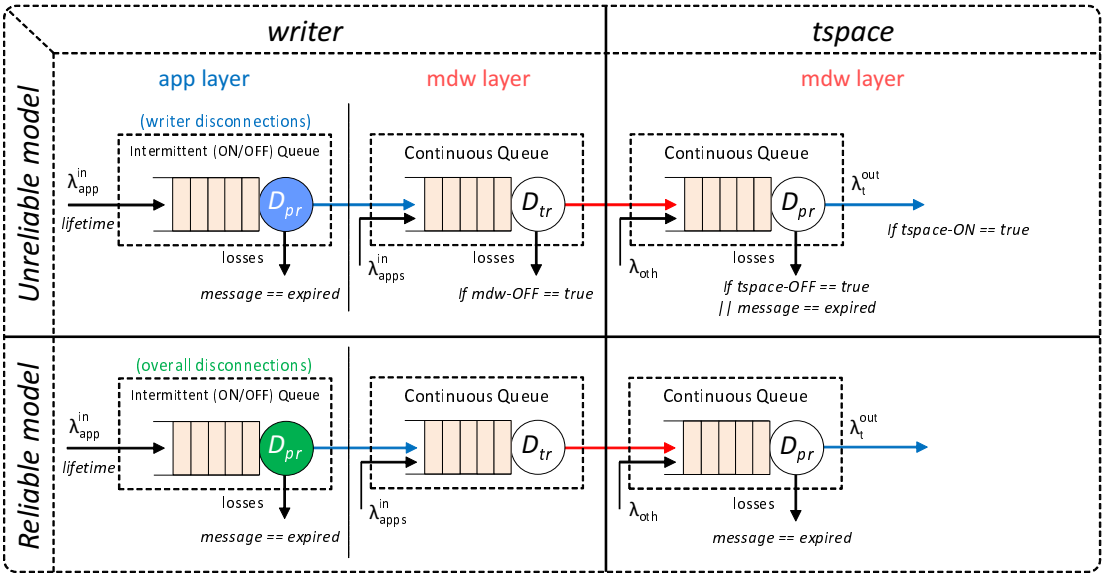


Figure 5.12: PerfMP for TS one-way interactions.

Characteristics

– Storing data to a shared data-space

Example. A writer writes a message to the shared data-space (`tspace`). For instance, a smartphone transmits in the tspace data related to the noise level of a city. Messages remain to the tspace until some readers/takers read/take them. The latter represents a two-way interaction, which is evaluated with *Pattern 7*.

Description. The *TS one-way pattern* is depicted in Fig. 5.12; it is used to model a reliable/unreliable Tuple Space one-way interaction. Such a model evaluates the end-to-end response time and message delivery success rate of messages sent from the writer's app, until are received by the tspace.

Similarly to the modeling of a publisher in *Pattern 4*, the writer's app produce messages with rate $\lambda_{\text{app}}^{in}$ that are buffered in an ON/OFF queueing center (which represents the writer's app connectivity in the unreliable model and the overall end-to-end connectivity in the reliable), to be forwarded to its mdw layer. For any produced message a `lifetime` period is applied. The mdw layer may receive messages from multiple other apps with rate $\lambda_{\text{apps}}^{in}$ through a continuous queueing center. Message losses occur due to middleware disconnections and message expirations in the unreliable model, and only due to message expirations in the reliable model.

At the tspace's side, messages arrive to the mdw layer through a continuous queueing center, along with messages from other writers with rate λ_{oth}, which are saved to memory in order to be taken/read from multiple readers/takers.

IoT protocols. The *TS one-way* pattern can be used to model several protocols. Especially for IoT application, SemiSpace [55] is a light weight implementation, inspired by the JavaSpaces [58] middleware protocol. Alternative light weight implementations include GigaSpaces [121], Terrastore [122] and Lime [123]. The above protocols do not provide any QoS built-in features and they rely on the transport protocol's delivery mechanisms. Thus, we can evaluate their one-way interactions using our unreliable/reliable models.

Pattern 7: TS two-way sync Interaction

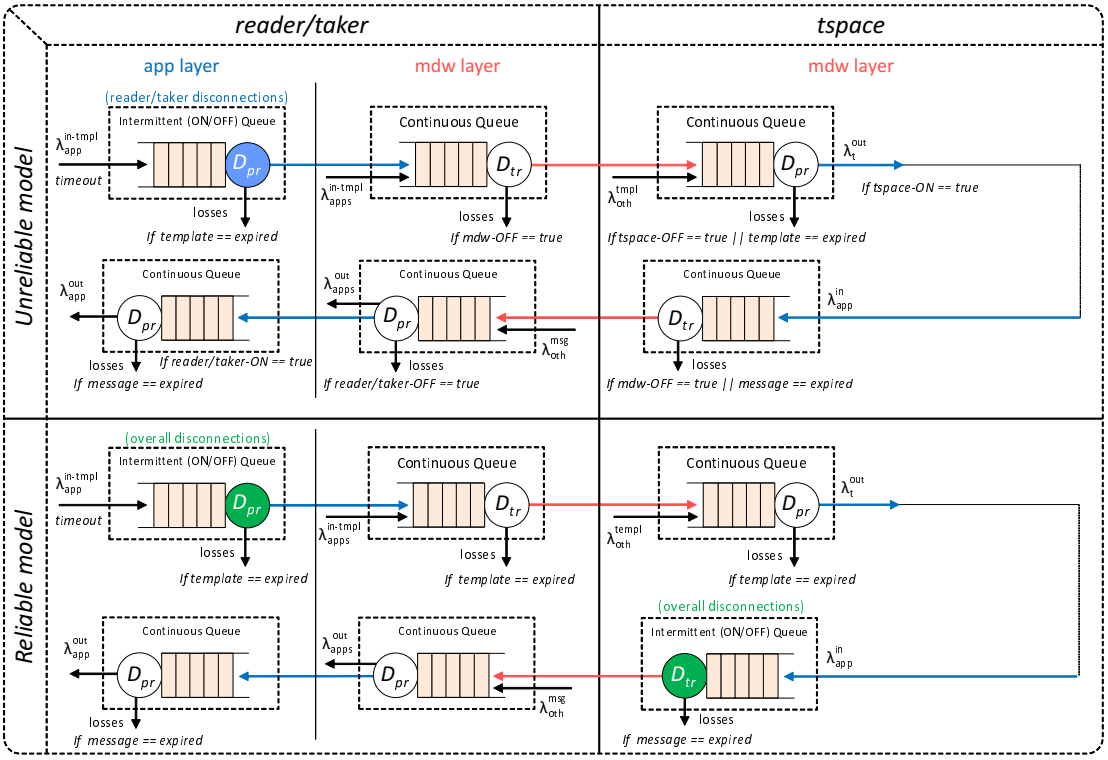


Figure 5.13: PerfMP for TS two-way sync interactions.

Characteristics

- Shared data-space interactions
- Synchronous interaction

Example. A reader/taker sends a template to a tuple space (tspace) and the tspace has to reply back within a `timeout` period. The tspace replies back a message matching the reader's/taker's template. For instance, a smartphone that requests the air quality from a shared memory for a specific area.

Description. The *TS two-way sync* pattern is depicted in Fig. 5.13; it is used to model a reliable/unreliable Tuple Space synchronous interaction. Such a model evaluates the end-to-end response time and message delivery success rate of a `template` since is sent form the

reader's/taker's app, until the app receives the response from the requested tspace. For each app, the response must be returned within a `timeout` period (synchronous interaction).

Similarly to the client's models in *Pattern 2*, the reader's/taker's app produce templates with rate $\lambda_{\text{app}}^{in-templ}$ that are buffered in an ON/OFF queueing center to be forwarded to its mdw layer. The mdw layer receives templates from multiple other apps with rate $\lambda_{\text{apps}}^{in-templ}$. After sending a template the reader's/taker's app must remain connected to receive the response at least for `timeout` period. Hence, for any produced template a `timeout` period is applied. The mdw layer transmits the templates through a continuous queueing center (with message losses due to middleware disconnections and message expirations in the unreliable model, and only due to message expirations in the reliable model).

At the tspace's side, templates arrive through a continuous queueing center. Additional templates arrive from other reader's/taker's with rate $\lambda_{\text{oth}}^{templ}$. For each template, a matching process follows at the output of the queueing center, in order to forward at the mdw layer the response. Subsequently, the mdw layer transmits the responses through a continuous queueing center (unreliable) or through an ON/OFF queueing center (reliable) which represents the overall end-to-end connectivity. Message losses occur at the exit of the queue due to message expirations (messages are maintained in the tspace for a `lifetime` period).

Finally, at the reader's/taker's side the app is blocked waiting its response at the mdw layer. Other responses may arrive from multiple other tspaces with rate $\lambda_{\text{oth}}^{msg}$. To distribute the received messages to multiple apps, a continuous queueing center is utilized. To represent the processing time and detect possible message expirations, an additional continuous queueing center is used at the reader's/taker's app layer.

IoT protocols. The TS two-way sync pattern can be used to model several IoT protocols. SemiSpace [55], GigaSpaces [121], Terrastore [122] and Lime [123] support PS two-way sync interactions. As already defined in *Pattern 6*, these protocols can be evaluated using our unreliable/reliable models.

5.3 End-to-end Performance Modeling

In this section we leverage the patterns introduced in the previous section to model the end-to-end performance of Things that: *i)* employ heterogeneous middleware protocols and reliability mechanisms; and *ii)* operate in large scale.

5.3.1 Interoperability Setup

In the previous section we introduced several performance modeling patterns for both unreliable/reliable middleware interactions among mobile Things. However, we assumed homogeneous interactions among Things following the same communication style and reliability mechanism. For instance, two mobile Things employing the CoAP CS middleware protocol interacting with each other in a two-way asynchronous manner. Nevertheless, IoT applications often include multiple interconnections between heterogeneous peers (Things employing various protocols with different QoS semantics). In such cases, to enable an end-to-end interaction between two heterogeneous Things, it is essential to map their functional and QoS semantics. For instance, to enable the interaction between an MQTT publisher and a REST server it is essential to: *i)* identify and map their supported interaction types; *ii)* map the MQTT topics with the REST resources; *iii)* map their app-layer timing behavior and *iv)* identify and map their performance patterns with regard to their unreliable/reliable infrastructure.

Chapters 3 and 4 deal with the first three steps. In this section we deal with the forth step, by composing the performance models of such end-to-end interconnections. Fig. 3.11 of Section 3.3, depicts the end-to-end interaction between the `vehicle-device` and the `estimation-service` of the TIM system. Based on this interconnection, the two Things employ MQTT and REST middleware protocols, respectively, and they interact with each other through the CoAP bus protocol. To enable such an interconnection, two BCs must be synthesized for connecting each middleware protocol to the bus protocol. This interconnection is represented in the figure with three GM connectors where each one abstracts a specific middleware protocol. More specifically: *i)* each `vehicle-device` produces notifications with the MQTT "fire-and-forget" delivery mode; *ii)* the `estimation-service` receives the notifications through the REST interface; *iii)* the bus protocol (CoAP) receives "non-confirmable" traffic-notifications. Therefore, "connector A" must implement an MQTT one-way interaction, "connector B" must implement a CoAP one-way interaction, and "connector C" a REST one-way interaction.

To model the performance of end-to-end heterogeneous interconnections, we utilize the performance patterns defined in the previous section. Particularly, we correspond to each GM connector a specific performance pattern by taking into account the interaction type it follows as well as the middleware protocol (along with the protocol's QoS semantics) it employs. The resulting end-to-end model, is the composition of multiple patterns – one for each GM connector from the end-to-end interconnection. Hence, the end-to-end performance model of the

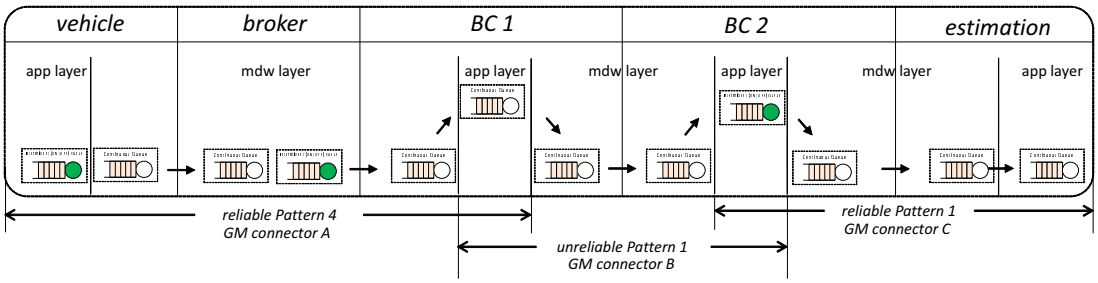


Figure 5.14: End-to-end queueing network for the `vehicle-device` → `estimation-service` interconnection.

`vehicle-device` → `estimation-service` interconnection, is composed through the following patterns:

reliable Pattern 4 → *unreliable Pattern 1* → *reliable Pattern 1*

When composing two patterns, we merge the app layers of the patterns – when existent – at the point of the interconnection. If neither app layer exists, we create a new one. The merged or created app layer corresponds to the conversion logic of the deployed BC that enables the interconnection. This app layer takes into account the constituent app layers (continuous or ON/OFF queues) but also the fact that the BC is considered as always connected. This is desirable in order to provide seamless interoperability and can be achieved, e.g., by the deployment of BCs in the Cloud. Hence, to create a queueing network that evaluates the performance of the `vehicle-device` → `estimation-service` interconnection, a system designer should operate as follows:

1. Identify the proper GM connector of each middleware protocol and the protocols' QoS semantics (e.g., "non-confirmable").
2. Select the performance patterns that correspond to "GM connector A", "GM connector B", "GM connector C".
3. Connect the patterns as follows: "GM connector A" → "GM connector B" → "GM connector C".

The patterns of each GM connector are defined as follows:

GM connector A: models the interaction between the mobile sender (`vehicle-device`) and BC 1 through the *reliable Pattern 4*. The app-layer queueing center at the receiver's side (BC 1) must be merged with the sender's side app-layer queueing center of the GM connector B. This queueing center represents the conversion logic of BC 1.

GM connector B: models the interaction between the two BCs through the *unreliable Pattern 1*. Since the selected pattern is unreliable and BCs must be always connected, we use a continuous queueing center to model the conversion logic of BC 1. The app-layer queueing center at the receiver's side (BC 2) must be merged with the sender's side app-layer queueing center of the GM connector C. This queueing center represents the conversion logic of BC 2.

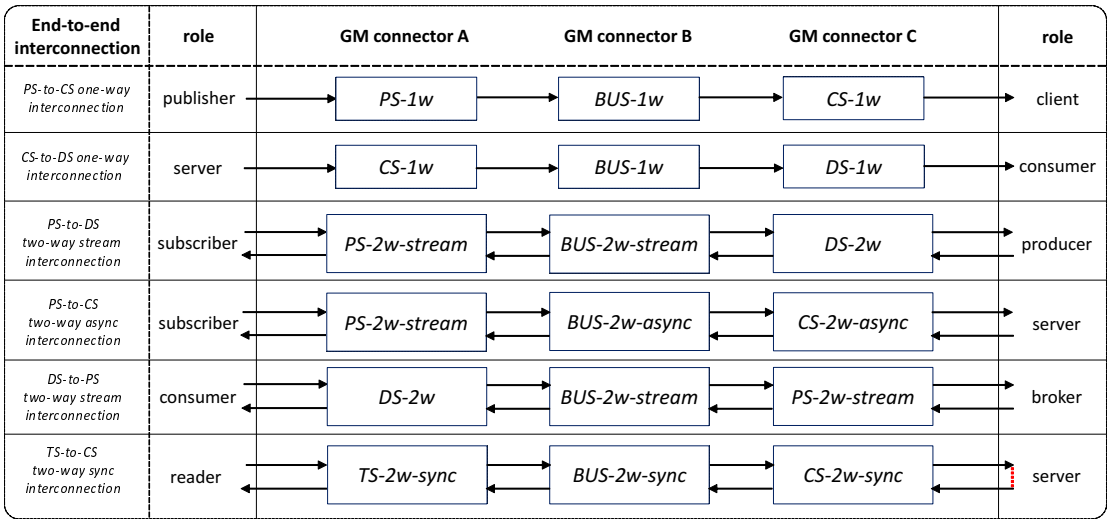


Figure 5.15: End-to-end queueing networks for several interconnections between different communication styles.

GM connector C: models the interaction between BC 2 and the receiver (`estimation-service`) through the *reliable Pattern 1*. Since the selected pattern is reliable and BCs must be always connected, we use an ON/OFF queueing center to model the conversion logic of BC 2. The ON/OFF queueing center represents the overall end-to-end connectivity between BC 2 and the `estimation-service`.

By following the above steps, the system designer creates the one-way end-to-end queueing network of Fig. 5.14. The queueing centers that represent the BCs' conversion logic, can be parameterized based on the required time for the conversion between specific protocols (e.g., MQTT to CoAP). We provide such times in Section 3.4 of Chapter 3; they depend on the message size as well.

By following the above approach we introduce end-to-end performance models for several interconnections between different communication styles as depicted in Fig. 5.15. Below we present in detail the combination of the selected performance patterns that represent the end-to-end interconnection (i.e., "GM connector A" + "GM connector B" + "GM connector C"):

- from *PS publisher* to *CS client* one-way: Pattern 4 + Pattern 1 + Pattern 1;
- from *CS server* to *DS consumer* one-way: Pattern 1; + Pattern 1/4 + Pattern 1;
- from *PS subscriber* to *DS producer* two-way stream: Pattern 5 + Pattern 3 + Pattern 3;
- from *PS subscriber* to *CS server* two-way async: Pattern 5 + Pattern 3 + Pattern 3;
- from *DS consumer* to *PS broker* two-way stream: Pattern 3 + Pattern 3 + Pattern 5;
- from *TS reader* to *CS server* two-way sync: Pattern 7 + Pattern 2 + Pattern 2;

To select either a reliable or an unreliable pattern for the above interconnections, the system designer must be aware of the QoS semantics of the utilized middleware protocols in the specific IoT application.

5.3.2 Wide-scale Setup

As an example of an IoT system comprising distributed peers that span a wide-area, we discuss in this section a large-scale PS system. For such deployment, the PS system is implemented as a set of independent, communicating brokers, forming a broker overlay. Based on [185], in such architectures, peers can access the system through any broker that becomes their *home broker*. Particularly, subscribers subscribe their interest (to specific types of events) to their home broker and through the *subscription partitioning process*, subscriptions are spread to a subset of existing brokers. Then, publishers produce events characterized each by specific types (e.g., a topic) to their home broker and the subset of corresponding subscribers is determined through the *matching process*. Finally, the produced events are delivered to all the determined subscribers by using the *event routing process*. This process is performed by using several algorithms, such as selective routing or event gossiping.

In [11], we model the end-to-end performance from publishers to subscribers interacting through a network of brokers. Specifically, we leverage the queueing centers presented in Section 5.1 to estimate through formal, analytical and simulation models the end-to-end response time from a publisher to a subscriber. This is done by following the message routing path taken via the broker overlay network and analyzing the rates and service demands at each station. We show the efficacy of work by studying multiple application scenarios and considering peers' disconnections due to: *i)* network issues; *ii)* voluntary reasons; and *iii)* degraded network. More details regarding the wide-scale performance evaluation can be found in [11].

5.4 Assessment

In this section, we present a simulator that implements the queueing models and performance patterns presented in Sections 5.1 and 5.2. We leverage our simulator for analytical model validation as well as for creating simulations models that represent our proposed performance patterns. Using such models, we perform statistical analysis and evaluate the tradeoff between response times and delivery success rates.

Our simulator, MobileJINQS[1], is an open-source library for building simulations encompassing constraints of mobile IoT applications. MobileJINQS is an extension of JINQS, a Java simulation library for multiclass queueing networks [184]. JINQS provides a suite of primitives that allow developers to rapidly build simulations for a wide range of QNMs [179].

MobileJINQS retains the generic model specification power of JINQS, while it provides additional features of interest to mobile or other systems such as the possibilities: *i)* `lifetime` limitations at each message entering a queueing network, *ii)* intermittently available (ON/OFF) queue servers representing the intermittent connectivity of mobile peers; *iii)* arrival and con-

[1] xsb.inria.fr/d4d#mobilejinqs

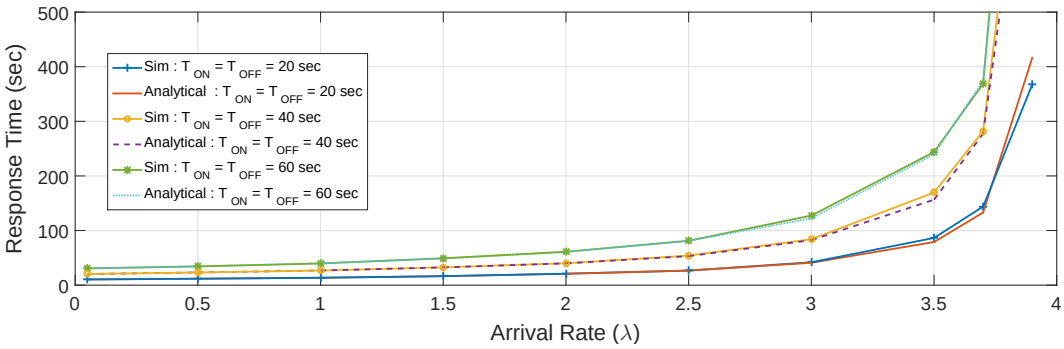


Figure 5.16: Analytical vs. simulated response times at the ON/OFF queueing center.

nectivity rates derived from real traces; *iv)* ON/OFF queues with buffers of finite capacity; and *v)* multiple sink nodes that collect lost messages due to middleware disconnections or message expirations. The implementation details of the above extensions can be found in the appendix C.

In what follows, we leverage MobileJINQS to validate the analytical model of the ON/OFF queueing center.

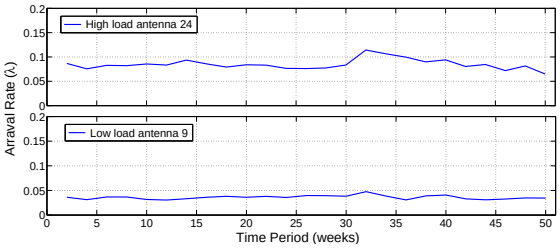
5.4.1 ON/OFF Queueing Center Validation using Real Traces

In order to validate our model, we use our simulator which can be parameterized using well-known probability distributions and actual data derived from a real setup (real traces).

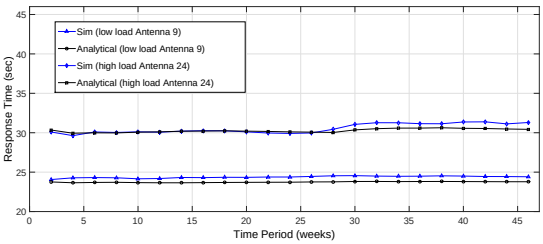
Analytical vs. Simulated Response Time

We utilize *MobileJINQS* to implement the *ON/OFF queueing center* described in subsection 5.1.2. Mobile peers connect and disconnect in the scale of seconds/minutes to send/receive messages, depending on the application context. To represent such behavior, we set the ON/OFF system parameters as follows: *i)* the server remains in the *ON* and *OFF* states for exponentially distributed time periods $T_{\text{ON}} = T_{\text{OFF}} = 20/40/60$ sec, thus, the server changes its state every 20, 40 and 60 sec; *ii)* messages are processed with a mean service demand $D = 0.125$ sec; *iii)* there is sufficient buffer capacity so that no messages are dropped; and *iv)* messages arrive to the queue with a mean rate varying from 0.05 to less than 4 messages per sec.

By applying λ rates equal or greater than 4 messages/sec, the system saturates. Using the above settings in our simulator, we run the system and derive the simulated curve of the mean response time for several λ rates as depicted in Fig. 5.16. The analytical results obtained by the Theorem 2 and depicted also in Fig. 5.16, show the high accuracy of Theorem 2. For a service center where its server is always *ON*, the system does not saturate if $\lambda D < 1$ (see Eq. 5.2). However, for the *ON/OFF queueing center* the system does not saturate if $\lambda D \frac{T_{\text{ON}}+T_{\text{OFF}}}{T_{\text{ON}}} < 1$ as indicated by the denominator of (see Eq. 5.7). Thus, this confirms that $\lambda_{\text{sat}} = 4$ messages/sec for this example. By comparing the curves for the simulated and analytical response times, we notice some small differences for rates equal to or higher than 3.5 messages/sec. This is



(a) Low and high load arrival rates derived from real traces.



(b) Analytical vs. simulated response times using arrival rates from real traces.

Figure 5.17: Validation using antenna real traces.

acceptable, since the system is close to saturation at these rates.

Validation using Arrival Rates from Real Traces

In order to further validate our ON/OFF queueing center, we parameterize it using input workloads derived from real traces. The real data, named the *D4D* dataset, was provided to us by Orange Labs in the context of the *D4D* challenge[2]. The *D4D* dataset contains Call Detail Records (CDRs) of users that are subscribed to the Sonatel Telecom mobile operator in Senegal. This data was collected for the whole country over a period of 50 weeks from 7 January 2013 until 23 December 2013. More details about our analysis of the *D4D* dataset and the way we have leveraged it to model the performance of large-scale mobile pub/sub systems can be found in our recent work in [12].

For our validation we used the antenna *traces*. An antenna trace reflects the number of calls made or SMS sent by many mobile users associated to this antenna for each 10 min interval, over the period of 50 weeks. We assume that user access patterns to mobile communication services (antennas) can also be used to represent user access to mobile IoT application services. In particular, for parameterizing the *ON/OFF queueing center*, we map the number of calls or SMS per 10 min interval at the selected antenna to an equal number of messages arriving over the same time interval. Based on [12], this mapping results in a *non-homogeneous Poisson process (or input flow)*, defined as λ_{in}, with rate parameter $\lambda_{(\text{t})}$ piecewise constant in each interval $t \in T$:

$$\lambda_{(\text{t})} = \frac{N_i^t}{|t|} \tag{5.16}$$

where T is the 50-week period, $|t|$ equals to 10 min, and N_i^t is the number of messages sent for each 10 min interval at a given antenna i.

Thus, in order to calculate the rate of the input flow (λ_{in}) for each 10 min interval over the 50-week period at a given antenna i, we use the eq. 5.16. Subsequently we use these rates to parameterize the *ON/OFF queueing center*. To perform our experiments with representative traces, we selected input flows from a *low load antenna* and a *high load antenna*. Fig. 5.17a depicts two antennas used for our experiments: *i)* antenna 9 has a low load input flow with

[2]http://www.d4d.orange.com/en/Accueil

overall average rate of 0.04; and *ii)* antenna 24 has a high load input flow with overall average rate of 0.075.

To perform simulations and compare the results with our analytical model, we extend the *MobileJINQS* simulator by enabling the application of non-homogeneous input flows to the *ON/OFF queueing center*. Thus, we set the system as follows: *i)* the server remains in the *ON* and *OFF* states for exponentially distributed time periods with parameters $\theta_{\text{ON}} = \theta_{\text{OFF}} = 0.025$ (i.e. $T_{\text{ON}} = T_{\text{OFF}} = 40$ sec); *ii)* messages are served with a mean service demand $D = 1$ sec; and *iii)* messages arrive to the queue with variable λ rate for each 10 min interval, based on the loads of antennas 9 and 24 (Fig. 5.17a). By running the system with the above settings we derive the simulated curves of the mean response times for the input flows of antennas 9 and 24 over the 50-week period, as depicted in Fig. 5.17b. The mean response times regarding the overall period are 24 and 31 sec, correspondingly for the two antennas.

Subsequently, we apply the same parameter values to the equation of Theorem 2 for each 10 min interval and we calculate the mean response times over the 50-week period as depicted in Fig. 5.17b through the analytical curves. The mean response times regarding the overall period are 23 and 30 sec, correspondingly for each antenna. Note that we selected these parameter values with respect to our condition ($\lambda_{(\text{t})} D \frac{T_{\text{ON}}+T_{\text{OFF}}}{T_{\text{ON}}} < 1$), in order to avoid the saturation of the *ON/OFF queueing center*. By comparing the curves for the simulated and analytical response times, we notice that the absolute deviation between the two is no more than 5% (approximately 1 sec).

It is worth noting that using the load of antenna 24, the mean response time is much higher in comparison to the one of antenna 9 (7 sec difference). In this case, to get a lower mean response time, an application developer should set the system to process faster the messages sent (lower service demand D) of antenna trace 24. In a way similar to our validation above, antenna traces can be leveraged for system capacity planning for IoT applications. However, performing simulations leads to high development and computational cost in order to obtain accurate results. Therefore, our analytical model can be a useful tool for evaluating the performance of IoT applications.

Validation using Connectivity Rates from Real Traces

In another validation of our *ON/OFF queueing center*, we parameterize it using connectivity data derived from real traces. In particular, we have collected data concerning the actual connections and disconnections in the metro. Towards this, we have developed an android application, named *Metro Cognition*[3], related to network connectivity data for metro passengers in Paris and Delhi. Metro Cognition collects connectivity tuples using the Android *BroadcastReceiver* while the user is traveling. Let `con_tuple` be the connectivity tuple with the following 4 elements:

[3]https://goo.gl/x6vuoB

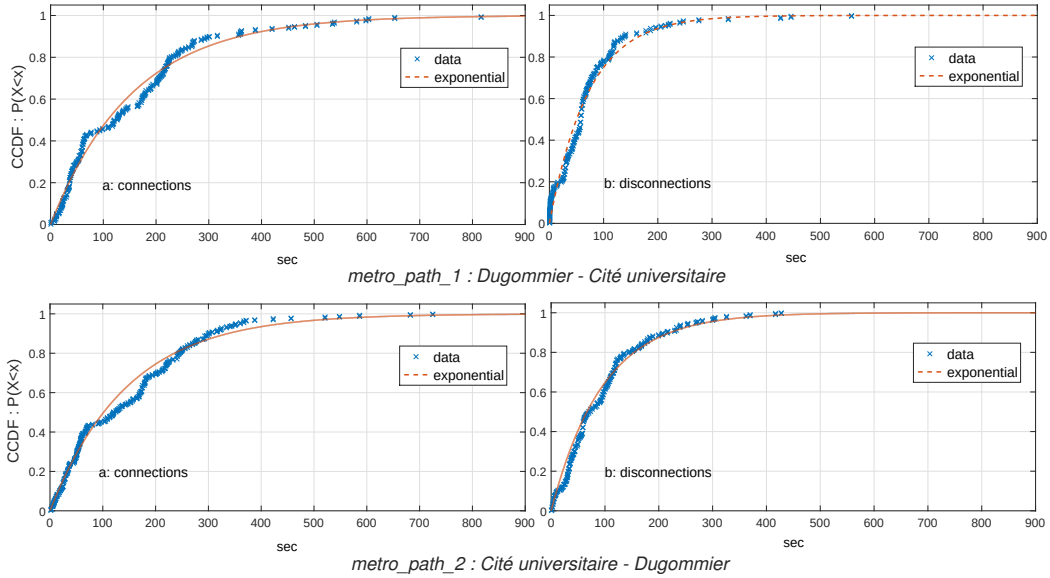


Figure 5.18: CCDF of connections and disconnections.

i) ON/OFF (qualifying the availability of the Internet connection); *ii) timestamp* (the exact time when the connectivity status ON/OFF is captured); *iii) mobile operator* (e.g. Vodafone) and; *vi) metro_path_id* (a unique identifier that corresponds to a specific path inside the metro system – e.g., metro station 1 → metro station 2).

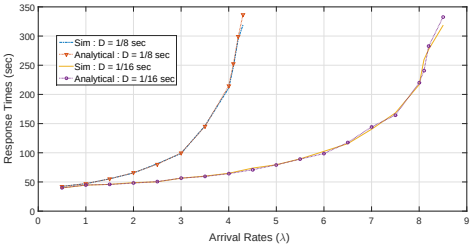
We also define a `con_pattern`, which consists of many `con_tuple`s. Each `con_pattern` is created as follows: *i)* the user starts the application and chooses the path between two metro stations; *ii)* every 30 seconds and additionally each time the connectivity status changes a tuple (`con_tuple`) is created; and *iii)* when the user's journey ends, the background service stops, the data are stored in JSON format and are sent to the Cloud server *GoFlow*[4]. In [13], we initiated the creation of a dataset related to network connectivity data for metro travelers in Paris.

To utilize our dataset for the validation of the ON/OFF queueing center, we concatenate all the `con_patterns` for each *metro_path_id*. So far, we have collected sufficient amount of data for the following metro paths:

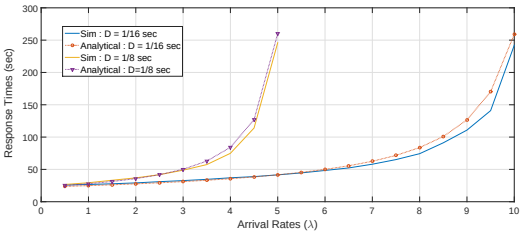
- **metro_path_1**: metro station "Cité Universitaire" → metro station "Dugommier"; *journeys*: 34; *total duration*: 15.18 hours; *average duration journey*: 26.8 min.
- **metro_path_2**: metro station "Dugommier" → metro station "Cité Universitaire"; *journeys*: 28; *total duration*: 12.13 hours; *average duration journey*: 26 min.

Our data[5] show that metro travelers lose and recover network connection for intervals ranging from several seconds to 5 minutes, maximum. In average, connected intervals are 1.5 longer larger than the disconnected intervals. As a first step, in our analysis, we specify the best

[4]https://goflow.ambientic.mobi
[5]https://goo.gl/1SBiaU



(a) Response time while traveling in metro path: Cité Universitaire - Dugommier.



(b) Response time while traveling in metro path: Dugommier - Cité Universitaire.

Figure 5.19: Validation using connectivity real traces.

fit of our data to existing probability distributions by applying the same method as in [134]. Fig. 5.18 shows the complementary cumulative distribution functions (CCDFs) of connection and disconnection intervals for each of the above metro paths. It is interesting to observe that our traces fit best with exponential distributions. More specifically, the measured statistics fit very well with 146.38 (T_{ON}) and 96 (T_{OFF}) seconds in average for connections and disconnections while traveling in metro_path_1. For metro_path_2, the average connection time is 155.88 sec (T_{ON}), and the average disconnection is 72.15 sec (T_{OFF}).

By relying on the above and using our simulator, we perform analysis of our ON/OFF queueing center with connectivity data generated from exponential distributions with parameters $T_{\text{ON}} = 146.38$ sec and $T_{\text{OFF}} = 96$ sec corresponding to metro_path_1. Our analysis provides the response times (latencies) of messages when sent and received by metro travelers. Fig. 5.19a, compares the mean response times between the analytical model and the model-based simulation when applying the above connectivity parameters, various arrival rates, and various service times. We notice that the results match with high accuracy. When applying a service time of $D = 0.125$ sec, the response time becomes too high for λ rates greater than 3.5 messages/sec. To tune the system for providing better response times, messages should be processed faster. Thus, by applying a service time of $D = 0.0625$ sec, the response time is too high for λ rates greater than 7 messages/sec.

To confirm the above analysis, we apply, in a second simulation experiment, directly the derived ON/OFF intervals from the real traces concerning metro_path_2. We use the same setup as previous and apply to the analytical model the average connected and disconnected intervals ($T_{\text{ON}} = 155.88$ sec and $T_{\text{OFF}} = 72.15$ sec). Fig. 5.19b compares the response times between the analytical model and the trace-based simulation. Results match with high accuracy for low arrival rates. For higher rates, there is a quite good match between the two with a maximum difference of about 10%. This is still acceptable accuracy for the analytical model, given that it relies on the assumption of exponential distributions for ON/OFF intervals, which is an approximation for the ON/OFF intervals from the real traces. In our future work, we intend to continue the collection of data to perform experiments in several other paths of the metro in Paris.

5.4.2 End-to-end Performance Evaluation

In this subsection, we perform analyses of delivery success rates and response times of mobile IoT interactions by relying on our performance modeling patterns described in Section 5.2. We develop simulation models which are parameterized with a variety of `service demands`, T_{ON}, T_{OFF} and `lifetime` periods. We demonstrate that varying these periods has a significant effect on the rate of successful interactions. Furthermore, the trade-off involved between success rates and response times is also evaluated. System designers can follow our approach to analyze homogeneous interactions among Things. In a similar way, they can further compose performance patterns in order to evaluate interconnections between heterogeneous Things. The code scripts of our performance patterns are provided in the Appendix C.

Evaluation of Reliable Pattern 4

In Section 5.2, we defined the pattern that models the performance of PS one-way interactions by incorporating the queueing models defined in Section 5.1. The resulting end-to-end queueing network, named *Pattern 4*, is depicted in Fig. 5.10. We select the *reliable* Pattern 4 for our experimental setup where losses occur only due to message expirations – i.e., there are no losses due to disconnections.

At the input of the pattern's queueing network, messages arrive with rate $\lambda_{\text{app}}^{in} = 2$ messages/sec (publishing rate). Each message is valid for a deterministic `lifetime` period and then discarded by the queueing network. We alternate between values of 10, 20 and 30 sec for the `lifetime` parameter. Arrival rates from/to multiple other apps going through the various queueing centers at the middleware layer are isolated; we assume that they have already been taken into account in the utilization of the servers of the queueing centers.

We parameterize the queueing network as follows: as already defined, the app layer's ON/OFF queueing center represents the overall end-to-end connectivity between the publisher and the broker. We set the total average connected + disconnected period to be $T_{\text{ON}}^{\text{pub}} + T_{\text{OFF}}^{\text{pub}} = 80$ sec. Experiments are performed by varying the $T_{\text{ON}}^{\text{pub}}$ and $T_{\text{OFF}}^{\text{pub}}$ periods inside the 80 sec interval. To process the produced messages and forward them to the mdw layer when connected, we apply a service rate of $\mu_{\text{pr}} = 64$ messages/sec. The applied service demand is very low, since the app layer's queue is used locally only to forward messages to the mdw layer. To transmit messages to the broker, we apply a service rate of $\mu_{\text{tr}} = 32$ messages/sec. The applied service demand can vary, depending on the bandwidth of the connection between the publisher and the broker. To process the incoming messages at the broker's side, we apply (at the continuous queue) a service rate of $\mu_{\text{pr}} = 64$ messages/sec.

At the subscriber's side, we consider that peers remain always ON for receiving the subscribed messages. However, middleware-layer disconnections may occur. Connection/disconnection periods depend on the type of user mobility. For example, these periods differ for pedestrians,

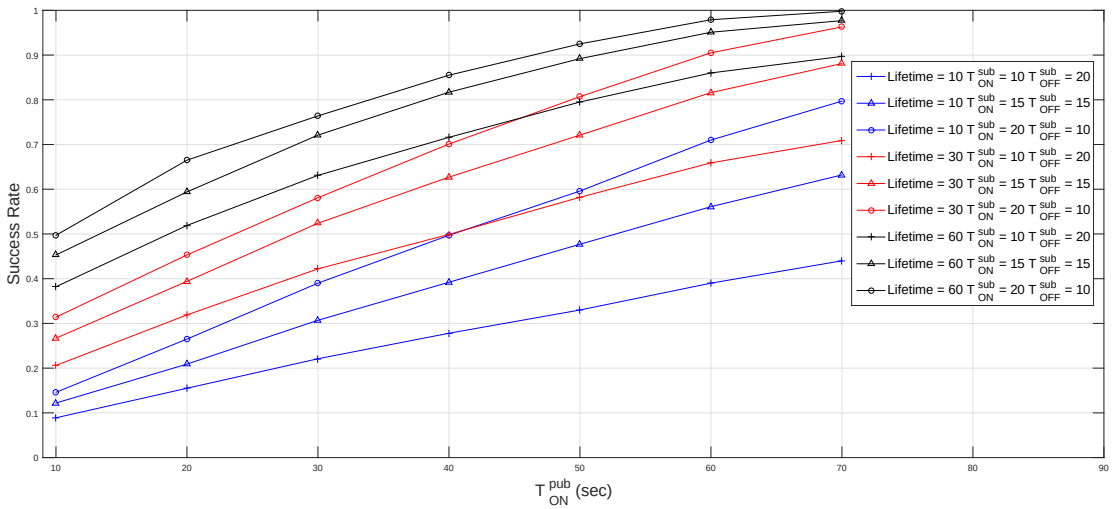


Figure 5.20: Success rates for the reliable pattern 4 with varying connection/disconnection and lifetime periods.

vehicular, rail or metro passengers. In [13], we concluded that connectivity patterns in the metro depend on both the network coverage and crowdedness of the metro. Moreover, we measured average ON/OFF periods in the scale of 0.5-1.5 min. Accordingly, we set the subscriber's total average connected + disconnected period to be $T_{\text{ON}}^{\text{sub}} + T_{\text{OFF}}^{\text{sub}} = 30$ sec. We apply this $T_{\text{ON}}^{\text{sub}} + T_{\text{OFF}}^{\text{sub}}$ period to the ON/OFF queue inside the broker transmitting messages to the subscriber. Experiments are performed by varying the $T_{\text{ON}}^{\text{sub}}$ and $T_{\text{OFF}}^{\text{sub}}$ periods inside the 30 sec interval. During ON periods, messages are transmitted to the subscriber with a service rate of $\mu_{\text{tr}} = 32$ messages/sec (again this represents the network delay on the broker/subscriber link). Finally, at the subscriber's side we apply a service rate of $\mu_{\text{pr}} = 64$ sec for the processing of incoming messages by the continuous queueing center.

Delivery Success Rates

In order to evaluate the effect of varying lifetime and connection/disconnection periods on delivery success rates, we perform simulations after applying the above parameters to the queueing network of Fig. 5.10. At the publisher's ON/OFF queueing center, the $T_{\text{ON}}^{\text{pub}}$ period varies from 10 to 70 sec, increased by 10 sec at each experiment. Thus, $T_{\text{OFF}}^{\text{pub}}$ equals to the remaining time from the 80 sec total. At the subscriber's side, connections ($T_{\text{ON}}^{\text{sub}}$) last 10, 15, 20 sec and disconnections ($T_{\text{OFF}}^{\text{sub}}$) equal to the remaining, 20, 15, 10 sec. The rates of successful interactions are shown in Fig. 5.20 for various values of lifetime, and $T_{\text{ON}}/T_{\text{OFF}}$ periods for both the publisher and the subscriber. Using MobileJINQS, we perform around 700000 interactions for each experiment. As expected, increasing T_{ON} (of the publisher or of the subscriber) periods for individual `lifetime` values improves the success rate. On the other hand, the success rate is severely bounded by lifetime periods, especially for lower values. Hence, increasing lifetime periods from 10 sec to 30 sec is necessary to have a success rate of more than 60% for a connectivity

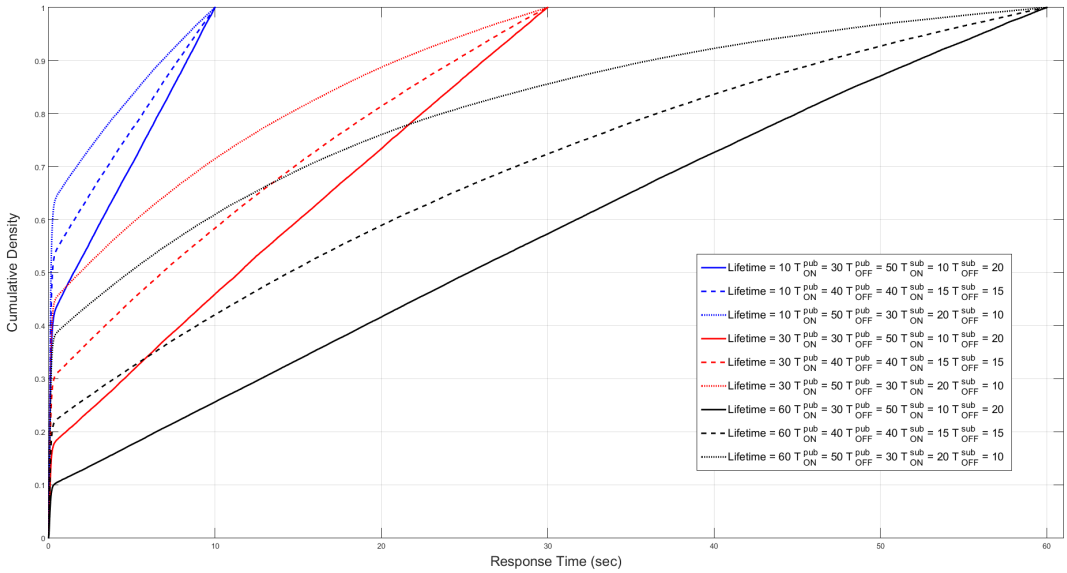


Figure 5.21: Cumulative distributions of response times for the reliable pattern 4 with varying connection/disconnection and lifetime periods.

of $T_{\text{ON}}^{\text{pub}} = 50$ sec (63% of the time) and $T_{\text{ON}}^{\text{sub}} = 20$ sec (67% of the time).

Response Time vs. Success Rate

In order to study the trade-off between end-to-end response times and delivery success rates, we present cumulative response time distributions in Fig. 5.21. In comparison to the previous set of experiments, we keep the same intervals for the subscriber's connections/disconnections ($T_{\text{ON}}^{\text{sub}}/T_{\text{OFF}}^{\text{sub}}$), while the publisher's connections ($T_{\text{ON}}^{\text{pub}}$) occur for 30, 40, 50 sec and disconnections ($T_{\text{OFF}}^{\text{pub}}$) equal to the remaining, 50, 40, 30 sec. Fig. 5.21, shows response times for successful interactions (i.e., we plot only interactions having response times lower than the lifetime period). From Fig. 5.21, lower lifetime periods produce markedly improved response time. For instance, with lifetime = 10 sec and equal $T_{\text{ON}}/T_{\text{OFF}}$ periods, 60% of the interactions complete within 1 sec. Comparing this to Fig. 5.20, with lifetime 10 sec, $T_{\text{ON}}^{\text{pub}} = T_{\text{OFF}}^{\text{pub}} = 40$ sec and $T_{\text{ON}}^{\text{sub}} = T_{\text{OFF}}^{\text{sub}} = 15$ sec, the probability of response time to be less than 10 sec is 1 while the success rate is 0.32. By increasing the lifetime to 30 sec, the probability of response time to be less than 10 sec is 0.58 and the success rate is 0.65. Generally, these tradeoffs confirm that higher lifetimes give better success rates but with higher response times. Through these experiments, we confirm (with respect to the similar evaluation performed in Chapter 4) that our analysis provides general guidelines for setting the lifetime and connection/disconnection periods to ensure successful interactions.

As already pointed out, this chapter models heterogeneous interactions by taking into account queueing and varying reliability effects of the end-to-end protocol infrastructure, are included

in the studied QoS semantics. In our analysis of Chapter 4, we assumed that the effect of the underlying protocol infrastructure was negligible. Fig. 5.21 and Fig. 4.11 of Chapter 4, present response times distributions with varying lifetime and connectivity periods. It is worth noting that Fig. 5.21 does not plot expired messages (i.e., messages with response times > lifetime), while Fig. 4.11 plots expired messages with response times = lifetime. By comparing these graphs, we notice that curves in Fig. 5.21 are smoother than the corresponding ones in Fig. 4.11. This result is expected since the model of Chapter 4 simulates an $M/G/\infty/\infty$ queueing model where the general distribution characterizing service times represents the disconnections of receivers and actual service times equal 0, and thus, when mobile subscribers reconnect, messages are delivered immediately. The simulation analysis of this chapter constitutes a more accurate tool for system designers.

5.5 Discussion

In this chapter, we model the interactions of mobile IoT middleware protocols using *Queueing Network Models* (QNMs). QNMs are employed to provide both analytical and simulation solutions for several performance metrics. To include the intermittent connectivity of mobile Things, we introduce the "ON/OFF queueing center". The ON/OFF queueing center is modeled as a separate queueing center and solved analytically. Such a queueing center can be then incorporated within a queueing network for the modeling of end-to-end IoT interactions.

By relying on the existing literature of QNMs and our ON/OFF queueing center, we create queueing networks that model the QoS of mobile IoT middleware protocols. In particular, we provide seven performance modeling patterns for middleware protocols which follow our CS, PS, DS and TS core communication styles. We include into these patterns several QoS semantics such as lifetime and timeout periods, intermittent connectivity, reliable/unreliable protocols, etc. Subsequently, we provide a method for modeling end-to-end interactions among heterogeneous Things interconnected via VSB by composing performance patterns.

We validate our analytical solution of the ON/OFF queueing center, using both probability distributions and real world traces. Our analytical model matches simulation results with small deviation, which demonstrates the efficacy of our work. We then explore the trade-off between response times and delivery success rates by applying various values to the lifetime of messages, intermittent connectivity intervals and arrival rates. To perform our simulation-based analyses of the above models, we have developed *MobileJIQS*, an open source simulator. Implementation details can be found in the appendix C.

The contributions of this chapter can provide system designers with precise design-time modeling and analysis methods and tools for heterogeneous mobile IoT systems. These can help designers in their design choices in order to ensure accurate runtime system behavior.

Chapter 6

Conclusions

Contents

The profusion of IoT middleware protocols introduces technology diversity which results in the deployment of highly heterogeneous Things, a considerable portion of which is mobile. The effect of heterogeneity requires the introduction of advanced interoperability solutions integrated with end-to-end performance modeling and evaluation techniques. In this thesis, we provided a comprehensive answer to this requirement. In this concluding chapter, we summarise our contributions and present future work.

6.1 Summary of contributions

This thesis tackles interoperability between heterogeneous Things employing different middleware protocols and QoS semantics. More specifically, we presented an overall platform to achieve interoperability through the automated synthesis of software artifacts that bridge the functional and QoS semantics of heterogeneous Things. Our contribution beyond the state-of-the-art primarily lies in enabling interactions of heterogeneous Things and at the same time evaluating the effectiveness of their interconnection. We recall our platform in Fig. 6.1. For the development of an IoT application, system designers can leverage our overall platform as follows:

1. Initially, designers utilize a tool (our Eclipse plugin) to define the functional semantics of existing heterogeneous Things to be integrated in the IoT application. Subsequently, the specified semantics are provided as input to our development framework, the *eVolution Service Bus* (VSB), which automatically synthesizes software artifacts enabling interoperability between the heterogeneous Things. These artifacts can be deployed in Cloud.

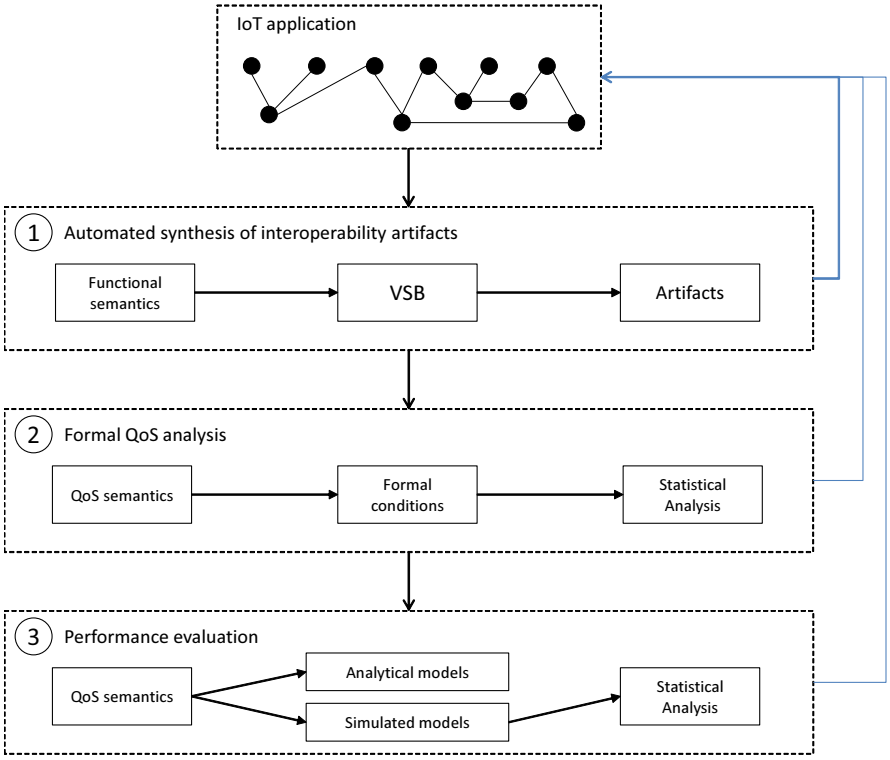


Figure 6.1: Platform for interoperability and interoperability effectiveness evaluation in IoT applications.

2. As a second step, designers can define non-functional (QoS) semantics related to: *i)* application's data availability/validity in time; and *ii)* the intermittent availability of data recipients. Subsequently, these semantics can analyzed by our platform, which provides designers with formal conditions for achieving successful interactions. Finally, designers can study the effect of varying QoS semantics, by using our statistical analysis method, and tune accordingly the IoT application.
3. In the final step, designers can perform a comprehensive performance evaluation of the IoT application. Specifically, apart from the QoS semantics of the previous step, they can introduce additional semantics related to the end-to-end protocol infrastructure. Reliable vs. unreliable infrastructures, as well as intermittent network connectivity can be introduced as model parameters. Our platforms provides analytical and simulation patterns that model the heterogeneous Things' end-to-end interconnections. Analytical models provide the estimation of average end-to-end response times and simulation models can be leveraged for further statistical analysis and tuning.

Our approach applies to the design and development phase of IoT applications. In our future work, we intend to extend our results to the runtime phase. The next section describes this perspective.

6.2 Perspectives for future work

In this thesis we have focused on design-time analysis, in order to ensure accurate runtime system behavior. However, Things can be mobile, low-powered and inexpensive which makes them vulnerable to system changes. Such changes can occur due to a variety of problems including faulty components, inaccurate sensing, intermittent connectivity and software bugs. Additionally, Things may participate in very dynamic, ad hoc IoT applications. For instance, after an earthquake, an IoT application may be dynamically deployed to handle its emergency situation. Such an application generates critical information and is expected to function correctly and reliably. Resilience is the ability of a system to persistently deliver trustworthy services despite system changes. Resilient operation of emergency IoT applications in the presence of failures and disruptions is a key requirement. Furthermore, as emergency situations may occur at various scales, such an operation must be equally ensured in large deployments.

Based on this context, we briefly introduce four possible directions that aim to extend this thesis for enabling interoperable, resilient and scalable interactions for emergency IoT applications.

Dynamic composition of Things in emergency scenarios. The aim is to enable the composition of Things available in the environment in order to face possible emergencies and ensure safety. Such composition requires the automated synthesis of interoperability artifacts, as well as their automated deployment and enactment.

QoS-aware adaptation of IoT middleware protocols. The aim is to dynamically manage the QoS and resource characteristics and needs of the heterogeneous Things and their synthesized interoperability artifacts, hence ensuring resilience at the middleware level. In particular, adaptation will require taking appropriate action, for instance, replacing a disconnected or low-performing system by another, substituting an artifact, or reserving on-demand additional resources (in Cloud) for a Thing.

Ensure IoT resilience for heterogeneous interactions. Interconnected heterogeneous mobile IoT devices may have different functional and QoS semantics, which need to be mapped. Such an end-to-end mapping may result in ineffective interactions. Our aim is to derive end-to-end performance models and analyses that take into account the underlying IoT networking capabilities *at runtime*.

Exploring large-scale IoT deployments. When introducing the above QoS models and interoperability artifacts, it is required to ensure their proper functionality in large-scale IoT deployments. Our final step aims to explore the proper deployment of interoperability artifacts in large-scale environments and at the same time ensure the applications' QoS requirements.

Bibliography

[1] T. S. Andrew and M. van Steen, "Distributed systems-principles and paradigms," 2007.

[2] H. Van der Veer and A. Wiles, "Achieving technical interoperability," *European Telecommunications Standards Institute*, 2008.

[3] S. Bandyopadhyay *et al.*, "Role of middleware for internet of things: A study," *International Journal of Computer Science and Engineering Survey*, 2011.

[4] V. Terziyan *et al.*, "Ubiroad: Semantic middleware for context-aware smart road environments," in *ICIW*, Barcelona, Spain, May 2010.

[5] P. Eugster *et al.*, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, 2003.

[6] G. Bouloukakis *et al.*, "Integration of Heterogeneous Services and Things into Choreographies," in *ACM ICSOC*, Banff, Alberta, Canada, October 2016.

[7] N. Georgantas *et al.*, "Service-oriented distributed applications in the future internet: The case for interaction paradigm interoperability," in *ESOCC*, Managa, Spain, September 2013.

[8] CHOReVOLUTION, "Chorevolution service bus, security and cloud - first outcomes," Automated Synthesis of Dynamic and Secured Choreographies, Tech. Rep., 2015.

[9] A. Kattepur *et al.*, "Analysis of timing constraints in heterogeneous middleware interactions," in *ACM ICSOC*, Goa, India, November 2015.

[10] G. Bouloukakis *et al.*, "Performance modeling of the middleware overlay infrastructure of mobile things," in *IEEE ICC*, Paris, France, May 2017.

[11] G. Bouloukakis *et al.*, "Timeliness evaluation of intermittent mobile connectivity over pub/sub systems," in *ACM/SPEC ICPE*, L Aquila, Italy, April 2017.

[12] G. Bouloukakis *et al.*, "Leveraging CDR datasets for Context-Rich Performance Modeling of Large-Scale Mobile Pub/Sub Systems," in *IEEE WiMob*, Abu Dhabi, UAE, October 2015.

[13] G. Bajaj *et al.*, "Toward Enabling Convenient Urban Transit through Mobile Crowdsensing," in *IEEE ITSC*, Las Palmas, Gran Canaria, September 2015.

[14] T. Teixeira *et al.*, "Service oriented middleware for the internet of things: A perspective," in *ServiceWave*. Springer-Verlag, Poznan, Poland, October 2011.

[15] A. Al-Fuqaha *et al.*, "Toward better horizontal integration among iot services," *IEEE Communications Magazine*, 2015.

[16] A. Molisch *et al.*, "Ieee 802.15. 4a channel model-final report," *IEEE P802*, 2004.

[17] Z. Specification, "Zigbee specification," *ZigBee Standard Organization*, 2008.

[18] A. Kim *et al.*, "When hart goes wireless: Understanding and implementing the wirelesshart standard," in *ETFA*, Hamburg, Germany, September 2008.

[19] T. Lennvall *et al.*, "A comparison of wirelesshart and zigbee for industrial applications," in *WFCS*, Dresden, Germany, May 2008.

[20] C. Gomez *et al.*, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, 2012.

[21] "Z-Wave Protocol Overview, v. 4, May 2007," https://wiki.ase.tut.fi/courseWiki/images/9/94/SDS10243_2_Z_Wave_Protocol_Overview.pdf.

[22] M. Park, "Ieee 802.11 ah: sub-1-ghz license-exempt operation for the internet of things," *IEEE Communications Magazine*, 2015.

[23] L. Alliance, "Lorawan specification," *LoRa Alliance*, 2015.

[24] "I. Poole, Weightless wireless - M2M white space communications - tutorial, 2014," http://www.radio-electronics.com/info/wireless/weightless-m2m-white-space-wireless-communications/basics-overview.php.

[25] A. Al-Fuqaha *et al.*, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, 2015.

[26] X. Zhang and K. G. Shin, "Gap sense: Lightweight coordination of heterogeneous wireless devices," in *IEEE INFOCOM*, Turin, Italy, April 2013.

[27] I. Tinnirello *et al.*, "Cross-technology wifi/zigbee communications: Dealing with channel insertions and deletions," *IEEE Communications Letters*, 2016.

[28] G. Aloi *et al.*, "A mobile multi-technology gateway to enable iot interoperability," in *IEEE IoTDI*, Berlin, Germany, April 2016.

[29] Q. Zhu *et al.*, "Iot gateway: Bridgingwireless sensor networks into internet of things," in *IEEE/IFIP EUC*, Hong Kong, China, September 2010.

[30] M. Tucic *et al.*, "Device abstraction and virtualization: Concept of device in device," in *IEEE ICCE*, Berlin, Germany, September 2015.

[31] Z. Shelby and C. Bormann, *6LoWPAN: The wireless embedded Internet*. John Wiley & Sons, 2011.

[32] F. Aıssaoui, G. Garzone, and N. Seydoux, "Providing interoperability for autonomic control of connected devices."

[33] "Semantic sensor network ontology," https://www.w3.org/2005/Incubator/ssn/ssnx/ssn.

[34] "Iot-o," https://www.irit.fr/recherches/MELODI/ontologies/IoT-O.html.

[35] J. Kiljander *et al.*, "Semantic interoperability architecture for pervasive computing and internet of things," *IEEE access*, 2014.

[36] G. Xiao *et al.*, "User interoperability with heterogeneous iot devices through transformation," *IEEE Transactions on Industrial Informatics*, 2014.

[37] D. Androcec and N. Vrcek, "Thing as a service interoperability: Review and framework proposal," in *IEEE FiCloud*, Vienna, Austria, August 2016.

[38] F. Michahelles and S. Mayer, "Toward a web of systems," *XRDS*.

[39] "Inter-iot - interoperability internet of things," http://www.inter-iot-project.eu.

[40] R. Agarwal *et al.*, "Unified iot ontology to enable interoperability and federation of testbeds," in *IEEE WF-IoT*, Reston, VA, USA, December 2016.

[41] "Fiesta-iot - federated interoperable semantic iot testbeds and applications," http://fiesta-iot.eu.

[42] "M3 lite," http://lov.okfn.org/dataset/lov/vocabs/m3lite.

[43] M. Bermudez-Edo *et al.*, "Iot-lite: A lightweight semantic model for the internet of things," in *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld*, 2016.

[44] J. R. Hobbs and F. Pan, "Time ontology in owl," *W3C working draft*, 2006.

[45] "Dul," http://lov.okfn.org/dataset/lov/vocabs/dul.

[46] E. Zeeb *et al.*, "Service-oriented architectures for embedded systems using devices profile for web services," in *AINA Workshops*, Niagara Falls, Canada, May 2007.

[47] M. Damm *et al.*, "Opc unified architecture," *OPC Unified Architecture. Springer-Verlag Berlin Heidelberg, 2009.*

[48] Z. Shelby *et al.*, "The constrained application protocol (coap)," Tech. Rep., 2014.

[49] T. F. Roy, "The representational state transfer (rest)," *Department of Information and Computer Science, UCI*, 2000.

[50] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," 2011.

[51] "Sun Microsystems. JMS Specifications and Reference Implementation," http://www.oracle.com/technetwork/java/jms/index.html.

[52] "DDS. Data Distribution Service," http://portals.omg.org/dds/.

[53] A. Banks and R. Gupta, "Mqtt version 3.1. 1," *OASIS standard*, 2014.

[54] O. Standard, "Oasis advanced message queuing protocol (amqp) version 1.0., 2012."

[55] "SemiSpace. Light weight Open Source interpretation of Tuple Space / Object Space," http://www.semispace.org/.

[56] I. Fette, "The websocket protocol," 2011.

[57] R. Kyusakov *et al.*, "Integration of wireless sensor and actuator nodes with it infrastructure using service-oriented architecture," *IEEE Transactions on industrial informatics*, 2013.

[58] "JavaSpaces. Beyond Conventional Distributed Programming Paradigms," http://www.oracle.com/technetwork/articles/java/javaspaces-140665.html.

[59] V. Karagiannis *et al.*, "A survey on application layer protocols for the internet of things," *Transaction on IoT and Cloud Computing*, 2015.

[60] K. Fysarakis *et al.*, "Which iot protocol? comparing standardized approaches over a common m2m application," WashingtonDC, USA, July 2016.

[61] A. Talaminos-Barroso *et al.*, "A machine-to-machine protocol benchmark for ehealth applications–use case: Respiratory rehabilitation," *Computer methods and programs in biomedicine*, 2016.

[62] L. Atzori *et al.*, "The internet of things: A survey," *Computer Networks*, October, 2010.

[63] J. Gubbi *et al.*, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, September, 2013.

[64] M. A. Razzaque *et al.*, "Middleware for internet of things: A survey," *IEEE Internet of Things Journal*, 2016.

[65] D. Guinard *et al.*, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE transactions on Services Computing*, 2010.

[66] M. Eisenhauer *et al.*, "Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems," in *The Internet of Things*. Springer, 2010.

[67] C. Fok *et al.*, "Servilla: a flexible service provisioning middleware for heterogeneous sensor networks," *Science of Computer Programming*, 2012.

[68] I. Corredor *et al.*, "Knowledge-aware and service-oriented middleware for deploying pervasive services," *Journal of Network and Computer Applications*, 2012.

[69] K. Aberer *et al.*, "Infrastructure for data processing in large-scale interconnected sensor networks," in *IEEE MDM*, Mannheim, Germany, May 2007.

[70] H. Meling *et al.*, "Sensewrap: A service oriented middleware with sensor virtualization and self-configuration," in *IEEE ISSNIP*, Melbourne, Australia, December 2009.

[71] C. Perera *et al.*, "Mosden: An internet of things middleware for resource constrained mobile devices," in *IEEE HICSS*, Waikoloa, HI, USA, January 2014.

[72] D. Athanasopoulos *et al.*, "An Architectural Style for the Development of Choreographies in the Future Internet," *Global Journal of Advanced Software Engineering*.

[73] S. Cherrier and Y. M. Ghamri-Doudane, "The "object-as-a-service" paradigm," in *IEEE GIIS*, Montreal, Canada, September 2014.

[74] S. Cherrier *et al.*, "D-lite: Building internet of things choreographies," *arXiv*, 2016.

[75] S. Cherrier *et al.*, "Fault-recovery and coherence in internet of things choreographies," in *IEEE WF-IoT*, Seoul, Korea (South), March 2014.

[76] D. Chappell, *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.

[77] L. Alboaie *et al.*, "Swarm communication-a messaging pattern proposal for dynamic scalability in cloud," in *IEEE HPCC EUC*, Zhangjiajie, China, November 2013.

[78] L. Alboaie *et al.*, "Extending swarm communication to unify choreography and long-lived processes," 2014.

[79] CHOReOS, "Final CHOReOS architectural style," Large Scale Choreographies for the Future Internet, Tech. Rep., 2013.

[80] CHOReOS, "Integrated choreos middleware - enabling large-scale, qos-aware adaptive choreographies," Large Scale Choreographies for the Future Internet, Tech. Rep., 2013.

[81] B. Negash *et al.*, "Lisa: lightweight internet of things service bus architecture," *Procedia Computer Science*, 2015.

[82] B. Negash *et al.*, "Lisa 2.0: lightweight internet of things service bus architecture using node centric networking," *Journal of Ambient Intelligence and Humanized Computing*, 2016.

[83] B. Negash *et al.*, "Enabling layered interoperability for internet of things through lisa."

[84] B. Cheng *et al.*, "Situation-aware iot service coordination using the event-driven soa paradigm," *IEEE Transactions on Network and Service Management*, 2016.

[85] Y. Tao *et al.*, "Service-based interactive proxy for sensor networks in smart home: An implementation of home service bus," in *IEEE ICDH*, Guangzhou , China, November 2014.

[86] A. Ismail and W. Kastner, "A middleware architecture for vertical integration," in *IEEE CPPS Workshop*. IEEE, Vienna, Austria, April 2016.

[87] "Ponte - m2m bridge framework," http://www.eclipse.org/proposals/technology.ponte/.

[88] "Intel iot gateway," https://software.intel.com/en-us/articles/what-is-the-gateway-and-why-should-i-care.

[89] M. Blackstock and R. Lea, "Iot interoperability: A hub-based approach," in *IEE IOT*, MIT Media Lab, Cambridge, MA, 2014.

[90] W. Macêdo *et al.*, "Gothings-an application-layer gateway architecture for the internet of things." in *WEBIST*, Lisbon, Portugal, May 2015.

[91] P. Desai *et al.*, "Semantic gateway as a service architecture for iot interoperability," in *IEEE MS*, New York, USA, June 2015.

[92] S. Datta *et al.*, "An iot gateway centric architecture to provide novel m2m services," in *WF-IoT*, Seoul, Korea (South), March 2014.

[93] M. Collina *et al.*, "Introducing the qest broker: Scaling the iot by bridging mqtt and rest," in *IEEE PIMRC*, Sydney, Australia, September 2012.

[94] A. Castellani *et al.*, "Http-coap cross protocol proxy: an implementation viewpoint," in *IEEE MASS*, Las Vegas, Nevada, USA, October 2012.

[95] S. Han *et al.*, "Extending the devices profile for web services standard using a rest proxy," *IEEE Internet Computing*, 2015.

[96] "Eclipse iot - open source for iot," https://iot.eclipse.org/.

[97] "Kura - osgi-based application framework for m2m service gateways," http://www.eclipse.org/proposals/technology.kura.

[98] "Eclipse neoscada," http://projects.eclipse.org/projects/technology.eclipsescada.

[99] "Eclipse smarthome," https://eclipse.org/proposals/technology.smarthome/.

[100] "Krikkit," http://www.eclipse.org/proposals/technology.krikkit/.

[101] S. K. Datta and C. Bonnet, "Smart m2m gateway based architecture for m2m device and endpoint management," in *iThings, GreenCom, CPSCom*, Taipei, Taiwan, September 2014.

[102] S. K. Datta and C. Bonnet, "A lightweight framework for efficient m2m device management in onem2m architecture," in *IEEE RIoT*, Singapore, April 2015.

[103] R. Morabito *et al.*, "Enabling a lightweight edge gateway-as-a-service for the internet of things," in *IEEE NOF*, Búzios, Rio de Janeiro, Brazil, November 2016.

[104] G. Suciu *et al.*, "Generic platform for iot and cloud computing interoperability study," in *IEEE ISSCS*, Iasi, Romania, July 2013.

[105] S. Soursos *et al.*, "Towards the cross-domain interoperability of iot platforms," in *IEEE EuCNC*, Athens, Greece, June 2016.

[106] A. Bröring *et al.*, "Enabling iot ecosystems through platform interoperability," *IEEE Software, forthcoming*, 2017.

[107] "Mda - the architecture of choice for a changing world," http://www.omg.org/mda/.

[108] "Vorto," https://projects.eclipse.org/proposals/vorto.

[109] "Eclipse modeling framework (emf)," https://eclipse.org/modeling/emf/.

[110] "Franca," https://www.eclipse.org/proposals/modeling.franca/.

[111] F. Ciccozzi *et al.*, "Model-driven engineering for mission-critical iot systems," *IEEE Software*, 2017.

[112] P. Grace *et al.*, "Model-driven interoperability: engineering heterogeneous iot systems," *Annals of Telecommunications*, 2016.

[113] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, 2011.

[114] K. Sood *et al.*, "Software-defined wireless networking opportunities and challenges for internet-of-things: A review," *IEEE Internet of Things Journal*, 2016.

[115] Z. Qin *et al.*, "A software defined networking architecture for the internet-of-things," in *IEEE NOMS*, Krakow, Poland, May 2014.

[116] O. Salman *et al.*, "Edge computing enabling the internet of things," in *IEEE WF-IoT*, Milan, Italy, December 2015.

[117] C. Lee *et al.*, "Interoperability enhancement for internet of things protocols based on software-defined network," in *IEEE GCCE*, Kyoto, Japan, October 2016.

[118] T. Zachariah *et al.*, "The internet of things has a gateway problem," in *ACM HotMobile Workshop*, Santa Fe, NM, USA, February 2015.

[119] "Pivotal, "RabbitMQ"," https://www.rabbitmq.com/.

[120] "Apache Kafka," http://kafka.apache.org/.

[121] "GigaSpaces. Empowering next Generation Web Scale Applications," http://www.gigaspaces.com/.

[122] "Terrastore. Ubiquitous, Distributed and Elastic clustering technology," https://code.google.com/archive/p/terrastore/.

[123] "Lime. Linda in a Mobile Enviroment," http://lime.sourceforge.net/Lime/index.html.

[124] D. Thangavel *et al.*, "Performance evaluation of mqtt and coap via a common middleware," in *IEEE ISSNIP*, Singapore, April 2014.

[125] S. Mukherjee *et al.*, "An applications interoperability model for heterogeneous internet of things environments," *Computers & Electrical Engineering*, 2017.

[126] S. Bandyopadhyay and A. Bhattacharyya, "Lightweight internet protocols for web enablement of sensors using constrained gateway devices," in *IEEE ICNC*, San Diego, USA, January 2013.

[127] Y. Sun *et al.*, "A low-delay, lightweight publish/subscribe architecture for delay-sensitive iot services," in *IEEE ICWS*, Santa Clara, CA, USA, June 2013.

[128] S. Lee *et al.*, "Correlation analysis of mqtt loss and delay according to qos level," in *IEEE ICOIN*, Bangkok, Thailand, January 2013.

[129] Z. Babovic *et al.*, "Web performance evaluation for internet of things applications," *IEEE Access*, 2016.

[130] E. Davis *et al.*, "Improving packet delivery performance of publish/subscribe protocols in wireless sensor networks," *Sensors*, 2013.

[131] F. Mehmeti and T. Spyropoulos, "Performance analysis of "on-the-spot" mobile data offloading," in *IEEE GLOBECOM*, Atlanta, GA USA, December 2013.

[132] F. Mehmeti and T. Spyropoulos, "Performance analysis of mobile data offloading in heterogeneous networks."

[133] F. Mehmeti and T. Spyropoulos, "Is it worth to be patient? analysis and optimization of delayed mobile data offloading," in *IEEE INFOCOM*, Toronto, Canada, April - May 2014.

[134] K. Lee *et al.*, "Mobile data offloading: how much can wifi deliver?" in *Proceedings of the 6th International COnference*. ACM, 2010.

[135] E. Hyytiä *et al.*, "Optimizing offloading strategies in mobile cloud computing," 2013.

[136] H. Wu and K. Wolter, "Tradeoff analysis for mobile cloud offloading based on an additive energy-performance metric," in *VALUETOOLS*. ICST, Bratislava, Slovakia, December 2014.

[137] T. Phung-Duc *et al.*, "A simple algorithm for the rate matrices of level-dependent qbd processes," in *ACM QTNA*, Beijing, China, July 2010.

[138] E. Altman and U. Yechiali, "Analysis of customers? impatience in queues with server vacations," *Queueing Systems*, 2006.

[139] G. Latouche and V. Ramaswami, *Introduction to matrix analytic methods in stochastic modeling*. Siam, 1999.

[140] M. Vernon *et al.*, *A comparison of performance Petri nets and queueing network models*. University of Wisconsin-Madison, Computer Sciences Department, 1986.

[141] L. Aldred *et al.*, "On the notion of coupling in communication middleware," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, Agia Napa, Cyprus, October 2005.

[142] A. Kattepur and M. Nambiar, "Performance modeling of multi-tiered web applications with varying service demands," in *IPDPS Workshops*, Hyderabad, India, May 2015.

[143] A. Basu *et al.*, "Statistical abstraction and model-checking of large heterogeneous systems," in *Formal Techniques for Distributed Systems*. Springer, 2010.

[144] L. Waszniowski, J. Krakora, and Z. Hanzalek, "Case study on distributed and fault tolerant system modeling based on timed automata," *The Journal of Systems and Software*, 2009.

[145] Y. Zhou *et al.*, "Model based verification of dynamically evolvable service oriented systems," *Science China Information Sciences*, 2016.

[146] M. Kim *et al.*, "Combining formal verification with observed system execution behavior to tune system parameters," in *Formats*. Springer, Québec, Canada, August 2007.

[147] B. Aziz, "A formal model and analysis of an iot protocol," *Ad Hoc Networks*, 2016.

[148] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, 1994.

[149] G. Behrmann *et al.*, "A tutorial on uppaal 4.0 (2006)."

[150] M. Kwiatkowska *et al.*, "PRISM: Probabilistic symbolic model checker," in *MMMECCS*, Aachen, Germany, September 2001.

[151] A. Nouri *et al.*, "Performance evaluation of complex systems using the sbip framework," in *VECoS*, Montreal, Québec, Canada, August 2016.

[152] F. He *et al.*, "Formal analysis of publish-subscribe systems by probabilistic timed automata," in *FORTE*. Springer, Tallinn, Estonia, June 2007.

[153] L. Baresi *et al.*, "On accurate automatic verification of publish-subscribe architectures," in *ICSE*. IEEE Computer Society, Minneapolis, May 2007.

[154] T. Pongthawornkamol *et al.*, "The analysis of publish/subscribe systems over mobile wireless ad hoc networks," in *IEEE MobiQuitous*, Philadelphia, PA, USA, August 2007.

[155] T. Pongthawornkamol *et al.*, "Probabilistic qos modeling for reliability/timeliness prediction in distributed content-based publish/subscribe systems over best-effort networks," in *ACM ICAC*, Washington, DC, USA, June 2010.

[156] T. Pongthawornkamol, "Reliability and timeliness analysis of content-based publish/subscribe systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2011.

[157] D. Kassa *et al.*, "Analytical models of short-message reliability in mobile wireless networks," in *ACM MSWiM*, Miami, FL, USA, October 2011.

[158] A. A. Gaddah, "A pro-active mobility management scheme for publish/subscribe middleware systems," Ph.D. dissertation, Citeseer, 2008.

[159] A. Gaddah and T. Kunz, "Extending mobility to publish/subscribe systems using a pro-active caching approach," *Mobile Information Systems*, 2010.

[160] S. Kounev *et al.*, "A methodology for performance modeling of distributed event-based systems," in *IEEE ISORC*. IEEE, Orlando, FL, USA, May 2008.

[161] G. Mühl *et al.*, "Stochastic analysis of hierarchical publish/subscribe systems," in *Euro-Par*. Springer, Delft, The Netherlands, August 2009.

[162] T. Martinec *et al.*, "Constructing performance model of jms middleware platform," in *ACM/SPEC ICPE*, Dublin, Ireland, March 2014.

[163] K. Sachs *et al.*, "Performance modeling and analysis of message-oriented event-driven systems," *Software & Systems Modeling*, 2013.

[164] G. Singh and A. Singh, "Measuring tradeoffs between performance and qos in event based systems," 2015.

[165] V. Setty *et al.*, "The hidden pub/sub of Spotify:(industry article)," in *ACM DEBS*, Arlington, TX, USA, June 2013.

[166] V. Setty *et al.*, "Maximizing the number of satisfied subscribers in Pub/Sub systems under capacity constraints," in *IEEE INFOCOM*, Toronto, Canada, May 2014.

[167] V. J. Setty, "Publish/subscribe for large-scale social interaction: Design, analysis and resource provisioning," 2015.

[168] D. Schrank *et al.*, "Tti's 2012 urban mobility report," *Texas A&M Transportation Institute. The Texas A&M University System*, 2012.

[169] "ITS. Intelligent Transportation Systems," http://www.flir.co.uk/traffic/content/?id=66601.

[170] "XD. Traffic," http://inrix.com/xd-traffic.

[171] J. Yoon *et al.*, "Surface street traffic estimation," in *ACM Mobisys*, PR, USA, June 2007.

[172] P. Mohan *et al.*, "Nericell: rich monitoring of road and traffic conditions using mobile smartphones," in *ACM SenSys*, Raleigh, NC, USA, November 2008.

[173] B. Billet and V. Issarny, "dioptase: data streaming middleware for the internet of things," *ERCIM News*, 2015.

[174] K. Ueno and M. Tatsubori, "Early capacity testing of an enterprise service bus," in *IEEE ICWS*, Chicago, USA, September 2006.

[175] S. Desmet *et al.*, "Throughput evaluation of different enterprise service bus approaches," 2007.

[176] E. Clarke *et al.*, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1986.

[177] N. De Caro *et al.*, "Comparison of two lightweight protocols for smartphone-based sensing," in *IEEE SCVT*, Namur, Belgium, November 2013.

[178] L. Durkop *et al.*, "Performance evaluation of m2m protocols over cellular networks in a lab environment," in *IEEE ICIN*, Paris, France, February 2015.

[179] E. Lazowska *et al.*, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[180] F. Baskett *et al.*, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, 1975.

[181] I. Adan and J. Resing, *Queueing Theory: Ivo Adan and Jacques Resing*. Eindhoven University of Technology. Dep. of Mathematics and Computing Science, 2005.

[182] W. Chang, "Preemptive priority queues," *Operations research*, 1965.

[183] G. R. Wright and W. R. Stevens, *TcP/IP Illustrated*. Addison-Wesley Professional, 1995.

[184] T. Field, "Jinqs: An extensible library for simulating multiclass queueing networks, v1. 0 user guide," August 2006.

[185] R. Baldoni and A. Virgillito, "Distributed event routing in publish/subscribe communication systems: a survey," *DIS, Universita di Roma La Sapienza, Tech. Rep*, 2005.

List of figures

List of tables

Appendix A

List of publications

In conference proceedings

- **G. Bouloukakis**, I. Moscholios, N. Georgantas, V. Issarny, *Performance Modeling of the Middleware Overlay Infrastructure of Mobile Things* in IEEE International Conference on Communications (ICC), Paris, France.

  `https://hal.inria.fr/hal-01470328`.

- **G. Bouloukakis**, N. Georgantas, A. Kattepur, V. Issarny, *Timeliness Evaluation of Intermittent Mobile Connectivity over Pub/Sub Systems* in 8th ACM/SPEC International Conference on Performance Engineering (ICPE), L'Aquila, Italy.

  `https://hal.inria.fr/hal-01415893`.

- V. Issarny, **G. Bouloukakis**, N. Georgantas, B. Billet, *Revisiting Service-oriented Architecture for the IoT: A Middleware Perspective* in International Conference on Service Oriented Computing (ICSOC), Banff, Alberta, Canada.

  `https://hal.inria.fr/hal-01358399`.

- G. Bajaj, R. Agarwal, **G. Bouloukakis**, P. Singh, N. Georgantas, V. Issarny, *Towards Building Real-Time, Convenient Route Recommendation System for Public Transit* in IEEE Second International Smart Cities Conference (ISC2), Trento, Italy.

  `https://hal.inria.fr/hal-01351068`.

- **G. Bouloukakis**, N. Georgantas, R. Agarwal, A. Pathak, V. Issarny, *Leveraging CDR datasets for Context-Rich Performance Modeling of Large-Scale Mobile Pub/Sub Systems* in 11th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Abu Dhabi, UAE, 2015.

  `https://hal.inria.fr/hal-01204871`.

- G. Bajaj, **G. Bouloukakis**, A. Pathak, P. Singh, N. Georgantas, V. Issarny, *Toward Enabling Convenient Urban Transit through Mobile Crowdsensing* in 18th IEEE International Conference on Intelligent Transportation Systems (ITSC), Las Palmas, Gran Canaria, 2015.

  `https://hal.inria.fr/hal-01204827`.

- A. Kattepur, N. Georgantas, **G. Bouloukakis**, V. Issarny, *Analysis of Timing Constraints in Heterogeneous Middleware Interactions* in International Conference on Service Oriented Computing (ICSOC), Goa, India, 2015.

  `https://hal.inria.fr/hal-01204786`.

- N. Georgantas, **G. Bouloukakis**, S. Beauche, V. Issarny, *Service-oriented Distributed Applications in the Future Internet: The case for Interaction Paradigm Interoperability* in European Conference on Service-Oriented and Cloud Computing (ESOCC), Malaga, Spain, 2013.

  `https://hal.inria.fr/hal-00841332`.

Demos, Posters

- **G. Bouloukakis**, N. Georgantas, S. Dutta, V. Issarny, *Integration of Heterogeneous Services and Things into Choreographies* Demo paper in International Conference on Service Oriented Computing (ICSOC), Banff, Alberta, Canada.

  `https://hal.inria.fr/hal-01358043`.

- **G. Bouloukakis**, N. Georgantas, R. Agarwal, A. Pathak, V. Issarny, *Towards Enabling Mobile Social Crowd-Sensing for Unstructured Transport Information Management* Poster Paper in D4D Challenge 2015, NetMob, Boston, USA, 8-10 April, 2015.

  `https://hal.inria.fr/hal-01206622`.

Research and technical reports

- F. Martelli, N. Georgantas, **G. Bouloukakis**, P. Ntumba, F. Motte, A. Carenini, *CHOReVOLUTION Service Bus, Security and Cloud – Intermediate outcomes* research report, April 2017.

- B. Billet, **G. Bouloukakis**, N. Georgantas, S. Hachem, V. Issarny, M. Autili, D. D. Ruscio, P. Inverardi, T. Massimo, A. D. Salle, D. Athanasopoulos, P. Vassiliadis, A. Zarras, *Final CHOReOS Architectural Style and its Relation with the CHOReOS Development Process and IDRE* research report, Dec. 2013.

  `https://hal.inria.fr/hal-00912869`.

Appendix B

VSB Framework

This appendix relates to the contributions presented in Chapter 3 where we presented VSB, a framework that seamlessly interconnects Things that employ heterogeneous interaction protocols at the middleware level (e.g., DPWS, CoAP, MQTT, etc). Particularly, to include a heterogeneous Thing inside an IoT application an application developer must describe it by creating its model (i.e., the description of a concrete Thing) using our GIDL metamodel. Then, the resulting GIDL model is utilized as input in our VSB synthesizer for generating an interoperability artifact (Binding Component). Such an artifact performs the bridging functionality between the Thing's middleware protocol and the application's common IoT protocol.

In this appendix, we describe in detail the attributes of the GIDL metamodel (presented in Fig. 3.12 in the Chapter 3). Then, we introduce a user guide for defining GIDL models of various Things, as well their utilization as input at the *VSB Manager* for synthesizing Binding Components.

B.1 GDIL metamodel Attributes

GIDLModel	
The GIDLModel metaclass is a root container and is used to specify the Thing's hostAddress, the middleware protocol that employs, and the number of the available Interfaces.	
Attribute Name	**Description**
hostAddress: EString [0..1]	This attribute represents the Thing's host IP address.
protocol: ProtocolTypes [1..1]	This attribute is used to define the Thing's employed middleware `protocol`. The value is derived by the set of the `ProtocolTypes` (e.g., REST, SOAP, etc).
hasInterfaces: InterfaceDescription [1..*]	It is a reference to a set of `InterfaceDescriptions` where each `InterfaceDescription` is another metaclass.

InterfaceDescription	
The `InterfaceDescription` metaclass is used to specify the `RoleType` of an `Interface` and the available `Operations`.	
Attribute Name	**Description**
role: RoleTypes [1..1]	This attribute specifies the role of the `Interface` which can be either `provider` or `consumer`.
hasOperations: Operation [1..*]	It is a reference to a set of `Operations` where each `Operation` is another metaclass.

Operation	
The `Operation` metaclass is used to specify its `name`, the `OperationType`, the `QosType`, its `Scope` and the `InputData`/`OutputData`.	
Attribute Name	**Description**
name: EString [0..1]	This attribute represents the Operation's name.
type: OperationTypes [1..1]	This attribute is used to define the `OperationType`. An operation type is related the specific interaction: `one-way`, `two-way sync`, `two-way async` and `stream`.
qos: QosTypes [1..1]	This attribute is used to define the QoS type. Types include `unreliable` and `reliable` interactions
outputData: Data [0..*]	It is a reference to a set of `Data`s that the operation provides. Each `Data` is another metaclass.
inputData: Data [0..*]	It is a reference to a set of `Data`s that the operation accepts. Each `Data` is another metaclass.
hasScope: Scope [1..1]	It is a reference to a `Scope` that is another metaclass.

Scope	
The `Scope` metaclass is used to specify several characteristics (`name`, `verb`, `uri`) of the corresponding operation. These charasteristics are related to the specific middleware protocol	
Attribute Name	**Description**
name: EString [0..1]	This attribute represents the Scope's name.
verb: EString [0..1]	This attribute represents an additional characteristic of a middleware protocol.
uri: EString [0..1]	This attribute represents an `URI`, if needed.

Data	
The `Scope` metaclass is used to specify the `DataTypes` and the specific `context`	
Attribute Name	**Description**
name: EString [0..1]	This attribute represents the Data's name.
context: ContextTypes [1..1]	This attribute represents the `context` of the data. Context types include `path`, `body`, `header`, etc. Thus it specifies the location of the exchange data.
hasDataType: DataType [1..*]	It is a reference to a set of `DataType`s where each `DataType` is an abstract metaclass.

DataType	
The `DataType` is an abstract metaclass that can be specialized using `ComplexType` or `SimpleType`.	
Attribute Name	**Description**
name: EString [0..1]	This attribute represents the DataType's name.
occurrences: OccurrencesTypes [1..1]	This attribute represents the `occurrences` of the data type. Occurrences Types include `one` or `unbounded`.

ComplexType	
The `ComplexType` metaclass specifies the different data types	
Attribute Name	**Description**
hasDataType: DataType [1..*]	It is a reference to a set of `DataTypes` where each `DataType` can be Simple or Complex type.

SimpleType	
The `SimpleType` metaclass specifies a simple data type	
Attribute Name	**Description**
type: SympleTypes [1..1]	This attribute represents the simple `type` of the data. Types include `integer`, `string`, etc.

B.2 Defining GDIL Models

To integrate a heterogeneous Thing inside an IoT application, an application developer must specify the related GIDL model. To specify such a model, we have developed an Eclipse plugin which can be installed in your favorite Eclipse package via the following site:

`http://nexus.disim.univaq.it/content/sites/chorevolution-modeling-notations`

In Chapter 3, we presented the TIM system, where participants interact with each other via the CoAP middleware protocol. However, our scenario includes Things that employ different protocols – i.e., `traffic-lights`, `fixed-sensors`, `vehicle-devices`, `smartphones` and the `estimation-service`. Hence, the application developer is able to integrate them in the scenario by creating a new GIDL project for each one of the heterogeneous Things as follows:

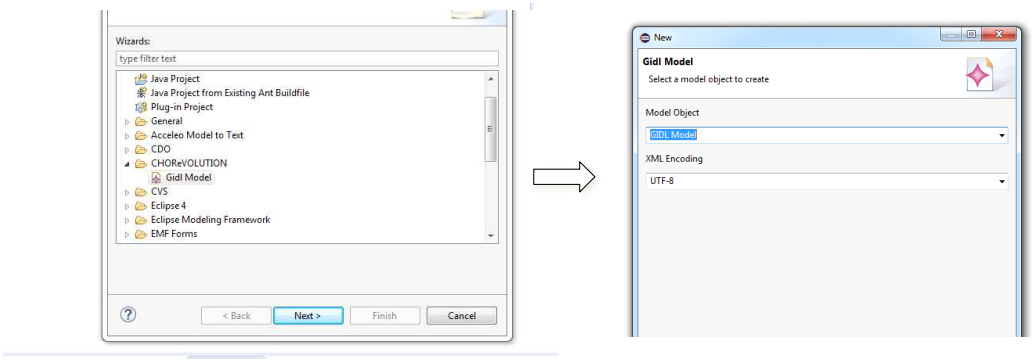


Figure B.1: Creating a new metamodel through our Eclipse plugin.

Subsequently, the Thing's metadata (i.e., operations, interaction types, input/output data, etc) must be defined. For instance, to include the Thing `traffic-light` (see Fig. 3.16 in Chapter 3), inside the TIM system, an application developer specifies its GIDL model as follows:

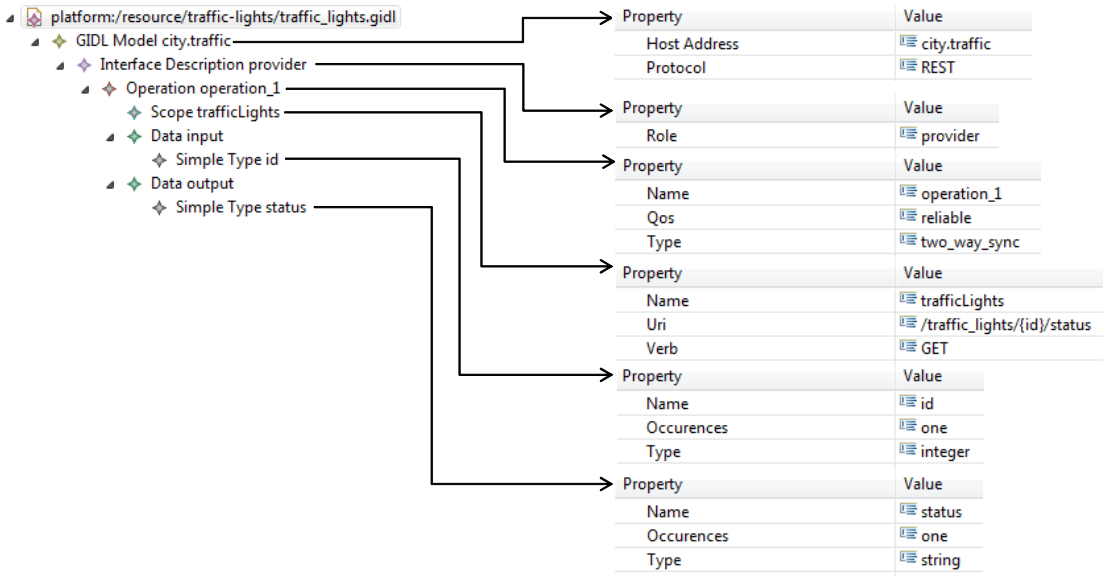


Figure B.2: GIDL model for the `traffic-light` Thing.

Which corresponds to the following XML representation:

```
<?xml version="1.0" encoding="UTF-8"?> <gidl:GIDLModel xmi:version="2.0"
     xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:gidl="http://eu.chorevolution/modelingnotations/gidl"
hostAddress="city.traffic" protocol="REST">
  <hasInterfaces role="provider">
    <hasOperations name="operation_1" type="two_way_sync" qos="reliable">
      <hasScope name="trafficLights" verb="GET" uri="/traffic_lights/{id}/status"/>
      <inputData name="input" context="path">
        <hasDataType xsi:type="gidl:SimpleType" name="id" occurences="one" type="integer"/>
      </inputData>
      <outputData name="output" context="body">
        <hasDataType xsi:type="gidl:SimpleType" name="status" occurences="one" type="string"/>
      </outputData>
    </hasOperations>
  </hasInterfaces>
</gidl:GIDLModel>
```

Listing B.1: XML representation of the `traffic-light` GIDL metamodel

The above model, represents a *server* Thing which implements an operation accepting requests and providing back the status of the corresponding traffic light. In the following, we provide the GIDL models for each one of the heterogeneous Things of the scenario, which include *server*, *client*, *sender*, *receiver*, *consumer* and *producer* roles.

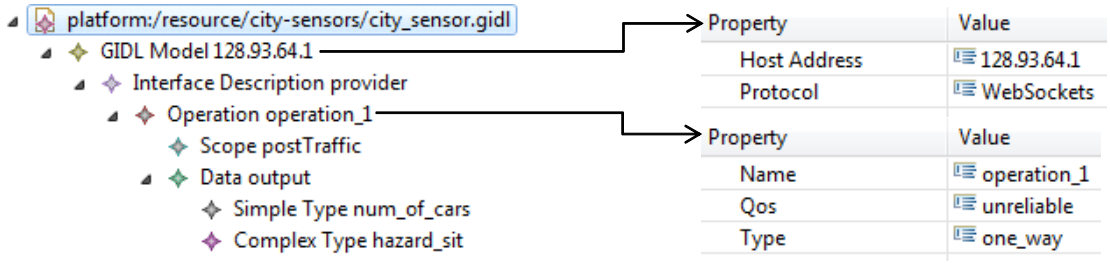


Figure B.3: GIDL model for `fixed-sensors`.

```
<?xml version="1.0" encoding="UTF-8"?> <gidl:GIDLModel xmi:version="2.0"
     xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:gidl="http://eu.chorevolution/modelingnotations/gidl"
hostAddress="128.93.64.1" protocol="WebSockets">
  <hasInterfaces role="provider">
    <hasOperations name="operation_1" type="one_way" qos="unreliable">
      <hasScope name="postTraffic" verb="" uri=""/>
      <outputData name="output" context="body">
        <hasDataType xsi:type="gidl:SimpleType" name="num_of_cars" occurences="one"
             type="integer"/>
        <hasDataType xsi:type="gidl:ComplexType" name="hazard_sit" occurences="one"/>
      </outputData>
    </hasOperations>
  </hasInterfaces>
</gidl:GIDLModel>
```

Listing B.2: XML representation of the `fixed-sensor` GIDL metamodel.

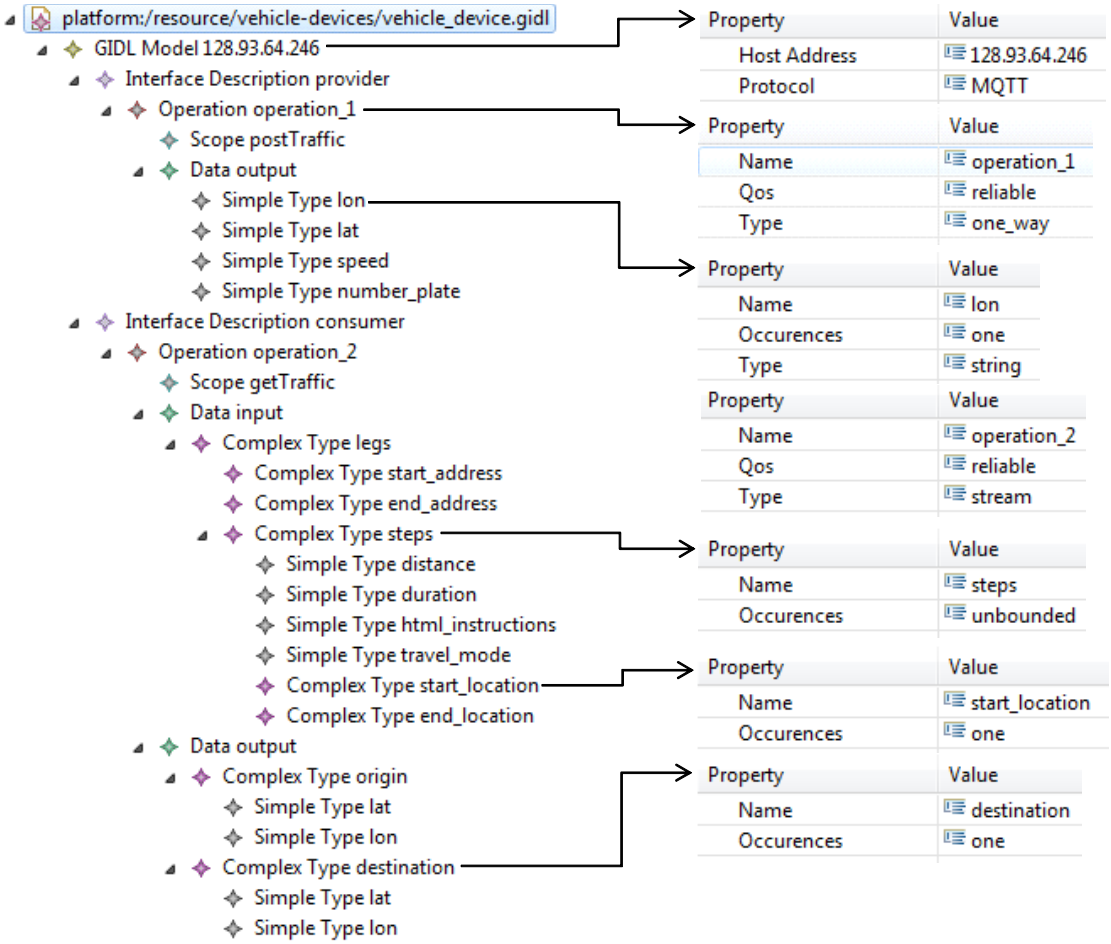


Figure B.4: GIDL model for `vehicle-devices`.

```
<?xml version="1.0" encoding="UTF-8"?> <gidl:GIDLModel xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gidl="http://eu.chorevolution/modelingnotations/gidl"
hostAddress="128.93.64.246" protocol="MQTT">
  <hasInterfaces role="provider">
    <hasOperations name="operation_1" type="one_way" qos="reliable">
      <hasScope name="postTraffic"/>
      <outputData name="output" context="body">
        <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
        <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
        <hasDataType xsi:type="gidl:SimpleType" name="speed" occurences="one" type="string"/>
        <hasDataType xsi:type="gidl:SimpleType" name="number_plate" occurences="one"
            type="string"/>
      </outputData>
    </hasOperations>
  </hasInterfaces>
  <hasInterfaces role="consumer">
    <hasOperations name="operation_2" type="stream" qos="reliable">
      <hasScope name="getTraffic"/>
      <inputData name="input" context="body">
        <hasDataType xsi:type="gidl:ComplexType" name="legs" occurences="unbounded">
          <hasDataType xsi:type="gidl:ComplexType" name="start_address" occurences="one"/>
```

```
21          <hasDataType xsi:type="gidl:ComplexType" name="end_address" occurences="one"/>
22          <hasDataType xsi:type="gidl:ComplexType" name="steps" occurences="unbounded">
23            <hasDataType xsi:type="gidl:SimpleType" name="distance" occurences="one"
                  type="string"/>
24            <hasDataType xsi:type="gidl:SimpleType" name="duration" occurences="one"
                  type="string"/>
25            <hasDataType xsi:type="gidl:SimpleType" name="html_instructions" occurences="one"
                  type="string"/>
26            <hasDataType xsi:type="gidl:SimpleType" name="travel_mode" occurences="one"
                  type="string"/>
27            <hasDataType xsi:type="gidl:ComplexType" name="start_location" occurences="one"/>
28            <hasDataType xsi:type="gidl:ComplexType" name="end_location" occurences="one"/>
29          </hasDataType>
30        </hasDataType>
31      </inputData>
32      <outputData name="output" context="body">
33        <hasDataType xsi:type="gidl:ComplexType" name="origin" occurences="one">
34          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
35          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
36        </hasDataType>
37        <hasDataType xsi:type="gidl:ComplexType" name="destination" occurences="one">
38          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
39          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
40        </hasDataType>
41      </outputData>
42    </hasOperations>
43  </hasInterfaces>
44 </gidl:GIDLModel>
```

Listing B.3: XML representation of the `vehicle-device` GIDL metamodel.

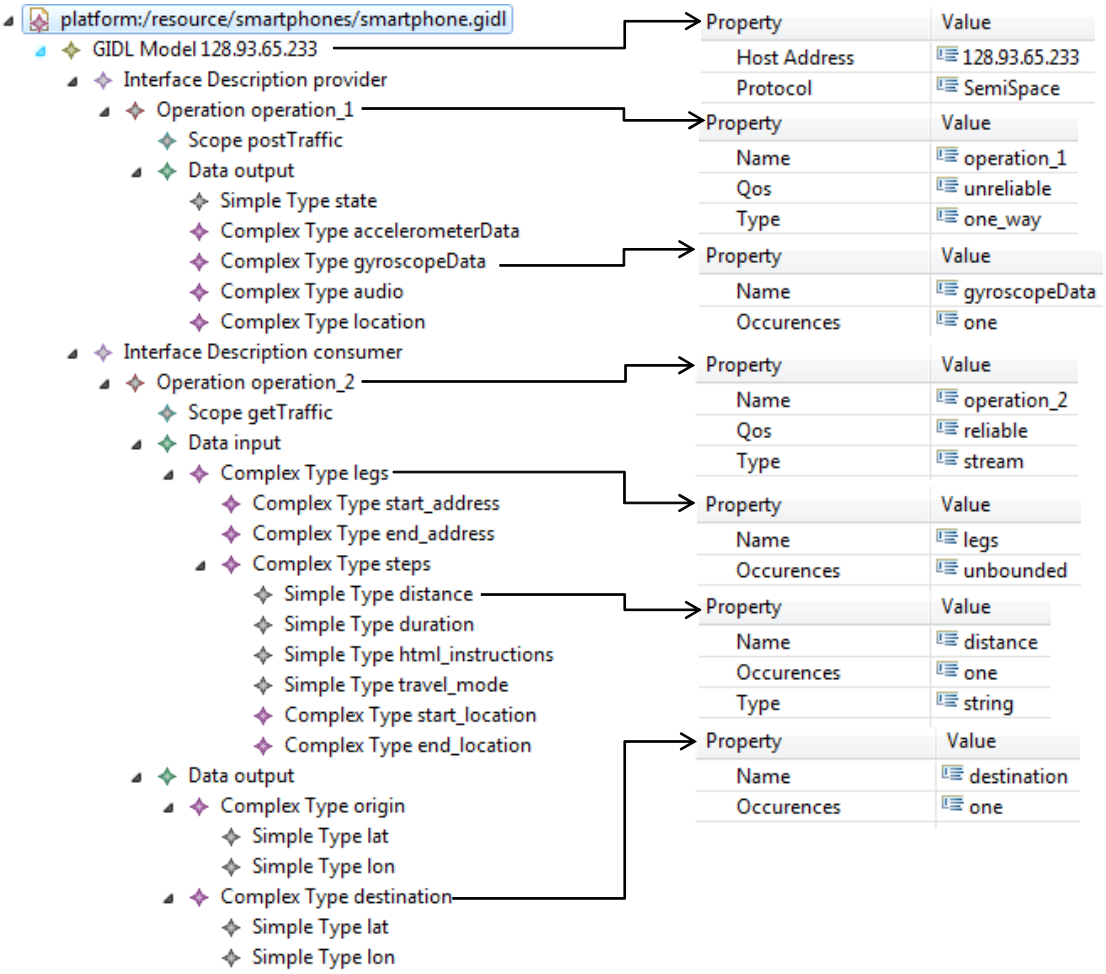


Figure B.5: GIDL model for `smartphones`.

```
<?xml version="1.0" encoding="UTF-8"?> <gidl:GIDLModel xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gidl="http://eu.chorevolution/modelingnotations/gidl"
hostAddress="128.93.65.233" protocol="SemiSpace">
  <hasInterfaces role="provider">
    <hasOperations name="operation_1" type="one_way" qos="unreliable">
      <hasScope name="postTraffic"/>
      <outputData name="output" context="body">
        <hasDataType xsi:type="gidl:SimpleType" name="state" occurences="one" type="string"/>
        <hasDataType xsi:type="gidl:ComplexType" name="accelerometerData" occurences="one"/>
        <hasDataType xsi:type="gidl:ComplexType" name="gyroscopeData" occurences="one"/>
        <hasDataType xsi:type="gidl:ComplexType" name="audio" occurences="one"/>
        <hasDataType xsi:type="gidl:ComplexType" name="location" occurences="one"/>
      </outputData>
    </hasOperations>
  </hasInterfaces>
  <hasInterfaces role="consumer">
    <hasOperations name="operation_2" type="two_way_sync" qos="reliable">
      <hasScope name="getTraffic"/>
      <inputData name="input" context="body">
        <hasDataType xsi:type="gidl:ComplexType" name="legs" occurences="unbounded">
```

```
21          <hasDataType xsi:type="gidl:ComplexType" name="start_address" occurences="one"/>
22          <hasDataType xsi:type="gidl:ComplexType" name="end_address" occurences="one"/>
23          <hasDataType xsi:type="gidl:ComplexType" name="steps" occurences="unbounded">
24            <hasDataType xsi:type="gidl:SimpleType" name="distance" occurences="one"
                  type="string"/>
25            <hasDataType xsi:type="gidl:SimpleType" name="duration" occurences="one"
                  type="string"/>
26            <hasDataType xsi:type="gidl:SimpleType" name="html_instructions" occurences="one"
                  type="string"/>
27            <hasDataType xsi:type="gidl:SimpleType" name="travel_mode" occurences="one"
                  type="string"/>
28            <hasDataType xsi:type="gidl:ComplexType" name="start_location" occurences="one"/>
29            <hasDataType xsi:type="gidl:ComplexType" name="end_location" occurences="one"/>
30          </hasDataType>
31        </hasDataType>
32      </inputData>
33      <outputData name="output" context="body">
34        <hasDataType xsi:type="gidl:ComplexType" name="origin" occurences="one">
35          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
36          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
37        </hasDataType>
38        <hasDataType xsi:type="gidl:ComplexType" name="destination" occurences="one">
39          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
40          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
41        </hasDataType>
42      </outputData>
43    </hasOperations>
44  </hasInterfaces>
45 </gidl:GIDLModel>
```

Listing B.4: XML representation of the `smartphone` GIDL metamodel.

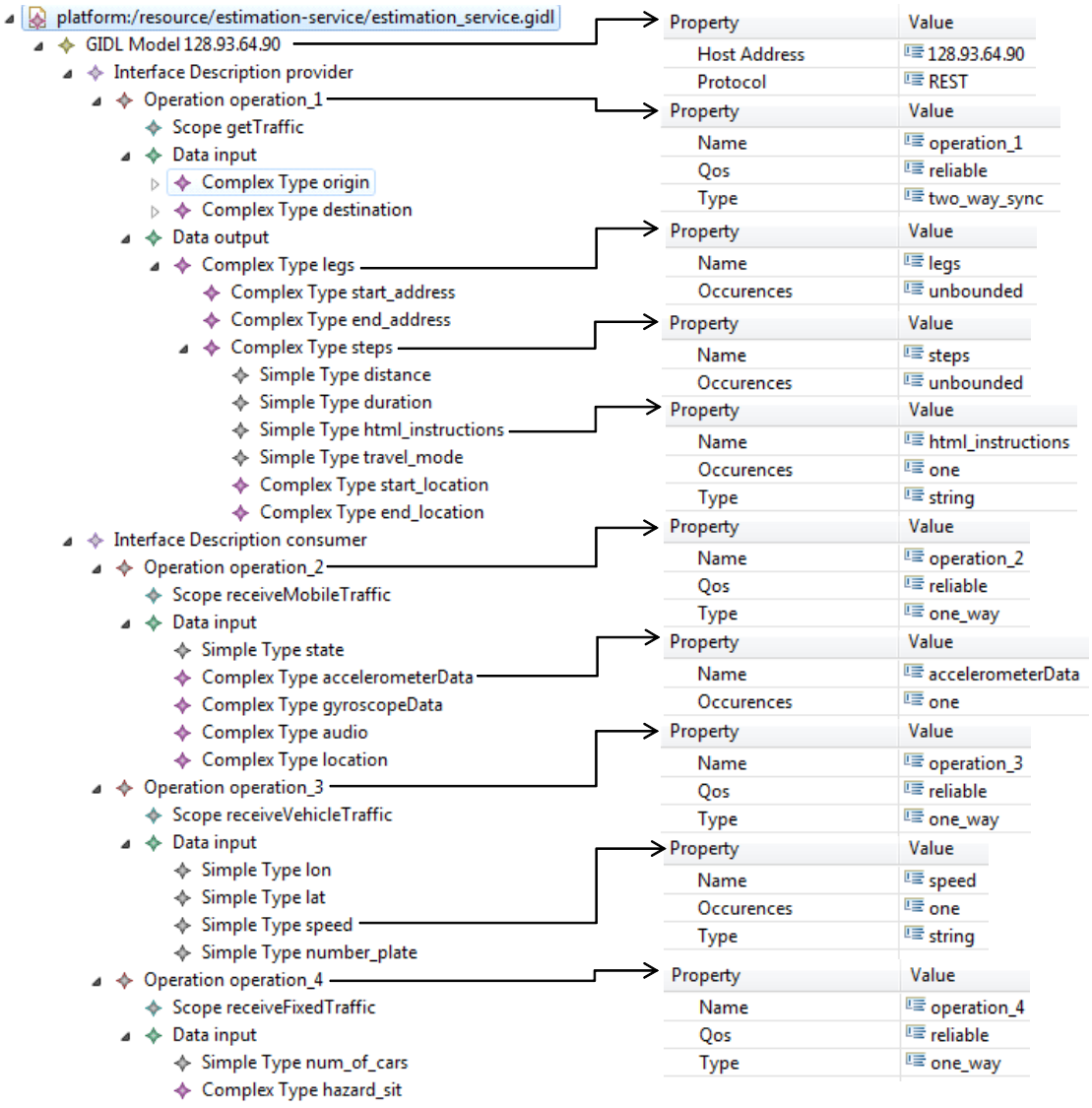


Figure B.6: GIDL model for the `estimation-service`.

```
<?xml version="1.0" encoding="UTF-8"?> <gidl:GIDLModel xmi:version="2.0"
     xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:gidl="http://eu.chorevolution/modelingnotations/gidl"
hostAddress="128.93.64.90" protocol="REST">
  <hasInterfaces role="provider">
    <hasOperations name="operation_1" type="two_way_sync" qos="reliable">
      <hasScope name="getTraffic"/>
      <inputData name="input" context="body">
        <hasDataType xsi:type="gidl:ComplexType" name="origin" occurences="one">
          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
        </hasDataType>
        <hasDataType xsi:type="gidl:ComplexType" name="destination" occurences="one">
```

```
13          <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
14          <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
15        </hasDataType>
16      </inputData>
17      <outputData name="output" context="body">
18        <hasDataType xsi:type="gidl:ComplexType" name="legs" occurences="unbounded">
19          <hasDataType xsi:type="gidl:ComplexType" name="start_address" occurences="one"/>
20          <hasDataType xsi:type="gidl:ComplexType" name="end_address" occurences="one"/>
21          <hasDataType xsi:type="gidl:ComplexType" name="steps" occurences="unbounded">
22            <hasDataType xsi:type="gidl:SimpleType" name="distance" occurences="one"
                  type="string"/>
23            <hasDataType xsi:type="gidl:SimpleType" name="duration" occurences="one"
                  type="string"/>
24            <hasDataType xsi:type="gidl:SimpleType" name="html_instructions" occurences="one"
                  type="string"/>
25            <hasDataType xsi:type="gidl:SimpleType" name="travel_mode" occurences="one"
                  type="string"/>
26            <hasDataType xsi:type="gidl:ComplexType" name="start_location" occurences="one"/>
27            <hasDataType xsi:type="gidl:ComplexType" name="end_location" occurences="one"/>
28          </hasDataType>
29        </hasDataType>
30      </outputData>
31    </hasOperations>
32  </hasInterfaces>
33  <hasInterfaces role="consumer">
34    <hasOperations name="operation_2" type="one_way" qos="reliable">
35      <hasScope name="receiveMobileTraffic" verb="POST" uri=""/>
36      <inputData name="input" context="body">
37        <hasDataType xsi:type="gidl:SimpleType" name="state" occurences="one" type="string"/>
38        <hasDataType xsi:type="gidl:ComplexType" name="accelerometerData" occurences="one"/>
39        <hasDataType xsi:type="gidl:ComplexType" name="gyroscopeData" occurences="one"/>
40        <hasDataType xsi:type="gidl:ComplexType" name="audio" occurences="one"/>
41        <hasDataType xsi:type="gidl:ComplexType" name="location" occurences="one"/>
42      </inputData>
43    </hasOperations>
44    <hasOperations name="operation_3" type="one_way" qos="reliable">
45      <hasScope name="receiveVehicleTraffic" verb="POST" uri=""/>
46      <inputData name="input" context="body">
47        <hasDataType xsi:type="gidl:SimpleType" name="lon" occurences="one" type="string"/>
48        <hasDataType xsi:type="gidl:SimpleType" name="lat" occurences="one" type="string"/>
49        <hasDataType xsi:type="gidl:SimpleType" name="speed" occurences="one" type="string"/>
50        <hasDataType xsi:type="gidl:SimpleType" name="number_plate" occurences="one"
              type="string"/>
51      </inputData>
52    </hasOperations>
53    <hasOperations name="operation_4" type="one_way" qos="reliable">
54      <hasScope name="receiveFixedTraffic" verb="POST"/>
55      <inputData name="input" context="body">
56        <hasDataType xsi:type="gidl:SimpleType" name="num_of_cars" occurences="one"
              type="integer"/>
57        <hasDataType xsi:type="gidl:ComplexType" name="hazard_sit" occurences="one"/>
58      </inputData>
59    </hasOperations>
60  </hasInterfaces>
61 </gidl:GIDLModel>
```

Listing B.5: XML representation of the `estimation-service` GIDL metamodel.

The models presented above, can be then provided to our framework to be parsed for synthesizing the corresponding Binding Components, as we describe in the next section.

B.3 BC Synthesis

To synthesize a BC, the GDIL model of the corresponding Thing must be provided to the *VSB Manager*. There are three possible ways to synthesize its interoperability artifact:

1. Through our Eclipse plugin, after specifying the Thing's GIDL model and through the selection "synthesize BC". The synthesized BC artifact is then located in the local directory of the Eclipse installation.

2. By incorporating the VSB Manager into a Java application using the `jar` package, which can be downloaded from here:

   `http://xsb.inria.fr/vsb.jar`

3. By using the Maven software project management and comprehension tool. Application developers can create a Maven project by incorporating the VSB Manager through our releases repository. Below we provide the required code script which must be included in the *project object model* (POM file) of the created Maven project:

```
<repositories>
  <repository>
    <id>ow2-nexus-snapshots</id>
    <url>http://repository.ow2.org/nexus/content/repositories/snapshots</url>
    <releases>
        <enabled>false</enabled>
    </releases>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>ow2-releases</id>
    <url>http://repository.ow2.org/nexus/content/repositories/releases</url>
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>eu.chorevolution.vsb</groupId>
    <artifactId>vsb-manager</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <classifier>jar-with-dependencies</classifier>
  </dependency>
</dependencies>
```

Listing B.6: Maven pom file configuration.

The VSB Manager can be used either through Maven or by including manually the `jar` file. To synthesize a new BC, we provide below a code script:

```
public static void main(String[] args) {
    String interfaceDescriptionPath = "PATH_TO_GIDL_FILE";
    generateWarBytes(interfaceDescriptionPath);
}

public static void generateWarBytes(String interfaceDescriptionPath) {

    byte[] interfaceDescriptionBytesArray = null;
    Path path = Paths.get(interfaceDescriptionPath);
    try {
        interfaceDescriptionBytesArray = Files.readAllBytes(path);
    } catch (IOException e) {
        e.printStackTrace();
    }
    VsbManager vsbm = new VsbManager();
    vsbm.generateWar(interfaceDescriptionBytesArray, ProtocolType.CoAP);
}
```

Listing B.7: BC code example generation.

It is worth noting that the VSB Manager accepts as input the `BytesArray` of the GIDL model and provides in the output a `war` artifact. This artifact can be then deployed into an *Apache Tomcat* server and configured through the `BC Manager`.

The software described here is freely available and can be downloaded by the instructions above. Any suggestions for improvements, bug reports etc, will be gratefully received; please email all feedback to `boulouk@gmail.com` (Georgios Bouloukakis).

Appendix C

MobileJINQS Simulator

This appendix relates to the contributions presented in Chapter 5 where we presented our performance modeling patterns. These patterns can be utilized from system designers to model the performance of their applications. Such applications may employ different middleware protocols following our core communication styles (i.e., CS, PS, DS and TS). As already pointed out, we have developed a simulator (MobileJINQS) that implements our queueing patterns. MobileJINQS is an extension of JINQS, a Java simulation library for multiclass queueing networks. Based on the user guide of JINQS [184], a queueing network is a network of `Nodes`. Any introduced queueing center is implemented as a subclass of the `Nodes` superclass. The queueing centers can be assembled to form a queueing network through the class `Network`, which is responsible for setting up a queueing network using:

```
Network.initialise();
```

`Source` nodes can be used to inject messages into a network with a specified inter-arrival time distribution. For instance, messages arrive based on a Poisson process with a specific arrival rate through:

```
Source source = new Source(new Exp(4));
```

To insert the arrival rates of messages into a queue and serve them through a server, there are particular components named `Queueing` nodes. A `QueueingNode` queues waiting messages in FIFO order by default and it supports their service through one or more servers. For instance:

```
Delay servTime = new Delay(new Exp(2));
QueueingNode qNode = new QueueingNode("QNode", servTime, 3);
```

which models a 3-server queueing node with exponentially-distributed service times (`servTime`) with rate 2 per server. The above nodes can be connected through the `Link` class. For instance, to insert the messages generated from the above `source` node into the above `qNode`, we type:

```
source.setLink(new Link(qNode));
```

Finally, there is the possibility to separate some traffic of messages through a `ProbabilisticBranch` which routes messages to the defined nodes depending on associated probabilities. For instance, based on the following branch `pb`:

```
double[] probs = new double[] {0.3, 0.7};
Node[] nodes = new Node[] {exit, node};
ProbabilisticBranch pb = new ProbabilisticBranch(probs, nodes);
```

with regard to the overall traffic arrived in `pb`, 30% of messages will enter `exit` and 70% will enter `node`. More details regarding the creation of multi-class queuing networks can be found in [184].

In this thesis, we retain the generic specification of JINQS and we provide additional features related to the mobile IoT. More specifically, upon the creation of a new message and prior to its insertion into the queueing network, a `lifetime` (or `timeout`) parameter can be applied though the following:

```
Deterministic lifetime = new Deterministic(10);
Source source = = new Source("Source", new Exp(2), lifetime, "lifetime");
```

where the lifetime period is deterministic. To represent the mobile Things' behavior, we introduce additional `Queueing` nodes based on the underlying middleware infrastructure. In reliable infrastructures messages may insert the queue and during the Thing's disconnection, messages wait in the queue to be processed. We represent such behavior through the `OnOffQN` node:

```
OnOffRQN onoffRnode = new OnOffRQN("onoffRnode", servTime, 1, on, off);
```

where the `on`, `off` parameters introduce the online/offline connectivity periods generated through exponential distributions. On the other hand, for unreliable infrastructures messages may insert the queue and during the Thing's disconnection, they exit the queueing center considered as lost. We represent such behavior through the `OnOffUnreliableQN` node:

```
OnOffUQN onoffUNode = new OnOffUQN("onoffUNode", servTime, 1, on, off);
```

To estimate the losses due to lifetime expirations or middleware disconnections, we introduce additional `Sink` nodes which can be utilized inside a queueing network:

```
SinkLftLses sinkL = new SinkLftLses("SinkLifetime");
SinkMdwLses sinkM = new SinkMdwLses("SinkMdw");
SinkBothLses sinkB = new SinkBothLses("SinkBoth");
```

Furthermore, we introduce particular branchers that are not probabilistic as they separate the traffic based on the expiration status of messages:

```
LftLsesBranch bl = new LftLsesBranch(new Node[] {sinkL, qNode});
MdwLsesBranch bm = new MdwLsesBranch(new Node[] {sinkM, onoffRnode});
BothLsesBranch bb = new BothLsesBranch(new Node[] {sinkB, onoffUNode});
```

Finally, the `FileDataSet` node allow us to derive connectivity periods and inter-arrival times from real datasets:

```
FileDataSet trace = new FileDataSet("data_file.txt");
Source source = new Source( "Source", trace, "Dataset");
```

The MobileJINQS simulator can be downloaded from here:

`http://xsb.inria.fr/MobileJINQS.jar`

which can be utilized from design designers for system tuning by applying lifetime periods, ON/OFF intervals, etc. The software described here is freely available and can be downloaded by the instructions above. Any suggestions for improvements, bug reports etc, will be gratefully received; please email all feedback to `boulouk@gmail.com` (Georgios Bouloukakis).

In the next sections (C.1,C.2 and C.3) we provide the implementation for some of our performance patterns using MobileJINQS. Particularly, we implement Patterns 1,2 and 4. The remaining patterns can be implemented in a similar way.

```
1  public class Params {
2  // general parameters
3      public static double LIFETIME = 10;
4      public static double TIMEOUT = 10;
5      public static double SRC_RATE = 2;
6      public static double PR_MSG_RATE = 64;
7      public static double TR_MSG_RATE = 16;
8      public static double ON_OVRL_RATE = 0.1;
9      public static double OFF_OVRL_RATE = 0.1;
10
11 //  parameters for pattern 1
12     public static double ON_PROD_RATE = 0.1;
13     public static double OFF_PROD_RATE = 0.1;
14     public static double ON_CON_RATE = 0.1;
15     public static double OFF_CON_RATE = 0.1;
16
17 //  parameters for pattern 1,2
18     public static double ON_MDW_RATE = 0.1;
19     public static double OFF_MDW_RATE = 0.1;
20
21 //  parameters for pattern 2
22     public static double PR_REQ_RATE = 128;
23     public static double TR_REQ_RATE = 32;
24     public static double PR_SERVER_APP_RATE = 16;
25     public static double ON_CL_RATE = 0.1;
26     public static double OFF_CL_RATE = 0.1;
27     public static double ON_SER_RATE = 0.1;
28     public static double OFF_SER_RATE = 0.1;
29
30 //  parameters for pattern 4
31     public static double ON_PUB_RATE = 0.1;
32     public static double OFF_PUB_RATE = 0.1;
33     public static double ON_SUB_RATE = 0.1;
34     public static double OFF_SUB_RATE = 0.1;
35     public static double ON_MDW_LINK1_RATE = 0.1;
36     public static double OFF_MDW_LINK1_RATE = 0.1;
37     public static double ON_MDW_LINK2_RATE = 0.1;
38     public static double OFF_MDW_LINK2_RATE = 0.1;
39     public static double ON_OVRL_PUB_BR_RATE = 0.1;
40     public static double OFF_OVRL_PUB_BR_RATE = 0.1;
41     public static double ON_OVRL_BR_SUB_RATE = 0.1;
42     public static double OFF_OVRL_BR_SUB_RATE = 0.1;
43 }
```

Listing C.1: Parameters used in patterns for simulation.

In listing C.1 we provide a set of parameters used to tune the patterns below. A system designer is able to change their parameters, run the simulation models and get the estimated

performance of each pattern.

C.1 Pattern 1 Simulation Constructors

```
public Pattern1UnreliableSim(double dur) {

    Network.initialise();

// deterministic lifetime parameter
    Deterministic lifetime = new Deterministic(Params.LIFETIME);
// arrival rate of messages
    Source src = new Source("Source", new Exp(Params.SRC_RATE), lifetime, "lifetime");
// parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
// parameters for the producer's app ON/OFF periods
    Exp ONProd = new Exp(Params.ON_PROD_RATE);
    Exp OFFProd = new Exp(Params.OFF_PROD_RATE);
// parameters for the middleware link ON/OFF periods
    Exp ONMdw = new Exp(Params.ON_MDW_RATE);
    Exp OFFMdw = new Exp(Params.OFF_MDW_RATE);
// parameters for the consumer's app ON/OFF periods
    Exp ONCon = new Exp(Params.ON_CON_RATE);
    Exp OFFCon = new Exp(Params.OFF_CON_RATE);

// queueing centers setup
    OnOffQN prod_app = new OnOffQN("PROD-APP", prMsg, 1, ONProd, OFFProd, dur);
    OnOffUnreliableQN prod_mdw = new OnOffUnreliableQN("PROD-MDW", trMsg, 1, ONMdw, OFFMdw, dur);
    OnOffUnreliableQN con_mdw = new OnOffUnreliableQN("CON-MDW", prMsg, 1, ONCon, OFFCon, dur);
    QueueingNode con_app = new QueueingNode("CON-MDW", prMsg, 1);

// sink centers setup
    SinkLftLses sinkProdApp = new SinkLftLses("SINK-PROD-APP");
    SinkMdwLses sinkProdMdw = new SinkMdwLses("SINK-PROD-MDW");
    SinkMdwLses sinkConMdw = new SinkMdwLses("SINK-CON-MDW");
    SinkOvrlNet sinkConEnd = new SinkOvrlNet("SINK-CON-END");

// branches setup
    LftLsesBranch branchProdApp = new LftLsesBranch(new Node[] {sinkProdApp, prod_mdw});
    MdwLsesBranch branchProdMdw = new MdwLsesBranch(new Node[] {sinkProdMdw, con_mdw}) ;
    MdwLsesBranch branchConMdw = new MdwLsesBranch(new Node[] {sinkConMdw, con_app}) ;

// queueing network creation
    src.setLink(new Link(prod_app));
    prod_app.setLink(branchProdApp);
    prod_mdw.setLink(branchProdMdw);
    con_mdw.setLink(branchConMdw);
    con_app.setLink(new Link(sinkConEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.2: Pattern 1 (unreliable) simulation constructor.

```
public Pattern1ReliableSim(double dur) {

    Network.initialise();

//  deterministic lifetime parameter
    Deterministic lifetime = new Deterministic(Params.LIFETIME);
//  arrival rate of messages
    Source src = new Source("Source", new Exp(Params.SRC_RATE), lifetime, "lifetime");
//  parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
//  parameters for the end-to-end ON/OFF periods
    Exp ONOvrl = new Exp(Params.ON_OVRL_RATE);
    Exp OFFOvrl = new Exp(Params.OFF_OVRL_RATE);

//  queueing centers setup
    OnOffQN prod_app = new OnOffQN("PROD-APP", prMsg, 1, ONOvrl, OFFOvrl, dur);
    QueueingNode prod_mdw = new QueueingNode("PRO-MDW", trMsg, 1);
    QueueingNode con_mdw = new QueueingNode("CON-MDW", prMsg, 1);
    QueueingNode con_app = new QueueingNode("CON-MDW", prMsg, 1);

//  sink centers setup
    SinkLftLses sinkProdApp = new SinkLftLses("SINK-PROD-APP");
    SinkOvrlNet sinkConEnd = new SinkOvrlNet("SINK-CON-END");

//  branches setup
    LftLsesBranch branchProdApp = new LftLsesBranch(new Node[] {sinkProdApp, prod_mdw});

//  queueing network creation
    src.setLink(new Link(prod_app));
    prod_app.setLink(branchProdApp);
    prod_mdw.setLink(new Link(con_mdw));
    con_mdw.setLink(new Link(con_app));
    con_app.setLink(new Link(sinkConEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.3: Pattern 1 (reliable) simulation constructor.

C.2 Pattern 2 Simulation Constructors

```
public Pattern2UnreliableSim(double dur) {

    Network.initialise();

// deterministic lifetime parameter
    Deterministic timeout = new Deterministic(Params.TIMEOUT);
// arrival rate of messages
    Source source = new Source("Source", new Exp(Params.SRC_RATE), timeout, "timeout");
// parameters for processing and transmitting requests
    Delay prReq = new Delay(new Exp(Params.PR_REQ_RATE));
    Delay trReq = new Delay(new Exp(Params.TR_REQ_RATE));
// parameter for processing requests and proving the response on the server side
    Delay prServerApp = new Delay(new Exp(Params.PR_SERVER_APP_RATE));
// parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
// parameters for the client's app ON/OFF periods
    Exp ONCl = new Exp(Params.ON_CL_RATE);
    Exp OFFCl = new Exp(Params.OFF_CL_RATE);
// parameters for the middleware link ON/OFF periods
    Exp ONMdw = new Exp(Params.ON_MDW_RATE);
    Exp OFFMdw = new Exp(Params.OFF_MDW_RATE);
// parameters for the server's app ON/OFF periods
    Exp ONSer = new Exp(Params.ON_SER_RATE);
    Exp OFFSer = new Exp(Params.OFF_SER_RATE);

// queueing centers setup for the requests transmission to the server
    OnOffQN cl_app1 = new OnOffQN("CL-APP-1", prReq, 1, ONCl, OFFCl, dur);
    OnOffUnreliableQN cl_mdw1 = new OnOffUnreliableQN("CL-MDW-1", trReq, 1, ONMdw, OFFMdw, dur);
    OnOffUnreliableQN ser_mdw1 = new OnOffUnreliableQN("SER-MDW-1", prReq, 1, ONSer, OFFSer, dur);
// queueing center setup for the requests processing and proving the responses
    QueueingNode ser_app = new QueueingNode("SER-MDW-1", prServerApp, 1);
// queueing centers setup for the messages transmission to the client
    OnOffUnreliableQN ser_mdw2 = new OnOffUnreliableQN("SER-MDW-2", trMsg, 1, ONMdw, OFFMdw, dur);
    QueueingNode cl_mdw2 = new QueueingNode("CL-MDW-2", prMsg, 1);
    QueueingNode cl_app2 = new QueueingNode("CL-APP-2", prMsg, 1);

// sink centers setup for requests
    SinkLftLses sinkClApp1 = new SinkLftLses("SINK-CL-APP-1");
    SinkMdwLses sinkClMdw1 = new SinkMdwLses("SINK-CL-MDW-1");
    SinkMdwLses sinkSerMdw1 = new SinkMdwLses("SINK-SER-MDW-1");
    SinkLftLses sinkSerApp1 = new SinkLftLses("SINK-SER-APP-1");
// sink centers setup for messages
    SinkMdwLses sinkSerMdw2 = new SinkMdwLses("SINK-SER-MDW-2");
    SinkOvrlNet sinkClEnd = new SinkOvrlNet("SINK-CL-END");

// branches setup for requests
    LftLsesBranch branchClApp1 = new LftLsesBranch(new Node[] {sinkClApp1, cl_mdw1});
    MdwLsesBranch branchClMdw1 = new MdwLsesBranch(new Node[] {sinkClMdw1, ser_mdw1});
    MdwLsesBranch branchSerMdw1 = new MdwLsesBranch(new Node[] {sinkSerMdw1, ser_app});
    LftLsesBranch branchSerApp1 = new LftLsesBranch(new Node[] {sinkSerApp1, ser_mdw2});
// branches setup for messages
    MdwLsesBranch branchSerMdw2 = new MdwLsesBranch(new Node[] {sinkSerMdw2, cl_mdw2});

// queueing network creation
    source.setLink(new Link(cl_app1));
    cl_app1.setLink(branchClApp1);
    cl_mdw1.setLink(branchClMdw1);
    ser_mdw1.setLink(branchSerMdw1);
    ser_app.setLink(branchSerApp1);
    ser_mdw2.setLink(branchSerMdw2);
    cl_mdw2.setLink(new Link(cl_app2));
    cl_app2.setLink(new Link(sinkClEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.4: Pattern 2 (unreliable) simulation constructor.

```
public Pattern2ReliableSim(double dur) {

    Network.initialise();

//  deterministic lifetime parameter
    Deterministic timeout = new Deterministic(Params.TIMEOUT);
//  arrival rate of messages
    Source source = new Source("Source", new Exp(Params.SRC_RATE), timeout, "timeout");
//  parameters for processing and transmitting requests
    Delay prReq = new Delay(new Exp(Params.PR_REQ_RATE));
    Delay trReq = new Delay(new Exp(Params.TR_REQ_RATE));
//  parameter for processing requests and proving the response on the server side
    Delay prServerApp = new Delay(new Exp(Params.PR_SERVER_APP_RATE));
//  parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
//  parameters for the end-to-end ON/OFF periods
    Exp ONOvrl = new Exp(Params.ON_OVRL_RATE);
    Exp OFFOvrl = new Exp(Params.OFF_OVRL_RATE);

//  queueing centers setup for the requests transmission to the server
    OnOffQN cl_app1 = new OnOffQN("CL-APP-1", prReq, 1, ONOvrl, OFFOvrl, duration);
    QueueingNode cl_mdw1 = new QueueingNode("CL-MDW-1", trReq, 1);
    QueueingNode ser_mdw1 = new QueueingNode("SER-MDW-1", prReq, 1);
//  queueing center setup for the requests processing and proving the responses
    QueueingNode ser_app = new QueueingNode("SER-MDW-1", prServerApp, 1);
//  queueing centers setup for the messages transmission to the client
    OnOffQN ser_mdw2 = new OnOffQN("SER-MDW-2", trMsg, 1, ONOvrl, OFFOvrl, duration);
    QueueingNode cl_mdw2 = new QueueingNode("CL-MDW-2", prMsg, 1);
    QueueingNode cl_app2 = new QueueingNode("CL-APP-2", prMsg, 1);

//  sink centers setup for requests
    SinkLftLses sinkClApp1 = new SinkLftLses("SINK-CL-APP-1");
    SinkLftLses sinkSerApp1 = new SinkLftLses("SINK-SER-APP-1");
//  sink centers setup for messages
    SinkOvrlNet sinkClEnd = new SinkOvrlNet("SINK-CL-END");

//  branches setup for requests
    LftLsesBranch branchClApp1 = new LftLsesBranch(new Node[] {sinkClApp1, cl_mdw1});
    LftLsesBranch branchSerApp1 = new LftLsesBranch(new Node[] {sinkSerApp1, ser_mdw2});

//  queueing network creation
    source.setLink(new Link(cl_app1));
    cl_app1.setLink(branchClApp1);
    cl_mdw1.setLink(new Link(ser_mdw1));
    ser_mdw1.setLink(new Link(ser_app));
    ser_app.setLink(branchSerApp1);
    ser_mdw2.setLink(new Link(cl_mdw2));
    cl_mdw2.setLink(new Link(cl_app2));
    cl_app2.setLink(new Link(sinkClEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.5: Pattern 2 (reliable) simulation constructor.

C.3 Pattern 4 Simulation Constructors

```
public Pattern4UnreliableSim(double dur) {

    Network.initialise();

// deterministic lifetime parameter
    Deterministic lifetime = new Deterministic(Params.LIFETIME);
// arrival rate of messages
    Source src = new Source("Source", new Exp(Params.SRC_RATE), lifetime, "lifetime");
// parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
// parameters for the publisher's app ON/OFF periods
    Exp ONPub = new Exp(Params.ON_PUB_RATE);
    Exp OFFPub = new Exp(Params.OFF_PUB_RATE);
// parameters for the middleware link 1 (publisher -> broker) ON/OFF periods
    Exp ONMdwLink1 = new Exp(Params.ON_MDW_LINK1_RATE);
    Exp OFFMdwLink1 = new Exp(Params.OFF_MDW_LINK1_RATE);
// parameters for the middleware link 2 (broker -> subscriber) ON/OFF periods
    Exp ONMdwLink2 = new Exp(Params.ON_MDW_LINK2_RATE);
    Exp OFFMdwLink2 = new Exp(Params.OFF_MDW_LINK2_RATE);
// parameters for the subscriber's app ON/OFF periods
    Exp ONSub = new Exp(Params.ON_SUB_RATE);
    Exp OFFSub = new Exp(Params.OFF_SUB_RATE);

// queueing centers setup
    OnOffQN pub_app = new OnOffQN("PUB-APP", prMsg, 1, ONPub, OFFPub, dur);
    OnOffUnreliableQN pub_mdw = new OnOffUnreliableQN("PUB-MDW", trMsg, 1, ONMdwLink1,
        OFFMdwLink1, dur);
    QueueingNode br_in = new QueueingNode("BR-IN", prMsg, 1);
    OnOffUnreliableQN br_out = new OnOffUnreliableQN("BR-OUT", trMsg, 1, ONMdwLink2, OFFMdwLink2,
        dur);
    OnOffUnreliableQN sub_mdw = new OnOffUnreliableQN("SUB-MDW", prMsg, 1, ONSub, OFFSub, dur);
    QueueingNode sub_app = new QueueingNode("SUB-APP", prMsg, 1);

// sink centers setup
    SinkLftLses sinkPubApp = new SinkLftLses("SINK-PUB-APP");
    SinkMdwLses sinkPubMdw = new SinkMdwLses("SINK-PUB-MDW");
    SinkLftLses sinkBrIn = new SinkLftLses("SINK-BR-IN");
    SinkBothLses sinkBrOut = new SinkBothLses("SINK-BR-OUT");
    SinkMdwLses sinkSubMdw = new SinkMdwLses("SINK-SUB-MDW");
    SinkOvrlNet sinkSubEnd = new SinkOvrlNet("SINK-SUB-END");

// branches setup
    LftLsesBranch branchPubApp = new LftLsesBranch(new Node[] {sinkPubApp, pub_mdw});
    MdwLsesBranch branchPubMdw = new MdwLsesBranch(new Node[] {sinkPubMdw, br_in});
    LftLsesBranch branchBrIn = new LftLsesBranch(new Node[] {sinkBrIn, br_out});
    BothLsesBranch branchBrOut = new BothLsesBranch(new Node[] {sinkBrOut, sub_mdw});
    MdwLsesBranch branchSubMdw = new MdwLsesBranch(new Node[] {sinkSubMdw, sub_app});

// queueing network creation
    src.setLink(new Link(pub_app));
    pub_app.setLink(branchPubApp);
    pub_mdw.setLink(branchPubMdw);
    br_in.setLink(branchBrIn);
    br_out.setLink(branchBrOut);
    sub_mdw.setLink(branchSubMdw);
    sub_app.setLink(new Link(sinkSubEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.6: Pattern 4 (unreliable) simulation constructor.

```
public Pattern4ReliableSim(double dur) {

    Network.initialise();

//  deterministic lifetime parameter
    Deterministic lifetime = new Deterministic(Params.LIFETIME);
//  arrival rate of messages
    Source src = new Source("Source", new Exp(Params.SRC_RATE), lifetime, "lifetime");
//  parameters for processing and transmitting messages
    Delay prMsg = new Delay(new Exp(Params.PR_MSG_RATE));
    Delay trMsg = new Delay(new Exp(Params.TR_MSG_RATE));
//  parameters for the end-to-end 1 (publisher -> broker) ON/OFF periods
    Exp ONOvrlPubBr = new Exp(Params.ON_OVRL_PUB_BR_RATE);
    Exp OFFOvrlPubBr = new Exp(Params.OFF_OVRL_PUB_BR_RATE);
//  parameters for the end-to-end 2 (broker -> subscriber) ON/OFF periods
    Exp ONOvrlBrSub = new Exp(Params.ON_OVRL_BR_SUB_RATE);
    Exp OFFOvrlBrSub = new Exp(Params.OFF_OVRL_BR_SUB_RATE);

//  queueing centers setup
    OnOffQN pub_app = new OnOffQN("PUB-APP", prMsg, 1, ONOvrlPubBr, OFFOvrlPubBr, dur);
    QueueingNode pub_mdw = new QueueingNode("PUB-MDW", trMsg, 1);
    QueueingNode br_in = new QueueingNode("BR-IN", prMsg, 1);
    OnOffQN br_out = new OnOffQN("BR-OUT", trMsg, 1, ONOvrlBrSub, OFFOvrlBrSub, dur);
    QueueingNode sub_mdw = new QueueingNode("SUB-MDW", prMsg, 1);
    QueueingNode sub_app = new QueueingNode("SUB-APP", prMsg, 1);

//  sink centers setup
    SinkLftLses sinkPubApp = new SinkLftLses("SINK-PUB-APP");
    SinkLftLses sinkBrIn = new SinkLftLses("SINK-BR-IN");
    SinkLftLses sinkBrOut = new SinkLftLses("SINK-BR-OUT");
    SinkOvrlNet sinkSubEnd = new SinkOvrlNet("SINK-SUB-APP");

//  branches setup
    LftLsesBranch branchPubApp = new LftLsesBranch(new Node[] {sinkPubApp, pub_mdw});
    LftLsesBranch branchBrIn = new LftLsesBranch(new Node[] {sinkBrIn, br_out});
    LftLsesBranch branchBrOut = new LftLsesBranch(new Node[] {sinkBrOut, sub_mdw});

//  queueing network creation
    src.setLink(new Link(pub_app));
    pub_app.setLink(branchPubApp);
    pub_mdw.setLink(new Link(br_in));
    br_in.setLink(branchBrIn);
    br_out.setLink(branchBrOut);
    sub_mdw.setLink(new Link(sub_app));
    sub_app.setLink(new Link(sinkSubEnd));

    simulate();
    Network.displayResults(0.01);
}
```

Listing C.7: Pattern 4 (reliable) simulation constructor.